



3RD EDITION

How to Build Android Applications with Kotlin

A hands-on guide to developing, testing, and publishing production-grade Android 16 apps

ALEX FORRESTER | ERAN BOUDJNAH
ALEXANDRU DUMBRAVAN | JOMAR TIGCAL

How to Build Android Applications with Kotlin

Third Edition

A hands-on guide to developing, testing, and publishing production-grade Android 16 apps

Alex Forrester

Eran Boudjnah

Alexandru Dumbravan

Jomar Tigcal



How to Build Android Applications with Kotlin

Third Edition

Copyright © 2025 Packt Publishing

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, except in the case of brief 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 authors, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Portfolio Director: Ashwin Nair

Relationship Lead: Sohini Ghosh

Program Manager: Ruvika Rao

Content Engineer: Adrija Mitra

Technical Editor: Rohit Singh

Copy Editor: Safis Editing

Indexer: Pratik Shiroadkar

Proofreader: Adrija Mitra

Production Designer: Alishon Falcon

Growth Lead: Sohini Ghosh

First published: February 2021

Second edition: May 2023

Third edition: September 2025

Production reference: 1250825

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul's Square

Birmingham

B3 1RB, UK.

ISBN 978-1-83588-276-4

www.packtpub.com

Dedicated to my wife, Angela, and daughter, Catherine, for all their love and support.

– Alex Forrester

To my endlessly supportive wife, Lea, for always being there.

– Eran Boudjnah

Dedicated to Niki, for her constant support.

– Alexandru Dumbravan

*To my loving wife, Celine, for her support and encouragement. To my parents, for all their sacrifices
and for raising me well.*

– Jomar Tigcal

Contributors

About the authors

Alex Forrester is an experienced software developer with more than 20 years of experience in mobile and web development and content management systems. He has worked with Android since 2010, creating flagship apps for blue-chip companies across a broad range of industries at Sky, The Automobile Association, HSBC, Discovery Channel, NatWest, and O2. Alex lives in Hertfordshire with his wife and daughter. When he's not developing, he likes playing touch rugby and running in the Chiltern Hills.

Eran Boudjnah has been developing apps and leading mobile teams for a wide range of clients, from start-ups (Just Eat and Plume Design) to large-scale companies (Sky and HSBC) and conglomerates, since 1997. He has been working with Android since around 2013. Eran is a developer with almost three decades of experience in developing mobile applications, websites, desktop applications, and interactive attractions. He is passionate about board games and has a 1990s Transformers collection that he's quite proud of. Eran lives in Brentwood, Essex, in the United Kingdom, with Lea, his incredibly supportive wife.

Alexandru Dumbravan is an Android developer with more than 10 years of experience in building Android apps, focusing on fintech applications since 2016 when he moved to London. In his spare time, Alex enjoys playing video games, watching movies, and occasionally visiting the gym.

Jomar Tigcal is an Android developer with over 15 years of experience in mobile and software development. He has worked on various stages of Android app development for small start-ups and large companies since 2012. Jomar has also given talks and conducted training and workshops on Android. In his free time, he likes running and reading. He lives in Vancouver, BC, Canada, with his wife, Celine.

About the reviewers

Swamy Gangadhar Pavan Vulisetti is a senior Android engineer with extensive experience in the healthcare, travel, and IoT industries. He specializes in designing scalable, high-performance mobile solutions and has a strong command of modern Android frameworks and SDK integrations. Pavan's exceptional computer science skills allow him to craft efficient algorithms and maintain robust code architecture. His expertise lies in building seamless user experiences while following industry best practices. With a passion for innovation and problem-solving, he is committed to delivering impactful, reliable software solutions that meet complex business challenges.

Hema Sai Charan Kothamasu also known as *hemAndroid*, is a mobile app developer with over 11 years of experience in Android, Flutter, and iOS native development. In 2013, after completing his studies, he started his career at a start-up, working on experimental Android projects. Over time, he transitioned from a solo developer to an active contributor to the global developer community. In 2018, inspired by Google Developer Groups, he started giving insightful talks, which transformed his career. Now a sought-after mentor and technical speaker, he guides aspiring developers through engaging workshops and events. Hema is a co-organizer of the Flutter Hyderabad community and has contributed as a reviewer to *Thriving in Android Development Using Kotlin*.

Gajendra Singh Rathore (also known as *gsrathoreniks*) is a dedicated mobile app developer with years of expertise in creating seamless and high-performing Android applications. Known for his meticulous attention to design and performance, Gajendra specializes in crafting apps that prioritize user experience. A passionate advocate for knowledge sharing, he actively contributes to the developer community through his insightful articles and blogs on cutting-edge tech topics. Gajendra is also a mentor to budding developers, helping them navigate the complexities of modern app development. An avid reader of tech articles and an open source enthusiast, Gajendra's commitment to learning and teaching is reflected in every project he undertakes.

Peter Gichia is a freelance software engineer focused on native Android development and an entrepreneur currently working on a self-checkout system for the retail sector, among other personal projects. He enjoys solving problems for his clients, whether through code or business strategy. He is also an active contributor to the Android development community through writing and publishing Android-related articles. In the process, he has successfully published a text-based course on building scalable applications with MVVM architecture in collaboration with a leading Edutech company. In his free time, Peter enjoys expanding his knowledge through podcasts and books.

Table of Contents

Preface	xix

Part 1: Android Foundation	1

Chapter 1: Creating Your First App	3

Getting the most out of this book – get to know your free benefits	4

Next-gen reader • 4

Interactive AI assistant (beta) • 5

DRM-free PDF or ePub version • 5

Technical requirements	6

Creating an Android project with Android Studio	6

Exercise 1.01 – Creating an Android Studio project for your app • 7

Setting up a virtual device and running your app	10

Exercise 1.02 – Setting up a virtual device and running your app on it • 12

The Android manifest file	17

Exercise 1.03 – Configuring the Android manifest internet permission • 20

Using Gradle to build, configure, and manage app dependencies	25

The project-level build.gradle.kts file • 25

The app-level build.gradle.kts file • 26

Exercise 1.04 – Exploring how Material Design in Jetpack Compose is used to theme an app • 32

Android application structure	37

Exercise 1.05 – Building a Jetpack Compose UI to display a bespoke greeting to the user • 44

Activity 1.01 – Producing an app to create RGB colors • 51

Summary	53

Chapter 2: Building User Screen Flows	55
Technical requirements	56
The activity lifecycle	56
Exercise 2.01 – logging activity callbacks • 59	
Saving and restoring the activity state	66
Exercise 2.02 – saving and restoring the state • 66	
Exercise 2.03 – saving and restoring state in Compose • 71	
Activity interaction with intents	73
Exercise 2.04 – an introduction to intents • 73	
Exercise 2.05 – retrieving a result from an activity • 82	
Intents, tasks, and launch modes	94
Exercise 2.06 – setting the launch mode of an activity • 95	
Activity 2.01 and/or Activity 2.02 – creating a login form • 99	
Summary	99
Chapter 3: Developing the UI with Jetpack Compose	101
Technical requirements	102
Transitioning from XML layouts to Jetpack Compose	102
Exercise 3.01 – Creating a counter app with legacy views • 102	
Exercise 3.02 – Creating an Android app with Compose • 106	
Essential composable functions	109
Text • 109	
Button • 111	
Icon • 112	
Image • 112	
Text input fields • 113	
Checkbox • 114	
Switch • 115	
Slider • 115	
RadioButton • 116	

<i>Progress indicators</i> • 116	
<i>AlertDialog</i> • 117	
Exercise 3.03 – Creating a Settings screen • 118	
Jetpack Compose layout groups	128
Box • 128	
Surface • 131	
Card • 132	
Column • 133	
Row • 135	
Exercise 3.04 – Creating a Profile page • 138	
Activity 3.01 – Creating a business metrics dashboard • 144	
Summary	145
 Chapter 4: Building App Navigation	 147
Technical requirements	147
Creating a screen structure with a Scaffold composable and slots	148
Building a navigation graph	151
Exercise 4.1 – Building simple navigation • 154	
Implementing a navigation drawer	161
Exercise 4.2 – Creating an app with a navigation drawer • 164	
Adding bottom navigation	171
Exercise 4.3 – Adding bottom navigation to your app • 171	
Introducing tabbed navigation	177
Exercise 4.4 – Using tabs for app navigation • 177	
Activity 4.1 – Building primary and secondary app navigation • 184	
Summary	185

Part 2: App Components 187

Chapter 5: Essential Libraries – Ktor, Kotlin Serialization, and Coil 189

Technical requirements	189
Introducing REST, API, JSON, and XML	190
REST and APIs • 190	
JSON • 191	
XML • 191	
Processing JSON payloads • 192	
Fetching data from a network endpoint	192
Setting up Ktor and internet permissions • 192	
Making API requests with Ktor and displaying data • 193	
Exercise 5.1 – reading data from an API • 195	
Parsing a JSON response	199
Configuring Kotlin Serialization with Ktor • 199	
Defining data models for JSON mapping • 200	
Exercise 5.2 – extracting the image URL from the API response • 202	
Loading images from a remote URL	205
Exercise 5.3 – loading the image from the obtained URL • 207	
Activity 5.1 – displaying the current weather • 209	
Summary	211

Chapter 6: Building Lists with Jetpack Compose 213

Technical requirements	214
Adding a lazy list to our layout	214
Exercise 6.01 – Adding an empty LazyColumn to your main activity • 215	
Populating a LazyColumn composable	217
Exercise 6.02 – Populating your LazyColumn composable • 222	
Responding to clicks in LazyColumn composables	227
Exercise 6.03 – Responding to clicks • 228	

Supporting different item types	231
Exercise 6.04 – Adding titles to lazy lists • 233	
Swiping to remove items	235
Exercise 6.05 – Adding swipe-to-delete functionality • 240	
Adding items interactively	244
Exercise 6.06 – Implementing an Add Cat button • 246	
Activity 6.01 – Managing a list of Items • 247	
Summary	251
Further reading	251
 Chapter 7: Android Permissions and Google Maps	 253
Technical requirements	254
Requesting permission from the user	254
Exercise 7.01 – requesting the location permission • 260	
Showing a map of the user’s location	266
Exercise 7.02 – obtaining the user’s current location • 272	
Map clicks and custom markers	279
Exercise 7.03 – adding a custom marker where the map was clicked • 283	
Activity 7.01 – creating an app to find the location of a parked car • 287	
Summary	289
 Chapter 8: Services, WorkManager, and Notifications	 291
Technical requirements	292
Starting a background task using WorkManager	292
Exercise 8.01 – Executing background work with the WorkManager class • 297	
Background operations noticeable to the user – using foreground workers	303
Exercise 8.02 – Tracking your SCA with a foreground worker • 307	
Canceling or updating ongoing work	318
Exercise 8.03 – Aborting SCA deployment by canceling a worker • 318	
Activity 8.01 – A reminder to drink water • 320	
Summary	321

Part 3: Code Structure 323

Chapter 9: Testing with JUnit, Mockito, MockK, and Compose 325

Technical requirements 326

Understanding the types of testing 326

Writing a simple JUnit test 328

 Creating and running basic unit tests • 328

 Handling edge cases and improving test reliability • 331

Using Android Studio to run tests 336

Mocking objects 341

 Using MockK • 346

 Exercise 9.01 – Testing the sum of numbers • 348

Writing integration tests 352

 Robolectric • 353

 Jetpack ComposeTestRule library • 358

 Exercise 9.02 – Double integration • 363

Running UI tests 373

 Exercise 9.03 – Dealing with random events • 377

Applying TDD 384

 Exercise 9.04 – Using TDD to calculate the sum of numbers • 385

 Activity 9.01 – Developing with TDD • 388

Summary 389

Chapter 10: Coroutines and Flow 391

Technical requirements 392

Using Coroutines on Android 392

 Coroutine builders • 394

 Coroutine scope • 394

 Creating coroutines • 396

 Coroutine dispatchers • 396

Coroutine contexts • 397	
Coroutine Job elements • 398	
Exercise 10.01 – using Coroutines in an Android app • 398	
Using Flow on Android	403
Collecting flows on Android • 404	
Creating flows with flow builders • 406	
Using operators with flows • 407	
Exercise 10.02 – using Flow in an Android application • 408	
Activity 10.01 – creating a TV guide app • 411	
Summary	413
 Chapter 11: Android Architecture Components	 415
Technical requirements	416
Understanding Android components' background	416
Exploring ViewModel	417
Exercise 11.01 – Compose and ViewModel components • 420	
Combining ViewModel with data streams	424
LiveData • 424	
Coroutines and flows • 426	
Exercise 11.02 – Compose, ViewModel components, and flows • 427	
Saving instance states inside ViewModels	431
Persisting data with Room	432
Entities • 433	
DAO • 436	
Setting up the database • 438	
Third-party frameworks • 442	
Exercise 11.03 – making a little room • 443	
Activity 11.01 – shopping notes app • 449	
Summary	450

Chapter 12: Persisting Data 453

Technical requirements 453

Using SharedPreferences and DataStore 454

 SharedPreferences • 454

Exercise 12.01 – wrapping SharedPreferences • 455

 DataStore • 461

Exercise 12.02 – Preferences DataStore • 462

Saving data into files 466

 Internal storage • 468

 External storage • 470

 FileProvider • 470

 The Storage Access Framework • 471

 Asset files • 472

 Exercise 12.03 – copying files • 473

Understanding scoped storage 482

 Camera and media storage • 483

 Exercise 12.04 – taking photos • 485

 Activity 12.01 – managing multiple persistence options • 501

Summary 503

Chapter 13: Dependency Injection with Dagger, Hilt, and Koin 505

Technical requirements 506

Handling manual DI 506

 Exercise 13.01 – manual injection • 510

Using Dagger 2 516

 Consumers • 517

 Providers • 517

 Connectors • 518

 Qualifiers • 519

 Scopes • 520

Subcomponents • 521	
Exercise 13.02 – Dagger injection • 523	
Switching to Hilt	529
Exercise 13.03 – Hilt injection • 533	
Using Koin	537
Exercise 13.04 – Koin injection • 540	
Activity 13.01 – injected repositories • 544	
Summary	545

Part 4: Polishing and Publishing an App 547

Chapter 14: Architecture Patterns 549

Technical requirements	550
Getting started with MVVM	550
Implementing the Repository pattern	552
Exercise 14.1 – using Repository with Room in an Android project • 553	
Activity 14.1 – revisiting the TV Guide app • 556	
Summary	558

Chapter 15: Advanced Jetpack Compose 559

Technical requirements	559
Using CompositionLocal	560
Exercise 15.01 – adding CompositionLocal in an app • 561	
Using side-effects with Jetpack Compose	563
LaunchedEffect • 564	
DisposableEffect • 564	
SideEffect • 565	
Exercise 15.02 – adding side-effects in an app • 565	
Creating animations using Jetpack Compose	567
Animating a single value • 567	
Animating the appearance or disappearance of elements • 568	

Animating multiple values • 570	
Animating size changes of elements • 571	
Animating changes between composables • 572	
Animating values indefinitely • 572	
Other compose animations • 573	
Customizing animations • 574	
Debugging animations • 574	
Exercise 15.03 – adding animations with Jetpack Compose • 576	
Activity 15.01 – adding Animations to the TV Guide app • 580	
Summary	580
 Chapter 16: Launching Your App on Google Play	 583
Preparing your apps for release	584
Versioning apps • 584	
Creating a keystore • 586	
Exercise 16.01 – creating a keystore in Android Studio • 586	
Storing the keystore and passwords • 589	
Signing your apps for release • 591	
Android App Bundle • 591	
Exercise 16.02 – creating a signed app bundle • 592	
App signing by Google Play • 594	
Creating a developer account	595
Uploading an app to Google Play	596
Creating a store listing • 596	
<i>App details</i> • 597	
<i>Graphic assets</i> • 597	
Preparing the release • 597	
<i>App bundle</i> • 598	
Rolling out a release • 599	

Managing app releases	600
Release tracks • 600	
<i>The feedback channel and opt-in link • 601</i>	
<i>Internal testing • 602</i>	
<i>Closed testing • 602</i>	
<i>Open testing • 602</i>	
Staged rollouts • 602	
Managed publishing • 605	
Activity 16.01 – publishing an app • 606	
Summary	607
 Chapter 17: Unlock Your Book’s Exclusive Benefits	 609
How to unlock these benefits in three easy steps	609
Step 1 • 609	
Step 2 • 610	
Step 3 • 610	
Need help? • 611	
 Other Books You May Enjoy	 615
Index	619

Preface

Written by four veteran developers with 60+ years of collective experience, this updated third edition will jumpstart your Android development journey, focusing on Kotlin libraries and Jetpack Compose, Google's powerful declarative UI framework.

You'll learn the fundamentals of app development, enabling you to use Android Studio, as well as getting to grips with Jetpack Compose to create your first screens, build apps to run them on virtual devices through guided exercises, and implement Jetpack Compose's layout groups to make the most of lists, images, and maps.

The book has been updated with Kotlin's powerful networking and Coroutines libraries to help you fetch data in the background from a web service and manage the display of data using Kotlin flows. You'll learn about testing, creating clean architecture, persisting data, and exploring the dependency injection pattern, as well as how to publish your apps on the Google Play Store. You'll also work on realistic projects split up into bite-sized exercises and activities, along with building apps to create quizzes, read news articles, check weather reports, store recipes, retrieve movie information, as well as to remind you where you parked your car.

By the end of this book, you'll have gained the skills and confidence to build your own creative Android apps using Kotlin.

Who this book is for

If you want to build your own Android apps using Kotlin but are unsure of how to begin, then this book is for you. Basic knowledge of the Kotlin programming language or experience in a similar programming language, along with a willingness to brush up on Kotlin, is required.

What this book covers

Chapter 1, Creating Your First App, shows how to use Android Studio to build your first Android app. You will create an Android Studio project, understand what it's made up of, and explore the tools necessary for building and deploying an app on a virtual device. You will also learn about the structure of an Android app.

Chapter 2, Building User Screen Flows, shows how the Android system interacts with your app through the Android lifecycle, how you are notified of changes to your app's state, and how you can use the Android lifecycle to respond to these changes.

Chapter 3, Developing the UI with Jetpack Compose, provides an in-depth look at basic composable functions and layout groups in Jetpack Compose. It demonstrates how to use them to build the UI and respond to state changes.

Chapter 4, Building App Navigation, goes through how to build user-friendly app navigation through three primary patterns – the navigation drawer, bottom navigation, and tabbed navigation – so that users can easily access your app's content.

Chapter 5, Essential Libraries – Ktor, Kotlin Serialization, and Coil, introduces Ktor, a popular networking library, and shows how to make network requests and how to encode and decode JSON strings. Finally, it shows you how to load images from the web and present them in your app.

Chapter 6, Building Lists with Jetpack Compose, teaches you how to present items in a list using Compose's lazy lists, which work even if the list is incredibly long. It also demonstrates how to let your users interact with items in a list.

Chapter 7, Android Permissions and Google Maps, goes over the concept of permissions in Android and shows you how to request relevant permissions. We then go on to implement an interactive map using Google Maps and see how we can add markers to the map.

Chapter 8, Services, WorkManager, and Notifications, covers the different ways in which background work can be processed in the Android world. This will give you the knowledge needed to download large files in the background, play music, and monitor events even when your app is not in the foreground.

Chapter 9, Testing with JUnit, Mockito, MockK, and Compose, covers the libraries and frameworks available to test the code base of an Android application.

Chapter 10, Coroutines and Flow, introduces you to doing background operations and data manipulations with Coroutines and flows. You'll also learn about manipulating and displaying data using flow operators.

Chapter 11, Android Architecture Components, goes over some of the more commonly used architecture components, such as `ViewModel`, which can be used to separate business logic from the user interface, and `Room`, which shows how you can persist data in a structured way.

Chapter 12, Persisting Data, covers the available options for persisting data on an Android device, from key-value formats to files.

Chapter 13, Dependency Injection with Dagger, Hilt, and Koin, presents the concept of dependency injection and the available libraries that can be used to implement dependency injection in an Android app.

Chapter 14, Architecture Patterns, explains the architecture patterns you can use to structure your Android projects to separate them into different components with distinct functionality. These make it easier for you to develop, test, and maintain your code.

Chapter 15, Advanced Jetpack Compose, discusses how to enhance your apps with effects and animations with Jetpack Compose.

Chapter 16, Launching Your App on Google Play, concludes this book by showing you how to publish your apps on Google Play: from preparing a release to creating a Google Play Developer account, and finally, releasing your app.

To get the most out of this book

Software/hardware covered in the book	Operating system requirements
Android Studio Meerkat or later versions	Windows, macOS, or Linux

If you are using the digital version of this book, we advise you to type the code yourself or access the code from the book's GitHub repository (a link is available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

Download the example code files

The code bundle for the book is hosted on GitHub at <https://github.com/PacktPublishing/How-to-Build-Android-Applications-with-Kotlin-Third-Edition>. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing>. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: <https://packt.link/gbp/9781835882764>.

Conventions used

There are a number of text conventions used throughout this book.

CodeInText: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. For example: “Execute the terraform graph command:”

A block of code is set as follows:

```
var counter = 0
val mainView = findViewById<ConstraintLayout>(R.id.main)
val counterValue = mainView.findViewById<TextView>(
    R.id.counter_value
)
val plusButton = mainView.findViewById<Button>(
    R.id.plus
)
val minusButton = mainView.findViewById<Button>(
    R.id.minus
)
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
@Composable
fun Employee(employee: EmployeeUiModel) {
    Row {
        if (employee.imageUrl.isEmpty()) {
            Spacer(modifier = Modifier.size(64.dp))
        } else {
            LoadedImage(
                imageUrl = employee.imageUrl,
                modifier = Modifier.size(64.dp)
            )
        }
    }
}
```

```
Column {  
    Text(text = employee.name)  
    Text(text = employee.role.label)  
}  
}  
}
```

Note: Although dependencies in the respective `libs.versions.toml` files may appear in two lines in the chapters, they should be added as one lines in your code.

Any command-line input or output is written as follows:

```
set KEYSTORE_PASSWORD=securepassword
```

Bold: Indicates a new term, an important word, or words that you see on the screen. For instance, words in menus or dialog boxes appear in the text like this. For example: “You will see a window with side tabs and **Projects** highlighted. Select the **New Project** option on the right-hand side.”



Warnings or important notes appear like this.



Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book or have any general feedback, please email us at customer@packt.com and mention the book’s title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you reported this to us. Please visit <http://www.packt.com/submit-errata>, click **Submit Errata**, and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit <http://authors.packt.com/>.

Share Your Thoughts

Once you've read *How to Build Android Applications with Kotlin, Third Edition*, we'd love to hear your thoughts! Please [click here](#) to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Part 1

Android Foundation

This first part introduces you to Android Studio, the **integrated development environment (IDE)** used for Android development, and then guides you through the building blocks of Android development using the Jetpack Compose Android UI toolkit. It provides a comprehensive overview of the Android framework, working through guided exercises that reinforce the learning objectives so that this knowledge can be retained.

This part of the book includes the following chapters:

- *Chapter 1, Creating Your First App*
- *Chapter 2, Building User Screen Flows*
- *Chapter 3, Developing the UI with Jetpack Compose*
- *Chapter 4, Building App Navigation*

1

Creating Your First App

This chapter is an introduction to Android, where you will set up the Android Studio environment and focus on the fundamentals of Android development.

You will cover creating an Android project with Android Studio. Then, you will set up a virtual device and run your app on it. You will explore the Android manifest file, which details all the Android app components, features, and permissions model. You will learn how to use the Gradle build system and analyze the Android application structure. You will start developing **user interfaces (UIs)** with the Android UI toolkit, Jetpack Compose.

By the end of this chapter, you will have gained the knowledge required to create an Android app from scratch and install it on a virtual Android device. You will be able to analyze and understand the importance of the `AndroidManifest.xml` file. You will know how an app project is structured in Android Studio and use the Gradle build tool to configure and build your app and manage library dependencies. Finally, you will be able to start implementing UI elements using Jetpack Compose with Material Design.

We will cover the following topics in the chapter:

- Creating an Android project with Android Studio
- Setting up a virtual device and running your app
- The Android manifest file
- Using Gradle to build, configure, and manage app dependencies
- Android application structure

Getting the most out of this book – get to know your free benefits

Unlock exclusive **free** benefits that come with your purchase, thoughtfully crafted to supercharge your learning journey and help you learn without limits.

Here's a quick overview of what you get with this book:

Next-gen reader



Figure 1.1: Illustration of the next-gen Packt Reader's features

Our web-based reader, designed to help you learn effectively, comes with the following features:

- ☁ **Multi-device progress sync:** Learn from any device with seamless progress sync.
- 📖 **Highlighting and notetaking:** Turn your reading into lasting knowledge.
- 🔖 **Bookmarking:** Revisit your most important learnings anytime.
- 🌙 **Dark mode:** Focus with minimal eye strain by switching to dark or sepia mode.

Interactive AI assistant (beta)



Figure 1.2: Illustration of Packt's AI assistant

Our interactive AI assistant has been trained on the content of this book, to maximize your learning experience. It comes with the following features:

- ✦ Summarize it: Summarize key sections or an entire chapter.
- ✦ AI code explainers: In the next-gen Packt Reader, click the Explain button above each code block for AI-powered code explanations.

Note: The AI assistant is part of next-gen Packt Reader and is still in beta.

DRM-free PDF or ePub version

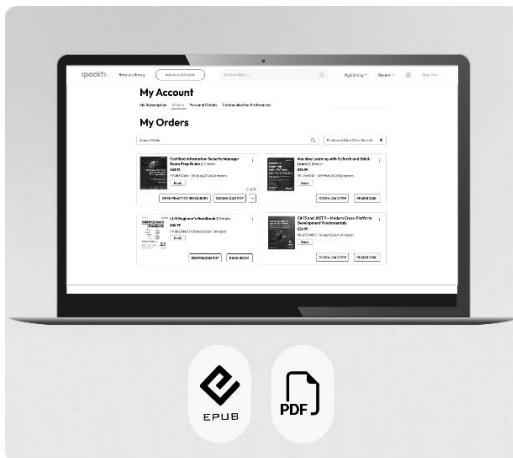


Figure 1.3: Free PDF and ePub

Learn without limits with the following perks included with your purchase:

- 📄 Learn from anywhere with a DRM-free PDF copy of this book.
- 📖 Use your favorite e-reader to learn using a DRM-free ePub version of this book.

Unlock this book's exclusive benefits now

Scan this QR code or go to packtpub.com/unlock, then search for this book by name. Ensure it's the correct edition.

Note: Keep your purchase invoice ready before you start.

UNLOCK NOW



Technical requirements

The complete code for all the exercises and the activity in this chapter is available on GitHub at <https://packt.link/EiB80>.

Creating an Android project with Android Studio

In order to be productive building Android apps, it is essential to become confident in using **Android Studio**. This is the official **integrated development environment (IDE)** for Android development, built on JetBrains' **IntelliJ IDEA** IDE and developed by the Android Studio team at Google. You will use it throughout this book to create apps and progressively add more advanced features.

Since Google I/O 2017 (the annual Google developer conference), **Kotlin** has been Google's preferred programming language for Android app development. You will be using Kotlin throughout this book to build Android apps.

Kotlin was created to address some of the shortcomings of Java in terms of verbosity, handling null types, and adding more functional programming techniques, among many other issues.

Getting to grips with and familiarizing yourself with Android Studio will enable you to feel confident building Android apps. So, let's get started with creating your first project.



The installation and setup of Android Studio are covered in the *Preface*. Please ensure you have completed those steps before you continue.

Exercise 1.01 – Creating an Android Studio project for your app

This is the starting point for creating a project structure that your app will be built upon. The template-driven approach will enable you to create a basic project in a short timeframe while setting up the building blocks you can use to develop your app.

To complete this exercise, perform the following steps:

1. Open Android Studio. You will see a window with side tabs and **Projects** highlighted. Select the **New Project** option on the right-hand side.
2. Now, you'll enter a simple wizard-driven flow, which greatly simplifies the creation of your first Android project. The next screen you will see has many options for the initial setup you'd like your app to have:

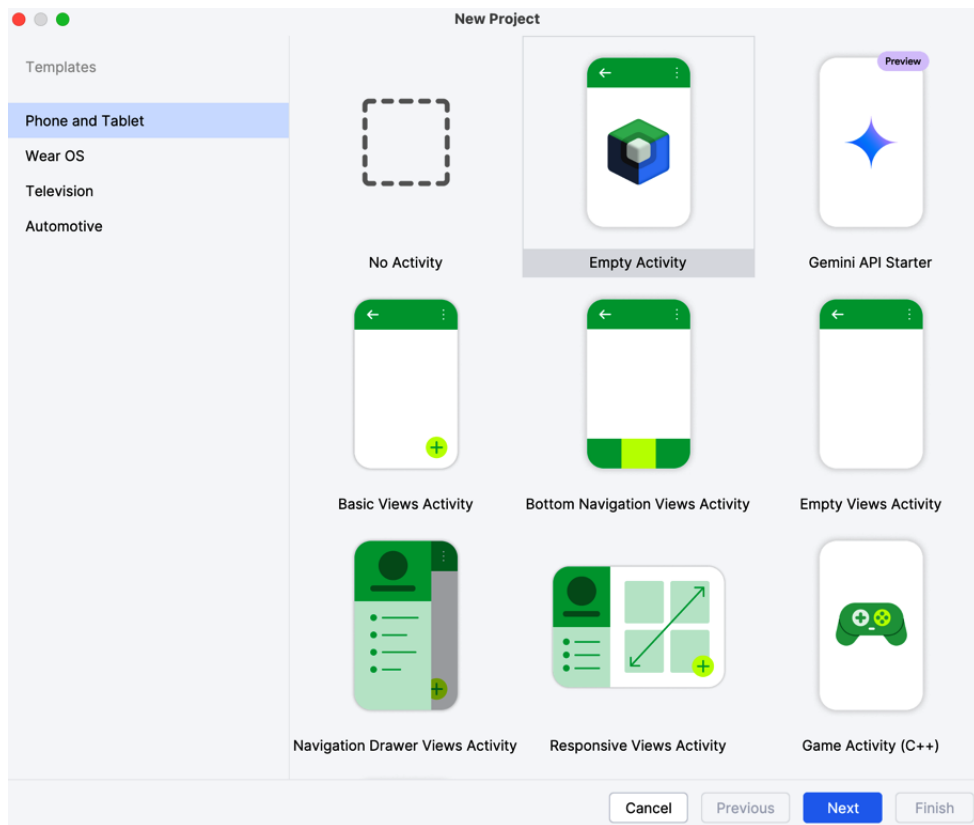
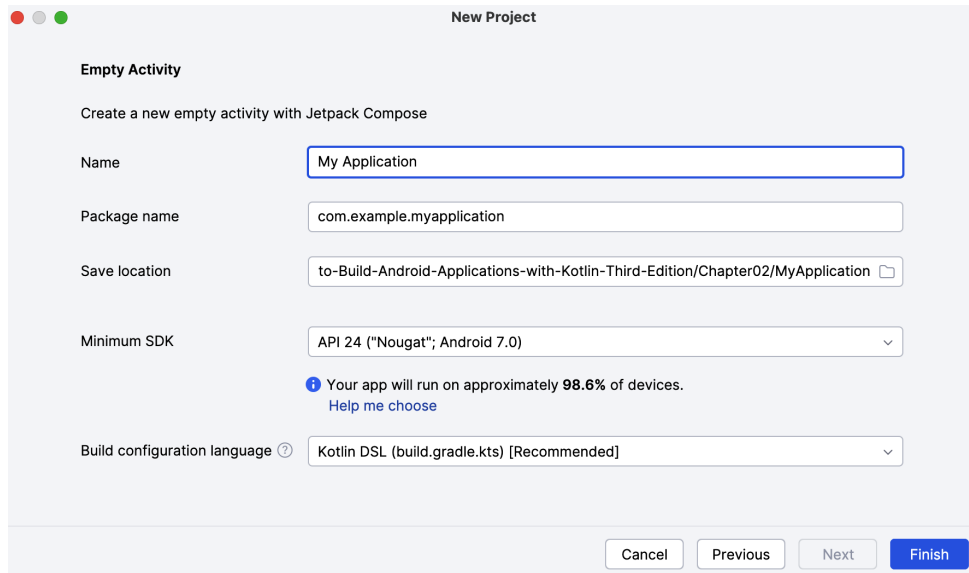


Figure 1.4 – Starting a project template for your app

Welcome to your first introduction to the Android development ecosystem. The word displayed in most of the project types is *Activity*. In Android, an **activity** represents a single screen in your app that a user can interact with. When creating a new project, you can choose from different templates, each of which sets up the initial screen in a specific way. These templates determine how the first screen will look and provide a starting point for building your app. Select **Empty Activity** from the template and click on **Next**. The project configuration screen is as follows:



The screenshot shows the 'New Project' dialog in Android Studio. The title bar says 'New Project'. The main heading is 'Empty Activity'. Below it, the text says 'Create a new empty activity with Jetpack Compose'. There are several input fields: 'Name' with the value 'My Application', 'Package name' with the value 'com.example.myapplication', 'Save location' with the value 'to-Build-Android-Applications-with-Kotlin-Third-Edition/Chapter02/MyApplication', 'Minimum SDK' with a dropdown menu showing 'API 24 ("Nougat"; Android 7.0)', and 'Build configuration language' with a dropdown menu showing 'Kotlin DSL (build.gradle.kts) [Recommended]'. There is also a blue information icon with the text 'Your app will run on approximately 98.6% of devices.' and a link 'Help me choose'. At the bottom right, there are four buttons: 'Cancel', 'Previous', 'Next', and 'Finish'.

Figure 1.5 – Project configuration

 **Quick tip:** Need to see a high-resolution version of this image? Open this book in the next-gen Packt Reader or view it in the PDF/ePub copy.

 **The next-gen Packt Reader** and a **free PDF/ePub copy** of this book are included with your purchase. Scan the QR code OR visit packtpub.com/unlock, then use the search bar to find this book by name. Double-check the edition shown to make sure you get the right one.



The preceding screen configures your app. Let's go through all the options:

- **Name:** Similar to the name of your Android project, this name will appear as the default name of your app when it's installed on a phone and visible on Google Play.
 - **Package name:** This uses the standard reverse domain name pattern to create a name. It will be used as an address identifier for source code and assets in your app. It's best to make this name as clear and descriptive and as closely aligned with the purpose of your app as possible. As shown in *Figure 1.5*, the **Name** value of the app, `My Application` (in lowercase with spaces removed), is appended to the domain (`com.example.myapplication`).
 - **Save location:** This is the local folder on your machine where the app will initially be stored. By default, the project will be saved into a new folder with the name of the application with spaces removed. This results in a `MyApplication` project folder being created.
 - **Minimum SDK:** Depending on which version of Android Studio you download, the default might be the same as shown in *Figure 1.5* or a different version. Keep this the same. Most of Android's new features are made backward compatible, so your app will run fine on most older devices. However, if you do want to target newer devices, you should consider raising the minimum API level. There is a **Help me choose** link to a dialog that explains the feature set in different versions of Android and the current percentage of devices worldwide running each Android version.
 - **Build configuration language:** The language used to build your app. Keep this as **Kotlin DSL** (DSL stands for **domain-specific language**). A DSL is a programming language used for a particular domain or a specific set of tasks. In this case, the domain is build scripts. Kotlin is the programming language you'll use to build apps, so you will have the familiarity of using it in build scripts.
3. Once you have filled in all these details, select **Finish**. Your project will be built, and you will then be presented with the following screen or similar:

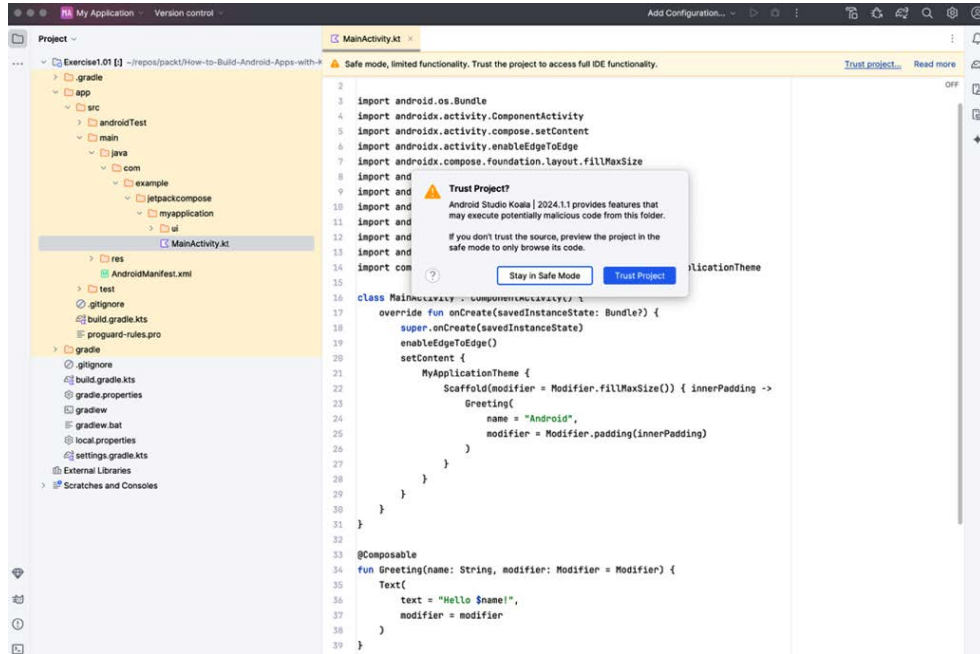


Figure 1.6 – Android Studio default project

You can immediately see the starting screen that has been created (MainActivity) and the application folder structure in the left panel. You may see a warning saying you are in safe mode. This will stay until you click **Trust Project**.

In this exercise, you have gone through the steps to create your first Android app using Android Studio. This template-driven approach has shown you the core options you need to configure for your app.

In the next section, you will set up a virtual device and see your app run for the first time.

Setting up a virtual device and running your app

As a part of installing Android Studio, you downloaded and installed the latest Android **software development kit (SDK)** components. This includes a base emulator, which you will configure to create an **Android Virtual Device (AVD)** to run Android apps on. An emulator mimics the hardware and software features and configuration of a real device. The benefit is that you can make changes and quickly see them on your desktop while developing your app. Although virtual devices do not have all the features of a real device, the feedback cycle is often quicker than going through the steps of connecting a real device.

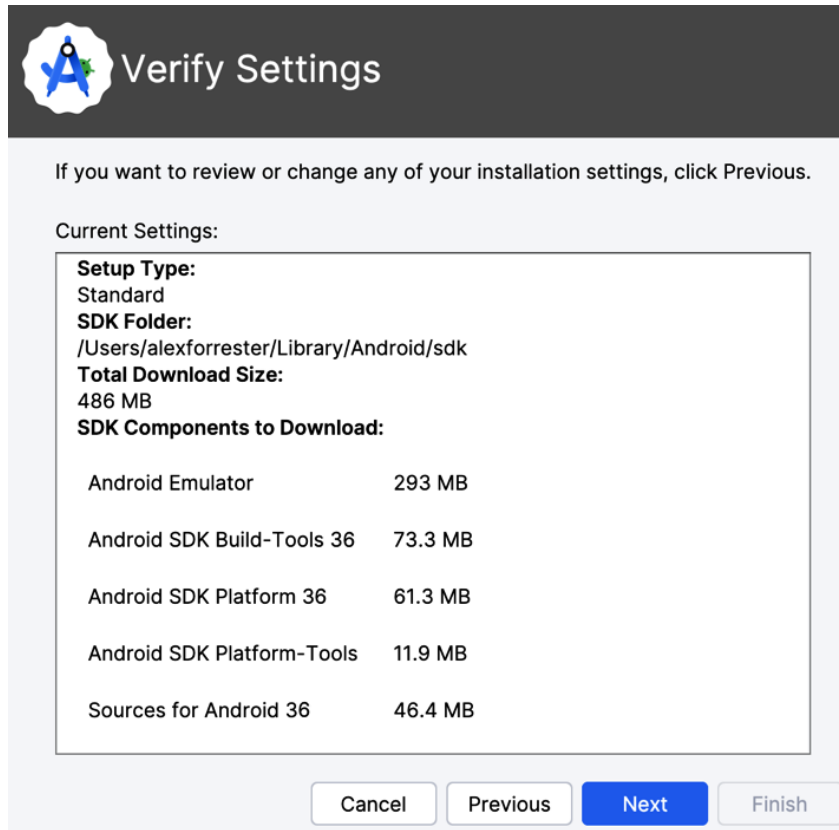


Figure 1.7 – SDK components

 **Quick tip:** Need to see a high-resolution version of this image? Open this book in the next-gen Packt Reader or view it in the PDF/ePub copy.

 **The next-gen Packt Reader** and a **free PDF/ePub copy** of this book are included with your purchase. Scan the QR code OR visit packtpub.com/unlock, then use the search bar to find this book by name. Double-check the edition shown to make sure you get the right one.



Let's take a look at the SDK components that are installed and how the virtual device fits in:

- **Android Emulator:** This is the base emulator, which we will configure to create virtual devices of different Android makes and models.
- **Android SDK Build-Tools 36:** Android Studio uses build tools to build your app. This process involves compiling, linking, and packaging your app to prepare it for installation on a device. The platform refers to the API level.
- **Android SDK Platform 36:** This is the version of the Android platform that you will use to develop your app.
- **Sources for Android 36:** When you are editing code, it is useful to see detailed information on the Android SDK within the source files (this is the code that will be compiled and run with your own code to create your app).

Exercise 1.02 – Setting up a virtual device and running your app on it

We set up an Android Studio project to create our app in *Exercise 1.01 – Creating an Android Studio project for your app*, and we are now going to run it on a virtual device. This process is a continuous cycle while working on your app. When implementing a feature, you can verify its look and behavior when you need to.

For this exercise, you will create a single virtual device, but you should ensure that you run your app on multiple devices to verify that its look and behavior are consistent. Perform the following steps:

1. On the right-hand side of Android Studio, there should be a **Device Manager** window open. If you can't see it, go to **View | Tool Windows | Device Manager** to display it.

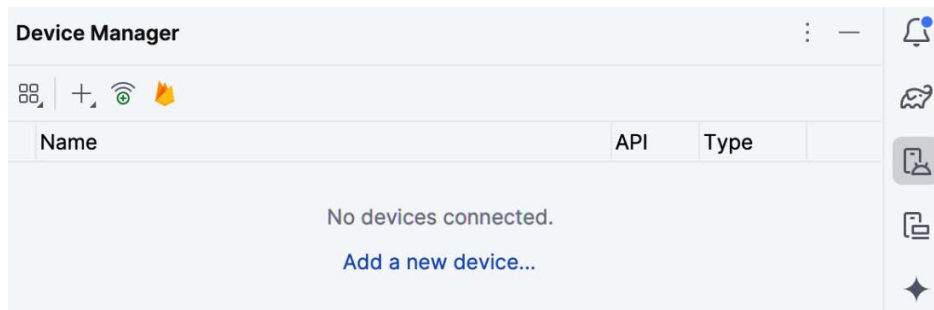


Figure 1.8 – The Device Manager window

2. On the right of the tool window, you will see an icon selected in a window that displays **No devices connected** and **Add a new device** below it.

3. Click **Add a new device**. There is also a + (plus) icon displayed at the top of the **Device Manager** window if you want to create another device. When selected, it displays a popup. Click on **Create Virtual Device**.

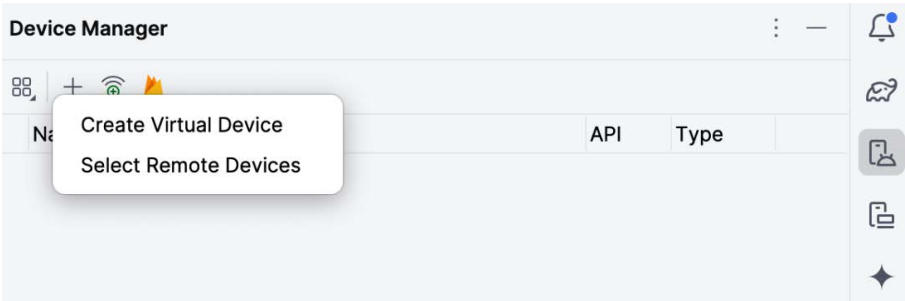


Figure 1.9 – Device Manager | Create Virtual Device

You will then be presented with this screen:

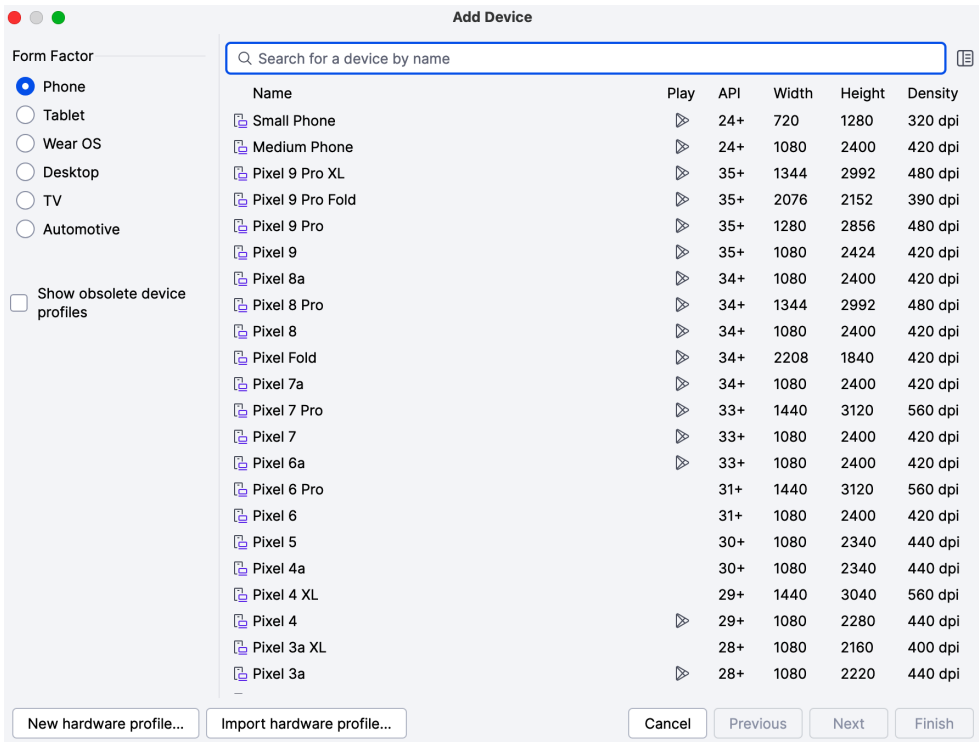


Figure 1.10 – Device definition

- 4. We are going to choose the **Pixel 9** device. The real (non-virtual device) Pixel range of devices is developed by Google and has access to the most up-to-date versions of the Android platform. Once selected, click the **Next** button:

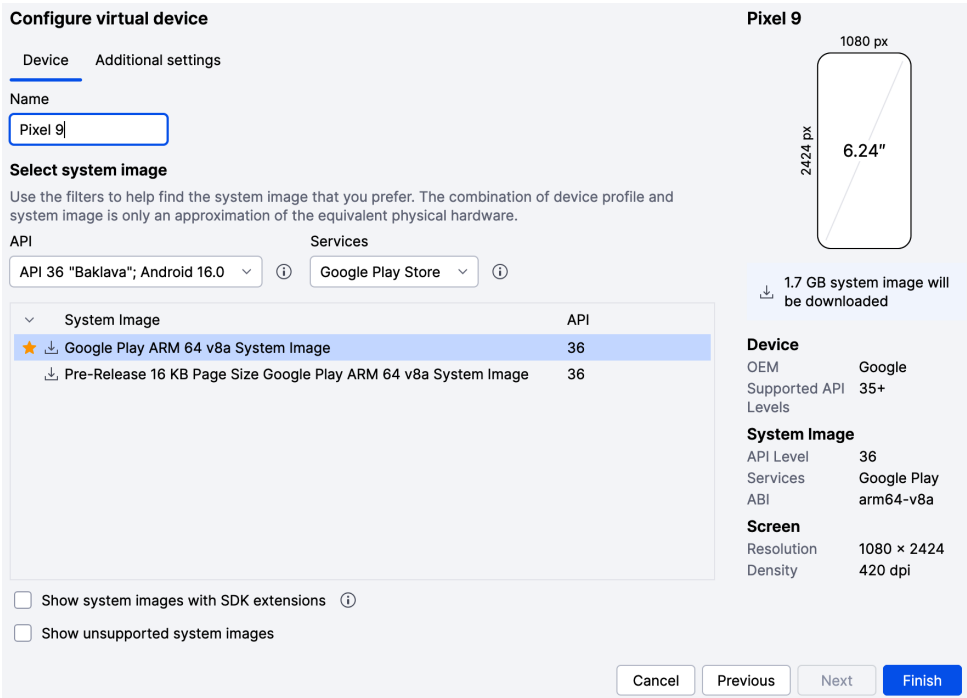


Figure 1.11 – System image

You can change the name of the system image in the top-left corner if you'd like to. Click the latest numbered API version. Here, it is **API 36**, which is the API level of Android 16. The **System Image** column might also show **Google Play** in the name, which means the system image comes pre-installed with Google Play services. You can also see a symbol in the **Play** column in *Figure 1.10*, which indicates this.

This is a rich feature set of Google APIs and Google apps that your app can use and interact with. On first running the app, you will see apps such as Maps, Chrome, and Play Store instead of a plain system image.

- 5. You should develop your app with the latest version of the Android platform to benefit from the latest features. On first creating a virtual device, you will have to download the system image.
- 6. Select **Yes** on the dialog to download the system image.

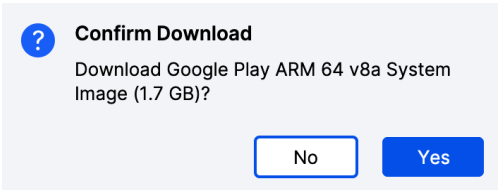


Figure 1.12 – System Image download

- 7. Click **Finish**, and your virtual device will be shown in the **Device Manager** window:

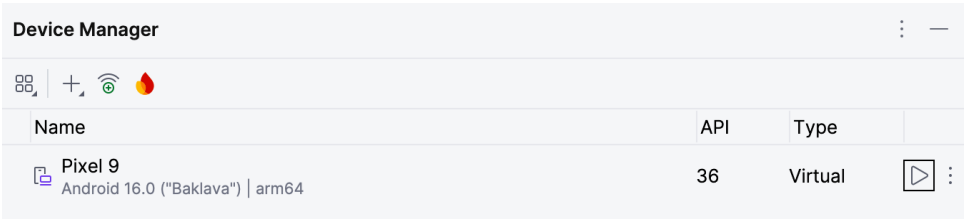


Figure 1.13 – The Device Manager window

- 8. Press the **Play** arrow button on the right-hand side to launch the **AVD**:



Figure 1.14 – Virtual device launched

9. The device name will also be visible in the top toolbar above the **Running Devices** window. If you can't see it, make sure the toolbar is visible by going to **View | Appearance | Toolbar**.



Figure 1.15 – Toolbar with launch options

10. Press the green triangle/**Play** button to launch your app.

This will load a basic app on the virtual device, as shown in *Figure 1.16*:



Figure 1.16 – The app running on a virtual device

In this exercise, you have gone through the steps to create a virtual device and run the app you created on it. The **Device Manager** window, which you have used to do this, enables you to create the device (or range of devices) you would like to target your app for. Running your app on the virtual device allows a quick feedback cycle to verify how a new feature behaves and displays the way you expect it to.

Next, you will explore the `AndroidManifest.xml` file of your project, which contains the information and configuration of your app.

The Android manifest file

The app you have just created, although simple, encompasses the core building blocks that you will use in all the projects you create. The app is driven from the `AndroidManifest.xml` file, a manifest file that details the contents of your app. To open it, locate the tool window by selecting **View | Tool Windows | Project**. Once displayed, the drop-down options at the top of the **Project** window allow you to change the way you view your project, with the most commonly used displays being **Android** and **Project**:

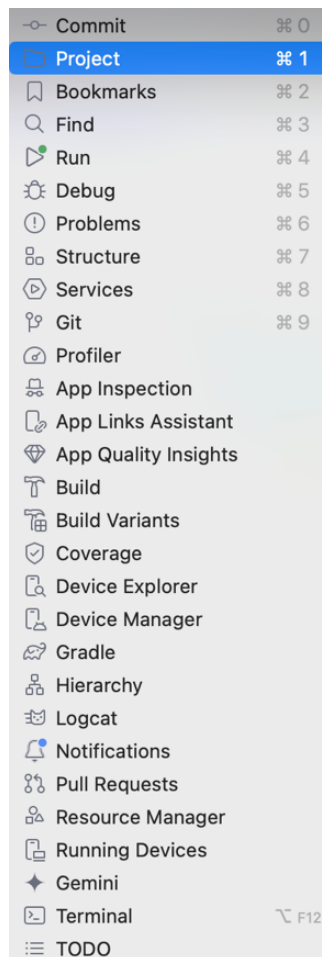


Figure 1.17 – The Tool Windows drop-down menu

The `AndroidManifest.xml` file is located at `app | manifests` in the **Android** display of the **Project** window:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">
    <application
        android:allowBackup="true"
        android:dataExtractionRules=
            "@xml/data_extraction_rules"
        android:fullBackupContent="@xml/backup_rules"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.MyApplication"
        tools:targetApi="31">
        <activity
            android:name=".MainActivity"
            android:exported="true"
            android:label="@string/app_name"
            android:theme="@style/Theme.MyApplication">
            <intent-filter>
                <action android:name=
                    "android.intent.action.MAIN" />
                <category android:name=
                    "android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

💡 **Quick tip:** Enhance your coding experience with the **AI Code Explainer** and **Quick Copy** features. Open this book in the next-gen Packt Reader. Click the **Copy** button (1) to quickly copy code into your coding environment, or click the **Explain** button (2) to get the AI assistant to explain a block of code to you.

```
function calculate(a, b) {  
  return {sum: a + b};  
};
```

Copy

1

Explain

2



🔒 **The next-gen Packt Reader** is included for free with the purchase of this book. Scan the QR code OR go to packtpub.com/unlock, then use the search bar to find this book by name. Double-check the edition shown to make sure you get the right one.



A typical manifest, in general terms, is a top-level file that describes the enclosed files or other data and associated metadata that form a group or unit. The Android manifest file applies this concept to your Android app as an XML file.

Every Android app has an application class that allows you to configure the app. After the `<application>` element opens, you define your app's components. As we have just created our app, it only contains the first screen, shown in the following code:

```
<activity android:name=".MainActivity">
```

The next child XML node specified is as follows:

```
<intent-filter>
```

Android uses intents as a mechanism for interacting with apps and system components. Intents get sent, and the intent filter registers your app's capability to react to these intents. `<android.intent.action.MAIN>` is the main entry point into your app, which, as it appears in the enclosing

XML of `.MainActivity`, specifies that this screen will be started when the app is launched. `android.intent.category.LAUNCHER` states that your app will appear in the list of installed apps on your user's device.

As you have created your app from a template, it has a basic manifest that will launch the app and display an initial screen at startup through an `Activity` component. Depending on which other features you want to add to your app, you may need to add permissions in the Android manifest file.

Permissions are grouped into three different categories – normal, signature, and dangerous:

- **Normal:** These permissions include accessing the network state, Wi-Fi, the internet, and Bluetooth. These are usually permitted without asking for the user's consent at runtime.
- **Signature:** These permissions are shared by the same group of apps that must be signed with the same certificate. This means these apps can share data freely, but other apps don't have access.
- **Dangerous:** These permissions are centered around the user and their privacy, such as sending **Short Message Service (SMS)** texts to access accounts and locations, and reading and writing to the filesystem and contacts.

These permissions must be listed in the manifest, and in the case of dangerous permissions, from Android Marshmallow API 23 (Android 6 Marshmallow) onward, you must also ask the user to grant the permissions at runtime.

In the next exercise, we will configure the Android manifest file. Detailed documentation on this file can be found at the link here: <https://packt.link/6LiNO>.

Exercise 1.03 – Configuring the Android manifest internet permission

The key permission that most apps require is access to the internet. This is not added by default. In this exercise, we will fix that and, in the process, load a `WebView` object, which is a view to show web pages. This use case is very common in Android app development as most commercial apps will display a privacy policy, terms and conditions, and so on. As these documents are likely common to all platforms, the usual way to display them is to load a web page. To do this, perform the following steps:

1. Create a new Android Studio project as you did in *Exercise 1.01 – Creating an Android Studio project for your app*.
2. Switch tabs to the `MainActivity` tab From the **Android** display of the **Project** window, it's located at `app | kotlin+java | com.example.myapplication`.

3. On opening the MainActivity tab you'll see that it has the MainActivity class. The following code is just a snippet of this file:

```
class MainActivity : ComponentActivity() {  
    override fun onCreate(savedInstanceState: Bundle?)  
    {  
        super.onCreate(savedInstanceState)  
        enableEdgeToEdge()  
        setContent {  
            MyApplicationTheme {  
                Scaffold(  
                    modifier = Modifier.fillMaxSize()  
                ) { innerPadding ->  
                    Greeting(  
                        name = "Android",  
                        modifier = Modifier  
                            .padding(innerPadding)  
                    )  
                }  
            }  
        }  
    }  
}
```

4. You'll examine the contents of this file in more detail in the next section of this chapter, but for now, you just need to be aware that the setContent function sets the layout of the UI you saw when you first ran the app in the virtual device. Update the preceding code to the following:

```
import android.webkit.WebView  
import androidx.compose.ui.viewinterop.AndroidView  
  
class MainActivity : ComponentActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContent {  
            val webView = WebView(this)  
            webView.settings.javaScriptEnabled = true  
            webView.loadUrl("https://www.google.com")  
        }  
    }  
}
```

```
        AndroidView(  
            modifier = Modifier.fillMaxSize(),  
            factory = { context ->  
                webView  
            }  
        )  
    }  
}
```

This creates a `WebView` object (which allows your app to show a web page) and adds it to an `AndroidView` composable.

The `val` keyword is a read-only property reference, which can't be changed once it has been set. JavaScript needs to be enabled in the `WebView` object to execute JavaScript. Then, we load the URL into the `WebView` object.

Android UIs are built with Jetpack Compose, which you will be using in this book. It's a declarative approach, which means that you describe the end state of how you want your app to display, and the framework draws/composes it. From here on, I'll refer to Jetpack Compose as simply *Compose*. The legacy method of building Android UIs used XML files, where you constructed the end display step by step by adding your views in an XML file before manually setting the views when loading, and when data changed. The way to use XML views in Compose is to wrap them in an `AndroidView` composable. As Compose doesn't have a built-in `WebView` composable, you need to use an `AndroidView` composable to embed the `WebView` object. The preceding code adds a `WebView` object with the `factory` argument, which takes a block of code of a legacy `View` and makes it available to use in Compose. The `modifier` argument enables you to add specific styling and layout behavior to the composable. `context` is an application-level abstract class that enables you to interact with the Android system.



We are not setting the type, but Kotlin has type inference, so it will infer the type if possible. So, specifying the type explicitly with `val webView: WebView = WebView(this)` is not necessary. Depending on which programming languages you have used in the past, the order of defining the parameter name and type may or may not be familiar. Kotlin uses the name followed by the type.

5. Now, run the app, and the text will appear as shown in the screenshot:

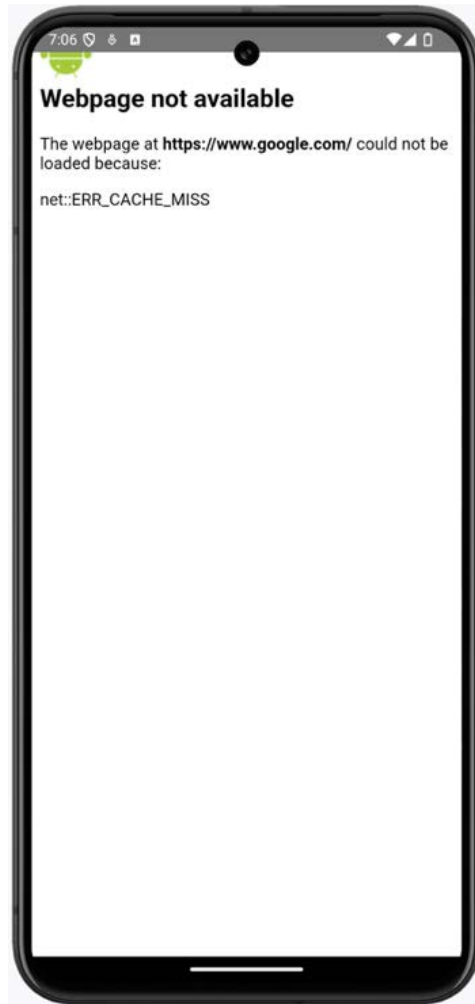


Figure 1.18 – No internet permission error message

6. This error occurs because there is no INTERNET permission added to your `AndroidManifest.xml` file. (If you get the `net::ERR_CLEARTEXT_NOT_PERMITTED` error, this is because the URL you are loading into `WebView` is not HTTPS, and non-HTTPS traffic is disabled from API level 28, Android 9.0 Pie, and above.)
7. Let's fix that by adding the INTERNET permission to the manifest. Open up the Android manifest file and add the following above the `<application>` tag:

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

You can find the full Android manifest file with the permission added here: <https://packt.link/ws71v>.

Uninstall the app from the virtual device before running the app again. You need to do this, as app permissions can sometimes get cached.

Do this by long-pressing the app icon, selecting the **App Info** option that appears, and then pressing the bin icon with the **Uninstall** text below it. Alternatively, long-press the app icon and then drag it to the bin icon with the **Uninstall** text beside it in the top-right corner of the screen.

8. Install the app again and see the web page appear in the WebView object.

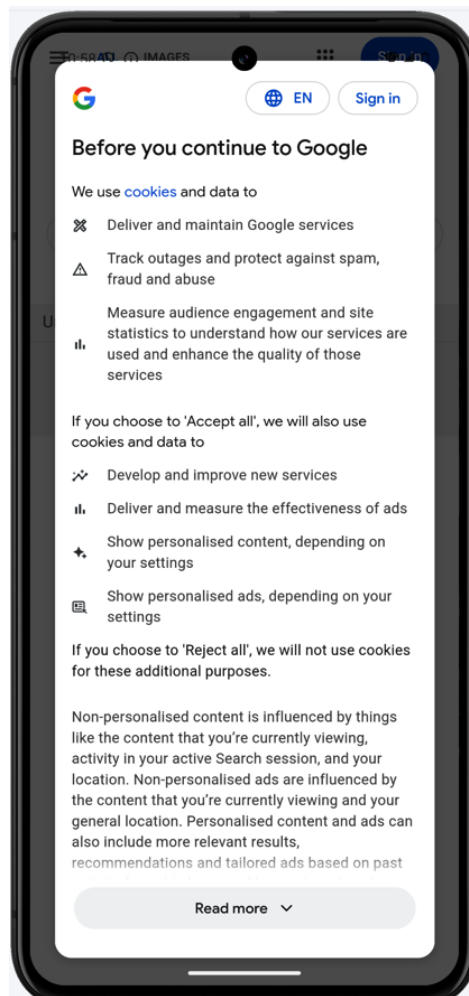


Figure 1.19 – App displaying WebView

In this example, you learned how to add a permission to the manifest. The Android manifest file can be thought of as a table of contents of your app. It lists all the components and permissions your app uses. As you have seen from starting the app from the launcher, it also provides the entry points into your app.

In the next section, you will explore the Android build system, which uses the Gradle build tool to get your app up and running.

Using Gradle to build, configure, and manage app dependencies

In the course of creating this project, you have principally used the Android platform SDK. The necessary Android libraries were downloaded when you installed Android Studio. However, these are not the only libraries that are used to create your app. To configure and build your Android project or app, a build tool called Gradle is used.



Gradle is a multi-purpose build tool that Android Studio uses to build your app. You can either use Kotlin or Groovy, a dynamically typed **Java virtual machine (JVM)** language, to configure the build process and allow easy dependency management so you can add libraries to your project and specify the versions.

As the default language when creating a new project in Android Studio is Kotlin, you will be using this. The files that this build and configuration information is stored in are named `build.gradle.kts`, which are Kotlin script files. You benefit from the same autocompletion and compile-time safety as your main app source files written in Kotlin.

When you first create your app, there are two `build.gradle.kts` files, one at the root/top level of the project and one specific to your app in the app module folder.

The project-level `build.gradle.kts` file

Let's now have a look at the project-level `build.gradle.kts` file. This is where you set up all the root project settings, which can be applied to sub-modules/projects:

```
plugins {  
    alias(libs.plugins.android.application) apply false  
    alias(libs.plugins.kotlin.android) apply false  
    alias(libs.plugins.kotlin.compose) apply false  
}
```

Gradle works on a plugin system, so you can write your own plugin that does a task or series of tasks and plug it into your build pipeline. The three plugins listed in the preceding snippet do the following:

- `libs.plugins.android.application`: This adds support to create an Android application
- `libs.plugins.jetbrains.kotlin.android`: This provides integration and language support for Kotlin in the project
- `libs.plugins.kotlin.compose`: This enables Jetpack Compose support for Kotlin

The `apply false` statement enables these plugins only to sub-projects/modules, and not the project's root level. The alias syntax refers to the version catalog we will discuss in the following subsection.

The app-level `build.gradle.kts` file

The `build.gradle.kts` app-level file is specific to your project configuration:

```
plugins {
    alias(libs.plugins.android.application)
    alias(libs.plugins.kotlin.android)
    alias(libs.plugins.kotlin.compose)
}
android {
    namespace = "com.example.myapplication"
    compileSdk = 35
    defaultConfig {
        applicationId = "com.example.myapplication"
        minSdk = 24
        targetSdk = 35
        versionCode = 1
        versionName = "1.0"
        testInstrumentationRunner = "androidx.test.runner.AndroidJUnitRunner"
    }
    buildTypes {
        release {
            isMinifyEnabled = false
            proguardFiles(
                getDefaultProguardFile("proguard-android-optimize.txt"),
                "proguard-rules.pro"
            )
        }
    }
}
```

```
        )
    }
}
compileOptions {
    sourceCompatibility = JavaVersion.VERSION_11
    targetCompatibility = JavaVersion.VERSION_11
}
kotlinOptions {
    jvmTarget = "11"
}
buildFeatures {
    compose = true
}
}
dependencies {...}
```

The plugins for Android, Kotlin, and Jetpack Compose, which are detailed in the `build.gradle.kts` root file, are applied to your project here in the plugins lines.

The `android` block provided by the `libs.plugins.android.application` plugin is where you configure your Android-specific configuration settings:

- **namespace:** This is set from the package name you specified when creating the project. It will be used for generating build and resource identifiers.
- **compileSdk:** This is used to define the API level the app has been compiled with, and the app can use the features of this API and lower.
- **defaultConfig:** This is the base configuration of your app.
- **applicationId:** This is set to your app's package and is the app identifier that is used on Google Play to uniquely identify your app. It can be changed to be different from the package name if required, but once you upload your app to the Google Play it is used to uniquely identify your app and can't be changed.
- **minSdk:** This is the minimum API level your app supports. This will filter out your app from being displayed in Google Play for devices that are lower than this.

- **targetSdk:** This is the API level you are targeting. This is the API level your built app is intended to work on and has been tested with.
- **versionCode:** This specifies the version code of your app. Every time an update needs to be made to the app, the version code needs to be increased by one or more.
- **versionName:** A user-friendly version name that usually follows semantic versioning of X.Y.Z, where X is the major version, Y is the minor version, and Z is the patch version, for example, 1.0.3.
- **testInstrumentationRunner:** This is the test runner to use for your UI tests.
- **buildTypes:** Under buildTypes, a release is added that configures your app to create a release build. The isMinifyEnabled value, if set to true, will shrink your app size by removing any unused code, as well as obfuscating your app. This obfuscation step changes the name of the source code references to values such as a.b.c(). This makes your code less prone to reverse engineering and further reduces the size of the built app.
- **compileOptions:** This is the language level of the Java source code (sourceCompatibility) and byte code (targetCompatibility).
- **kotlinOptions:** This is the jvm library that the kotlin gradle plugin should use.
- **buildFeatures:** This is where you configure specific parts of your build. In this case, we are specifying that we will use compose to create the UI rather than the legacy view system.
- The dependencies block specifies the libraries your app uses on top of the Android platform SDK, as shown here (with added comments):

```
dependencies {  
    // Kotlin extensions denoted by .ktx  
    // Android Kotlin Language features  
    implementation(libs.androidx.core.ktx)  
    implementation(libs.androidx.lifecycle.runtime.ktx)  
  
    // Jetpack Compose UI  
    implementation(libs.androidx.activity.compose)  
  
    // Jetpack Compose Versioning Library  
    implementation(platform(libs.androidx.compose.bom))  
  
    // ALL Android UI SDK and tooling  
    implementation(libs.androidx.ui)  
    implementation(libs.androidx.ui.graphics)  
    implementation(libs.androidx.ui.tooling.preview)
```

```

implementation(libs.androidx.material3)

// Standard Test Libraries for unit tests
testImplementation(libs.junit)

// UI Test runner
androidTestImplementation(libs.androidx.junit)

// Libraries for creating Android UI tests
androidTestImplementation(libs.androidx.espresso.core)
androidTestImplementation(libs.androidx.ui.test.junit4)

// AndroidX Versioning Library
androidTestImplementation(platform(
    libs.androidx.compose.bom))

// Debugging Tooling
debugImplementation(libs.androidx.ui.tooling)
debugImplementation(libs.androidx.ui.test.manifest)
}

```



The dependencies listed here are a simplified version of the full details of the dependencies. This is configured in a file called the `libs.versions.toml` configuration file. **Tom's Obvious, Minimal Language (TOML)** files are standard industry files designed to make configuring complex dependencies as simple as possible.

Go to the top-level Gradle folder and open the `libs.versions.toml` file. You can use the shortcut *double Shift* in Android Studio to open the file dialog, start typing the file, and be given possible suggestions of what you are looking for. The following or similar will be shown (it is an abridged version and doesn't show all the dependencies):

```

[versions]
agp = "8.9.2"
kotlin = "2.0.21"
coreKtx = "1.16.0"
junit = "4.13.2"

```

```
junitVersion = "1.2.1"
espressoCore = "3.6.1"
lifecycleRuntimeKtx = "2.8.7"
activityCompose = "1.10.1"
composeBom = "2024.09.00"

[libraries]
androidx-core-ktx = { group = "androidx.core", name = "core-ktx", version.
ref = "coreKtx" }
junit = { group = "junit", name = "junit", version.ref = "junit" }
androidx-junit = { group = "androidx.test.ext", name = "junit", version.
ref = "junitVersion" }
androidx-espresso-core = { group = "androidx.test.espresso", name =
"espresso-core", version.ref = "espressoCore" }
androidx-lifecycle-runtime-ktx = { group = "androidx.lifecycle", name =
"lifecycle-runtime-ktx", version.ref = "lifecycleRuntimeKtx" }
androidx-activity-compose = { group = "androidx.activity", name =
"activity-compose", version.ref = "activityCompose" }
androidx-compose-bom = { group = "androidx.compose", name = "compose-bom",
version.ref = "composeBom" }
androidx-ui = { group = "androidx.compose.ui", name = "ui" }
androidx-ui-graphics = { group = "androidx.compose.ui", name = "ui-
graphics" }
androidx-ui-tooling = { group = "androidx.compose.ui", name = "ui-tooling"
}
androidx-ui-tooling-preview = { group = "androidx.compose.ui", name = "ui-
tooling-preview" }
androidx-ui-test-manifest = { group = "androidx.compose.ui", name = "ui-
test-manifest" }
androidx-ui-test-junit4 = { group = "androidx.compose.ui", name = "ui-
test-junit4" }
androidx-material3 = { group = "androidx.compose.material3", name =
"material3" }

[plugins]
android-application = { id = "com.android.application", version.ref =
"agp" }
kotlin-android = { id = "org.jetbrains.kotlin.android", version.ref =
"kotlin" }
kotlin-compose = { id = "org.jetbrains.kotlin.plugin.compose", version.ref
= "kotlin" }
```



The dependency versions specified in the previous code section and the following sections of this and other chapters are subject to change and are updated over time, so they are likely to be higher when you create these projects.

The configuration groups of [versions], [libraries], and [plugins] listed here contain the details of the dependencies we saw in the `build.gradle.kts` app file.

The dependencies are identified in the [libraries] section by a group and a name, followed by a version. Although the names are simplified in the version catalog, this follows the Maven **Project Object Model (POM)** convention of using `groupId`, `artifactId`, and `versionId`. The `groupId` value is the group of dependencies the dependency comes from, `artifactId` is the unique name of the dependency, and `versionId` is the version number of the dependency. The versions are grouped together separately for ease of updating.

The build system locates and downloads these dependencies to build the app from the repositories block detailed in the `settings.gradle` file, as explained in the following section.

The implementation notation for adding these libraries in `build.gradle.kts` means that their internal dependencies will not be exposed to your app, making compilation faster.

The next Gradle file to examine is `settings.gradle`, which initially looks like this:

```
pluginManagement {
    repositories {
        google {
            content {
                includeGroupByRegex("com\\.android\\..*")
                includeGroupByRegex("com\\.google\\..*")
                includeGroupByRegex("androidx\\..*")
            }
        }
        mavenCentral()
        gradlePluginPortal()
    }
}

dependencyResolutionManagement {
    repositoriesMode.set(RepositoriesMode.FAIL_ON_PROJECT_REPOS)
    repositories {
        google()
```

```
mavenCentral()
}
}
rootProject.name = "My Application"
include(":app")
```

When you first create a project with Android Studio, there will only be one app module, `include(:app,)`, but when you add more features, you can add new modules that are dedicated to containing the source of a feature rather than packaging it in the main app module.

These are called **feature modules**, and you can supplement them with other types of modules, such as shared modules, which are used by all other modules, such as a networking module. This file also contains the repositories of the plugins and dependencies to download from, in separate blocks for plugins and dependencies.

Setting the value of `RepositoriesMode.FAIL_ON_PROJECT_REPOS` ensures that all dependency repositories are defined here; otherwise, a build error will be triggered.

Exercise 1.04 – Exploring how Material Design in Jetpack Compose is used to theme an app

In this exercise, you will learn about the design language **Material Design**, created by Google in 2014 (now in its third version, Material 3), and use it to load a Material-Design-themed app. Material Design adds enriched UI elements based on real-world effects, such as lighting, depth, shadows, and animations. Perform the following steps to complete the exercise:

1. Create a new Android Studio project as you did in *Exercise 1.01 – Creating an Android Studio project for your app*.
2. The dependency that loads Material Design in `build.gradle.kts` is as follows:

```
implementation(libs.androidx.material3)
```

3. Next, open the `Color.kt` file located in the `app | src | main | java | com.example.myapplication | ui.theme` package (the import statements have been omitted):

```
val Purple80 = Color(0xFFD0BCFF)
val PurpleGrey80 = Color(0xFFCCC2DC)
val Pink80 = Color(0xFFFB8C8)

val Purple40 = Color(0xFF6650a4)
```

```
val PurpleGrey40 = Color(0xFF625b71)
val Pink40 = Color(0xFF7D5260)
```

The format is based on the **alpha, red, green, blue** (ARGB) color space, so the first four characters are for *alpha* (transparency), the next two for *red*, the next two for *green*, and the last two for *blue*. For alpha, 0x00 is completely transparent through to 0xFF, which is completely opaque. For the colors, 00 means none of the color is added to make up the composite color, and FF means all of the color is added.

4. Open the Theme.kt file in the same ui.theme package:

```
private val DarkColorScheme = darkColorScheme(
    primary = Purple80,
    secondary = PurpleGrey80,
    tertiary = Pink80
)

private val LightColorScheme = lightColorScheme(
    primary = Purple40,
    secondary = PurpleGrey40,
    tertiary = Pink40

    /* Other default colors to override
    background = Color(0xFFFFFBFE),
    surface = Color(0xFFFFFBFE),
    onPrimary = Color.White,
    onSecondary = Color.White,
    onTertiary = Color.White,
    onBackground = Color(0xFF1C1B1F),
    onSurface = Color(0xFF1C1B1F),
    */
)

@Composable
fun MyApplicationTheme(
    darkTheme: Boolean = isSystemInDarkTheme(),
    // Dynamic color is available on Android 12+
    dynamicColor: Boolean = true,
    content: @Composable () -> Unit
) {
    val colorScheme = when {
```

```

        dynamicColor &&
        Build.VERSION.SDK_INT >=
        Build.VERSION_CODES.S -> {
            val context = LocalContext.current
            if (darkTheme)
                dynamicDarkColorScheme(context)
            else
                dynamicLightColorScheme(context)
        }
        darkTheme -> DarkColorScheme
        else -> LightColorScheme
    }
    MaterialTheme(
        colorScheme = colorScheme,
        typography = Typography,
        content = content
    )
}

```

The main color schemes for light and dark modes are defined as `DarkColorScheme` and `LightColorScheme` and use three colors from `Color.kt` to override some of the principal colors set in the app. This is only a fraction of the colors that are available to override and customize the color palette used in your app.

`MyApplicationTheme` is what `MainActivity` uses within the `setContent` block to theme your app. Let's examine the function. `darkTheme` is set to `true` depending on the system dark mode setting that is retrieved with the `isSystemInDarkTheme()` function. This is the `darkTheme` display toggle in the **Settings** app on the device. `colorScheme` is then set to either `dynamicColor` if it is set to `true`, or either `DarkColorScheme` or `LightColorScheme`, depending on whether `darkTheme` has been applied. `dynamicColor` is constructed from colors that are extracted from the device's home screen and lock screen. Most apps will require setting their own theming rather than a system-derived one, so this is usually not the styling you require and can be removed to leave it as follows:

```

val colorScheme = when {
    darkTheme -> DarkColorScheme
    else -> LightColorScheme
}

```

5. Next, open the `Type.kt` file:

```
val Typography = Typography(  
    bodyLarge = TextStyle(  
        fontFamily = FontFamily.Default,  
        fontWeight = FontWeight.Normal,  
        fontSize = 16.sp,  
        lineHeight = 24.sp,  
        letterSpacing = 0.5.sp  
    ),  
    /* Other default text styles to override  
    titleLarge = TextStyle(  
        fontFamily = FontFamily.Default,  
        fontWeight = FontWeight.Normal,  
        fontSize = 22.sp,  
        lineHeight = 28.sp,  
        letterSpacing = 0.sp  
    ),  
    labelSmall = TextStyle(  
        fontFamily = FontFamily.Default,  
        fontWeight = FontWeight.Medium,  
        fontSize = 11.sp,  
        lineHeight = 16.sp,  
        letterSpacing = 0.5.sp  
    )  
    */  
)
```

The Material Design specification consists of 13 Text styles. Each one has a default, but you can override them. Here, `bodyLarge` `TextStyle` is being overridden with some custom settings – `bodyLarge` is the default `TextStyle` value used for all Text composables in your app.

6. Now, go back to the `Theme.kt` file, and you will see that the `Typography` property defined in `Type.kt` is used to set the typography used on the `MaterialTheme` theme that is created:

```
MaterialTheme(  
    colorScheme = colorScheme,  
    typography = Typography,  
    content = content  
)
```

7. The final argument in `MaterialTheme` is `content`, which applies the theme in `MainActivity`:

The great thing about using a theme is that you can style all your composables in one place instead of individually setting them on each composable. There is always the option to set a style on a composable to override the global setting if you wish.

8. To see dynamic theming in action, open the `Type.kt` file again, and change it to add the color property with a value of `Color.Red` to the `bodyLarge` style. Replace the existing code with the following:

```
import androidx.compose.ui.graphics.Color  
  
val Typography = Typography(  
    bodyLarge = TextStyle(  
        fontFamily = FontFamily.Default,  
        fontWeight = FontWeight.Normal,  
        fontSize = 16.sp,  
        lineHeight = 24.sp,  
        letterSpacing = 0.5.sp,  
        color = Color.Red  
    )  
)
```

9. The use of `sp` in the preceding code block stands for *scale-independent pixels*. This unit type represents the same values as density-independent pixels, which define the size measurement according to the density of the device that your app is being run on, and also changes the text size according to the user's preference, defined in **Settings | Display | Font style** (this might be **Font size and style** or something similar, depending on the exact device you are using).

10. Now, run the app. The Greeting composable text will be shown in red.
11. Go back to the MainActivity Greeting composable, add the line `import androidx.compose.material3.MaterialTheme` to the list of imports, and set a style directly on the Text Composable:

```
@Composable
fun Greeting(
    name: String, modifier: Modifier = Modifier
) {
    Text(
        style = MaterialTheme.typography.displayLarge,
        text = "Hello $name!",
        modifier = modifier
    )
}
```

The text color now goes back to black and displays the `displayLarge` style from `MaterialTheme`.

In this exercise, you’ve learned how Material Design in Jetpack Compose can be used to theme an app. Now that you’ve learned how the project is built and configured, in the next section, you’ll explore the project structure in detail, learn how it has been created, and gain familiarity with the core areas of the development environment.

Android application structure

Now that we have covered how the Gradle build tool works, we’ll explore the rest of the project. The simplest way to do this is to examine the folder structure of the app. Open up the **Project** tool window.

When you select it, you will see a view like the screenshot in *Figure 1.20*. If you can’t see any window bars on the left-hand side of the screen, then go to the top toolbar and select **View | Appearance | Tool Window Bars** and make sure it is ticked.

Open the corresponding folders in the **Android** display. This view neatly groups the app folder structure, so let’s look at it:

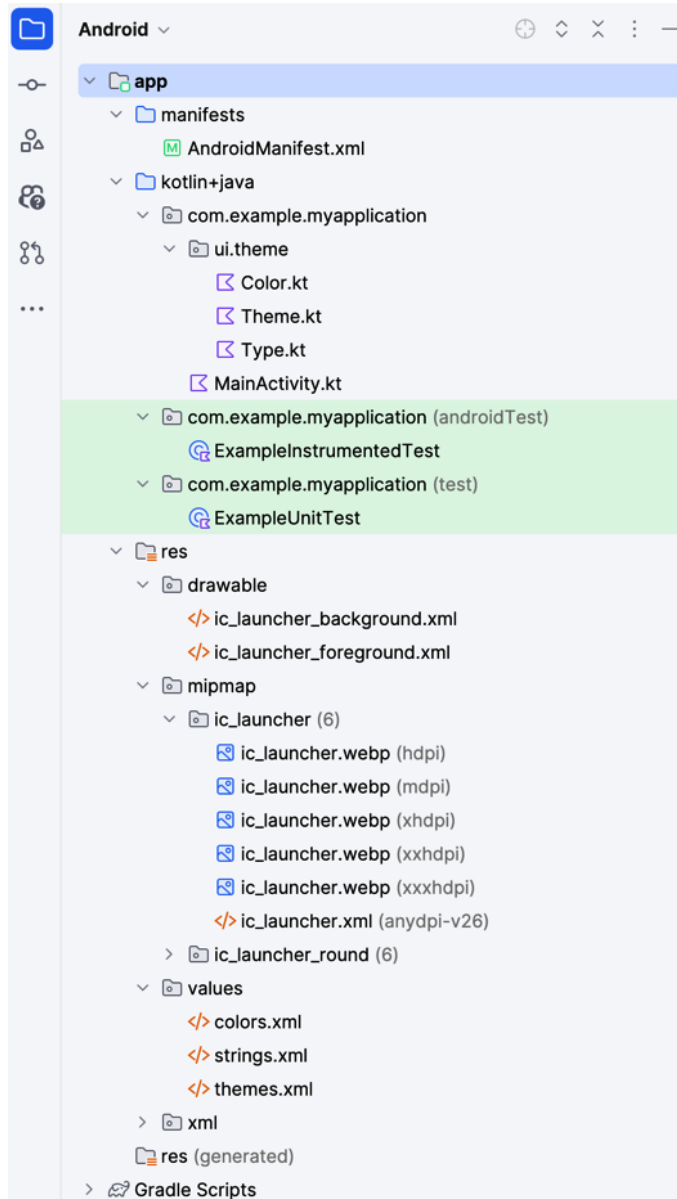


Figure 1.20 – Overview of the files and folder structure in the app

The Kotlin file (MainActivity), which you’ve specified as running when the app starts, is as follows:

```
import...

class MainActivity : ComponentActivity() {
```

```

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {
            MyApplicationTheme {
                Scaffold(
                    modifier = Modifier.fillMaxSize()
                ) { innerPadding ->
                    Greeting(
                        name = "Android",
                        modifier = Modifier
                            .padding(innerPadding)
                    )
                }
            }
        }
    }
}

@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
    Text(
        text = "Hello $name!",
        modifier = modifier
    )
}

@Preview(showBackground = true)
@Composable
fun GreetingPreview() {
    MyApplicationTheme {
        Greeting("Android")
    }
}

```

The import statements include the libraries and the source of what this activity uses. The class `MainActivity : ComponentActivity()` class header creates a class that extends `ComponentActivity`. In Kotlin, the `:` character is used for both deriving from a class (also known as inheritance) and implementing an interface.

`ComponentActivity` is the foundational base class for activities that provides commonly used component and feature support for Android development.

Android activities have many callback functions you can override at different points of the activity's life. This is known as the **activity lifecycle**. For this activity, as you want to display a screen with a layout, you override the `onCreate` function, as shown here:

```
override fun onCreate(savedInstanceState: Bundle?)
```

The `override` keyword in Kotlin specifies that you are providing a specific implementation for a function defined in the parent class. The `fun` keyword (as you may have guessed) stands for *function*. The `savedInstanceState: Bundle?` parameter is Android's mechanism for restoring previously saved state. For this simple activity, you haven't stored any state, so this value will be `null`. The question mark, `?`, that follows the type declares that this type can be `null`.

The `super.onCreate(savedInstanceState)` line calls through to the overridden method of the base class, and finally, `setContent {...}` loads the content we want to display.

`enableEdgeToEdge()` allows your app to use the largest screen display it can to create an immersive experience for your app.

`MyApplicationTheme` applies the material theme hierarchically to the composables contained within it. The first composable is `Scaffold`, which provides the structure of how the screen is made up. It can contain top bars, bottom bars, and other features you will learn about in subsequent chapters. Currently, it just holds an example composable called `Greeting`, which displays a `Text` composable. `innerPadding` accounts for managing the padding between the container, which is `Scaffold` and its content here. This is usually to ensure that the content does not overlap with UI elements, such as the status bar.

The `Modifier` object enables you to add a host of appearance and behaviors to your composables, and the `@Preview` annotation enables you to preview the result of those changes without having to run the app up. In the top-right corner of the main window, there are icons that change the view of the code editor to see solely code (the left-hand icon), half code and half previews (the middle icon), and just the previews (the right-hand icon).



Figure 1.21 – Code editing view icons

Let's have a look at some other files (*Figure 1.21*) present in the folder structure:

- `ExampleInstrumentedTest`: This is an example UI test. You can check and verify the flow and structure of your app by running tests on the UI when the app is running.
- `ExampleUnitTest`: This is an example unit test. An essential part of creating an Android app is writing unit tests to verify that the source code works as expected.
- `ic_launcher_background.xml` and `ic_launcher_foreground.xml`: These two files together make up the launcher icon of your app in vector format, which will be used by the `ic_launcher.xml` launcher icon file in Android API 26 (Oreo) and above.

The `ic_launcher.webp` files are the `.webp` launcher icons that have an icon for every different density of devices. This image format was created by Google and has greater compression compared to the `.png` images. As the minimum version of Android we are using is *API 24: Android 7.0 (Nougat)*, these `.webp` images are included, as support for the launcher vector format was not introduced until Android API 26 (Oreo).



Google created density bucket resources so that the correct image would be selected to be displayed depending on how many **dots per inch (DPI)** the device has. These density bucket resources are created as folders that contain the resources for the different densities. As well as density, the Android SDK supports using bucket resources for device localization, screen sizes, and more. See the full details at the link here: <https://packt.link/KIyHQ>.

The different density qualifiers and their details are as follows:

- `nodpi`: Density-independent resources
- `ldpi`: Low-density screens of 120 DPI
- `mdpi`: Medium-density screens of 160 DPI (the baseline)
- `hdpi`: High-density screens of 240 DPI
- `xhdpi`: Extra-high-density screens of 320 DPI
- `xxhdpi`: Extra-extra-high-density screens of 480 DPI
- `xxxhdpi`: Extra-extra-extra-high-density screens of 640 DPI
- `tvdpi`: Resources for televisions (approx. 213 DPI)

The baseline density bucket was created at 160 DPI for medium-density devices and is called `mdpi`. This represents a device where an inch of the screen is 160 dots/pixels, and the largest display bucket is `xxxhdpi`, which has 640 DPI. Android determines the appropriate image to display based on the individual device.

The Pixel 9 virtual device has a density of 422 DPI, so it uses resources from the extra-extra-high-density bucket (`xxxhdpi`), which is the closest match. Android has a preference for scaling down resources to best match density buckets, so a device with 400 DPI, which is halfway between the `xhdpi` and `xxhdpi` buckets, is likely to display the 480 DPI asset from the `xxhdpi` bucket. The terms `dpi` and `dp` are used interchangeably to mean DPI.

To create alternative bitmap drawables for different densities, you should follow the 3:4:6:8:12:16 scaling ratio between the six primary densities. For example, if you have a bitmap drawable that's 48x48 pixels for medium-density screens, all the different sizes should be as follows:

- 36x36 (0.75x) for low density (`ldpi`)
- 48x48 (1.0x) for medium density (`mdpi`)
- 72x72 (1.5x) for high density (`hdpi`)
- 96x96 (2.0x) for extra-high density (`xhdpi`)
- 144x144 (3.0x) for extra-extra-high density (`xxhdpi`)
- 192x192 (4.0x) for extra-extra-extra-high density (`xxxhdpi`)

For a comparison of these physical launcher icons per density bucket, refer to the following table:

<code>mdpi</code>	<code>hdpi</code>	<code>xhdpi</code>	<code>xxhdpi</code>	<code>xxxhdpi</code>
				

Figure 1.22 – Comparison of principal density bucket launcher image sizes



Launcher icons are made slightly larger than normal images within your app, as they will be used by the device's launcher. As some launchers can scale up the image, this ensures there is no pixelation or blurring of the image.

Now, you are going to look at some of the resources the app uses. These can be referenced in code, as you have seen in the `Color.kt` file for creating colors to set up `MaterialTheme` in Jetpack Compose. They can also be created in XML to be used by the legacy XML UI display.

In the `colors.xml` file, you define the colors you want to use in hexadecimal format for XML layouts:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <color name="purple_200">#FFBB86FC</color>
  <color name="purple_500">#FF6200EE</color>
  <color name="purple_700">#FF3700B3</color>
  <color name="teal_200">#FF03DAC5</color>
  <color name="teal_700">#FF018786</color>
  <color name="black">#FF000000</color>
  <color name="white">#FFFFFFFF</color>
</resources>
```

The only way they differ from the colors defined in `Color.kt` is that the transparency is created with two characters and does not contain the leading `0x`.

If no transparency is required, you can omit the first two characters. So, to create fully blue and 50% transparent blue colors, here's the format:

```
<color name="colorBlue">#0000FF</color>
<color name="colorBlue50PercentTransparent">#770000FF</color>
```

The `strings.xml` file lists all the text displayed in the app:

```
<resources>
  <string name="app_name">My Application</string>
</resources>
```

You can use hardcoded strings in your app, but this leads to duplication. By adding strings as resources, you can also update the string in one place if it is used in different places in the app.

Common styles you would like to use in the XML UI display throughout your app are added to `themes.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <style
    name="Theme.MyApplication"
```



```
        parent="android:Theme.Material.Light.NoActionBar"  
    />  
</resources>
```

You've now explored the core areas of the app. In the next exercise, you will be introduced to UI elements allowing the user to interact with your app.

Exercise 1.05 – Building a Jetpack Compose UI to display a bespoke greeting to the user

The goal of this exercise is to add the capability for users to add and edit text and then submit this information to display a bespoke greeting with the entered data. You will need to add editable text composables to achieve this. The `TextField` composable is typically how this is done. Let's look at an example and see how it renders on the screen:

```
var firstName by remember { mutableStateOf("") }  
TextField(  
    value = firstName,  
    onChange = {firstName = it},  
    label = { Text("First Name") },  
)
```

This uses the default Material style to display a title, as shown in *Figure 1.23*:



Figure 1.23 – TextField with a label

There are a few fundamental concepts of Compose to explore here. First, is the concept of **state**, which, in Compose, is any value that is subject to change. When a property has a state, then changes to that state cause a recomposition.

Recomposition is a redrawing of the UI in response to state changes. Depending on the state that is updated, the whole UI may not be redrawn. Compose evaluates the parts of the UI that are affected by the change in state and only redraws the necessary composables that render the state.

In order to tell Compose that there is a state that has to be observed for changes, you have to create these state objects. `mutableStateOf("Alex")` creates one of the state objects with an initial value of Alex. The mutable naming signifies it is subject to change.

Returning to the `TextField` example, every time the value of this state changes, the UI updates. So, within the `onValueChange` parameter, every time a character in `TextField` is added or deleted, the UI will recompose.

The `remember` function in `Compose` works hand in hand with `MutableState`. Changes to `State` trigger a recomposition, but the value will not be remembered and the UI updated unless the `remember` function is used. `MutableState` is a state holder, which has one property named `value`. You can access the `value` property using the `by` keyword, which delegates getting and setting the value to `MutableState`:

```
var firstName by remember { mutableStateOf("") }
firstName = "Alex"
println(firstName)
```

If `remember` is not used, then only the initial value of the state will be shown, which happens when the composition happens for the first time.

Every time a user interacts with `TextField` by entering text, the value of `firstName` is updated. This is done with the `onValueChange` parameter. It's a callback that is triggered whenever the text changes. Within `onValueChange`, the parameter named `it` is the value of the updated text. In the preceding example, we are updating the `firstName` property with this field.

There is another editable text composable called `OutlinedTextField`, which adds some more attractive styling with an outline around the text:

```
var firstName by remember { mutableStateOf("Alex")}
OutlinedTextField(
    value = firstName,
    onValueChange = {firstName = it},
    label = { Text("First Name") },
)
```

The output is as follows:



Figure 1.24 – `OutlinedTextField` material with a hint

You will now change the default Greeting composable text in your app so a user can enter their first and last name and display a greeting by pressing a button. To do this, perform the following steps:

1. Create a new Android Studio project, as you did in *Exercise 1.01 – Creating an Android Studio project for your app*, called *My Application*.
2. Create the labels and text you are going to use in your app by adding these entries to app | src | main | res | values | strings.xml:

```
<string name="first_name">First name:</string>
<string name="last_name">Last name:</string>
<string name="enter_button">Enter</string>
<string name="welcome_to_the_app">Welcome to the app
</string>
<string name="please_enter_a_name">Please enter a full name!</
string>
```

3. Next, add the required imports to the end of the import list in MainActivity:

```
import androidx.compose.runtime.getValue
import androidx.compose.runtime.setValue
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.ui.res.stringResource
import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.padding
import androidx.compose.material3.Button
import androidx.compose.material3.OutlinedTextField
import androidx.compose.ui.Alignment
import androidx.compose.ui.platform.LocalContext
import androidx.compose.ui.unit.dp
```

4. Create three properties to store the first name, last name, and full name at the top of the Scaffold composable and remove the Greeting composable:

```
var firstName by remember { mutableStateOf("") }
var lastName by remember { mutableStateOf("") }
var fullName by remember { mutableStateOf("") }
```

5. Next, add the two string resources, which will be used to display the welcome message and also the error text. In Compose, in order to access a string resource, you use the `stringResource` composable, passing in the ID of the string you want to retrieve from the `strings.xml` file:

```
var welcomeMessage =
    stringResource(id = R.string.welcome_to_the_app)
var enterNameErrorMessage =
    stringResource(id = R.string.please_enter_a_name)
```

6. Next, we need to add a layout composable to arrange how we add the content. Add a `Column` layout composable to display the content in a column:

```
Column(
    modifier = Modifier
        .fillMaxSize()
        .padding(innerPadding)
        .padding(16.dp),
    horizontalAlignment =
        Alignment.CenterHorizontally,
    verticalArrangement = Arrangement.spacedBy(16.dp)
) {
    // Content goes here
}
```

`Column` has a `Modifier` extension function of `fillMaxSize()` to make the column fill all the available height and width. `innerPadding` sets the content area of the app padding, taking the Android system UI into account, such as the status bar. This is followed by adding padding, which is the space between the inside of the composable and its content, which should be `16.dp`. Modifiers serve as fundamental building blocks of Compose UIs. They are used to decorate and also modify a composable's behavior and appearance. Successive changes can be added by chaining some `Modifier` properties (with the `.` notation), so setting these as individual named arguments is not required, making the code more concise.

`horizontalAlignment` is the horizontal alignment of the content within the `Column` layout, which is centered. `verticalArrangement` is how the content is spaced from top to bottom. In this case, the distance between the embedded composables will be 16 DPI.

7. Add two `OutlinedTextField` composables for the first name and last name in the `Column` layout composable displaying the `firstName` and `lastName` `MutableState` properties, and update the mutable properties as text gets entered:

```
OutlinedTextField(
    modifier = Modifier.fillMaxWidth(),
    value = firstName,
    onValueChange = { firstName = it },
    label = {
        Text(
            text = stringResource(
                id = R.string.first_name
            )
        )
    },
)
OutlinedTextField(
    modifier = Modifier.fillMaxWidth(),
    value = lastName,
    onValueChange = { lastName = it },
    label = {
        Text(
            text = stringResource(
                id = R.string.last_name
            )
        )
    }
)
```

8. Now, we add the interaction to display a welcome message with validation of the text input. Add a button with behavior to display the user's full name and welcome them to the app, or show an error message if either of the `OutlinedTextField` composables are empty:

```
val context = LocalContext.current
Button(
    modifier = Modifier.fillMaxWidth(),
    onClick = {
        if (
```

```
        firstName.isNotBlank() &&
        lastName.isNotBlank())
        fullName = "$firstName $lastName"
    else {
        fullName = ""
        val toast = Toast.makeText(
            context,
            enterNameErrorMessage,
            Toast.LENGTH_LONG
        )
        toast.setGravity(Gravity.CENTER, 0, 0)
        toast.show()
    }
}
) {
    Text("Enter")
}
if (fullName.isNotBlank()) {
    Text(text = "$welcomeMessage $fullName!")
}
}
```

The code block within `onClick` here is a callback of the action that happens when the user clicks the button. It checks that `firstName` and `lastName` are not blank, which means no characters have been entered, and also that no whitespace characters, such as tabs or spaces, have been entered. If the validation is successful, then the welcome message is formatted with Kotlin's string templates, `Text(text = "$welcomeMessage $fullName!")`, which enables text to be evaluated and output using the `$` (dollar) sign.

9. If the form fields have not been filled in correctly, then `fullName` is cleared and a Toast message is displayed:

```
    else {
        fullName = ""
        val toast = Toast.makeText(
            context,
            enterNameErrorMessage,
            Toast.LENGTH_LONG
        )
```

```
toast.setGravity(Gravity.CENTER, 0, 0)
toast.show()
}
```

The Toast object specified is a small text dialog that appears above the main layout for a short time to display a message to the user if validation fails.

10. Run the app, enter text into the fields, and verify that a greeting message is shown when both text fields are filled in, and a pop-up message appears with why the greeting hasn't been set if both fields are not filled in. You should see the following display for each one of these cases:

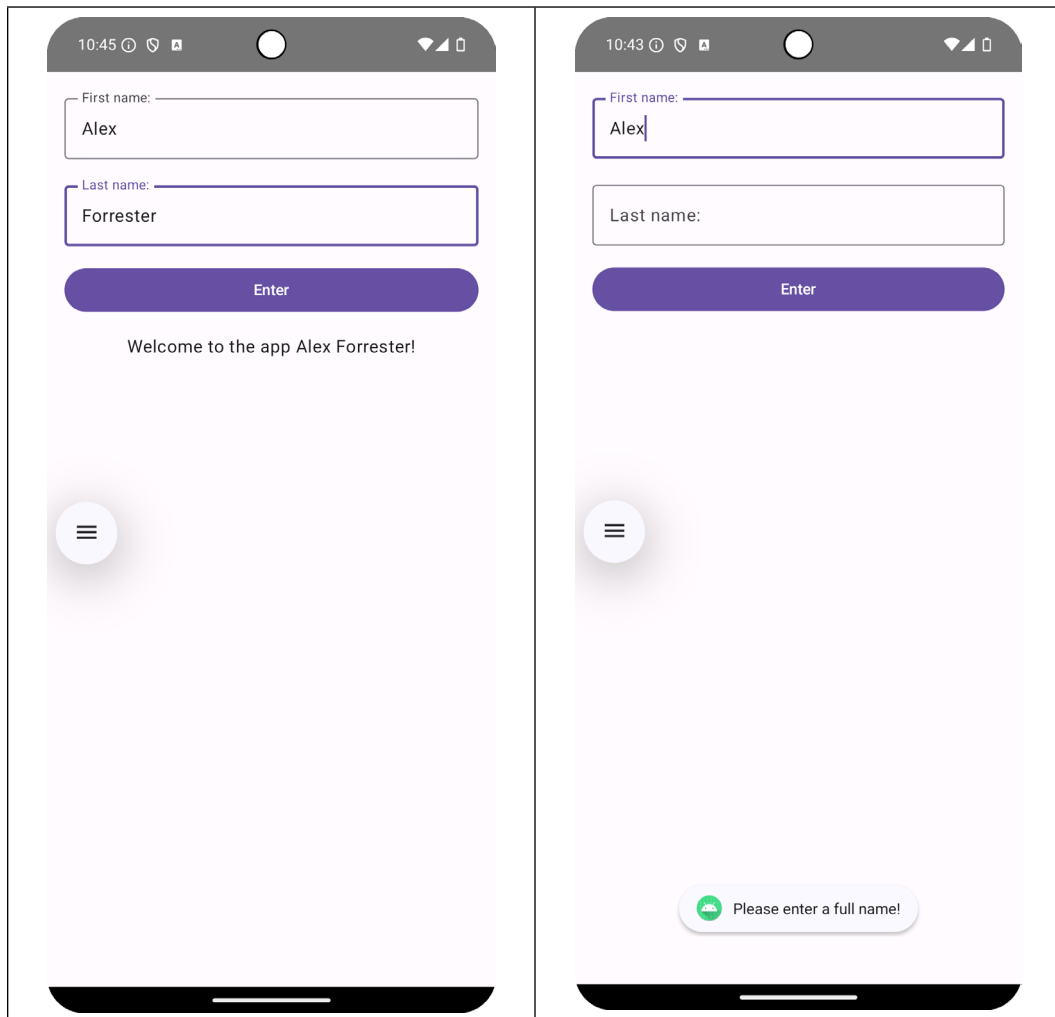


Figure 1.25 – The app with the name filled in correctly and with an error

The preceding exercise introduced you to adding interactivity to your app with `OutlinedTextField`, adding `clicklistener` to respond to button events, and performing some validation.

With the knowledge gained from the chapter, let's start with the following activity.

Activity 1.01 – Producing an app to create RGB colors

In this activity, we will look into a scenario that uses validation. Suppose you have been tasked with creating an app that shows how the RGB channels of red, green, and blue are added together in the RGB color space to create a color.

Each RGB channel should be added as two hexadecimal characters, where each character can be a value of 0–9, A–F, or a–f. The values will then be combined to produce a six-character hexadecimal string that is displayed as a color within the app.

This activity aims to produce a form with editable fields in which the user can add two hexadecimal values for each color. After filling in all three fields, the user should click a button that takes the three values and concatenates them to create a hexadecimal color string. This should then be converted to a color and displayed in the UI of the app.

The following steps will help you complete the activity:

1. Create a new Android Studio project, as you did in *Exercise 1.01 – Creating an Android Studio project for your app*.
2. Add a title Text composable.
3. Add a brief description telling the user how to complete the form.
4. Add three `MutableState` properties for each of the three colors and another for a default color with `mutableStateOf(androidx.compose.ui.graphics.Color.White)`.
5. Add three material `OutlinedTextField` composables with labels of Red Channel, Green Channel, and Blue Channel, initialized with an empty string.
6. Add a restriction to each field to allow entry of only two hexadecimal characters.

You can achieve this with the following function:

```
fun isValidHexInput(input: String): Boolean {  
    return input.filter {  
        it in '0'..'9' ||  
        it in 'A'..'F' ||  
        it in 'a'..'f'  
    }.length == 2  
}
```


7. Add a button that takes the inputs from the three color fields.
8. Add a view that displays the produced color in the layout. This can be achieved by creating a color string starting with a # character and then using Kotlin string templates to concatenate the colors together.
9. Convert the string to a color using `Color(colorString.toColorInt())`, and set it as the background of Text with `Modifier.background(colorToDisplay).padding(24.dp)`.
10. Finally, display the RGB color created from the three channels in the layout. The final output should look like this (the color will vary depending on the inputs):

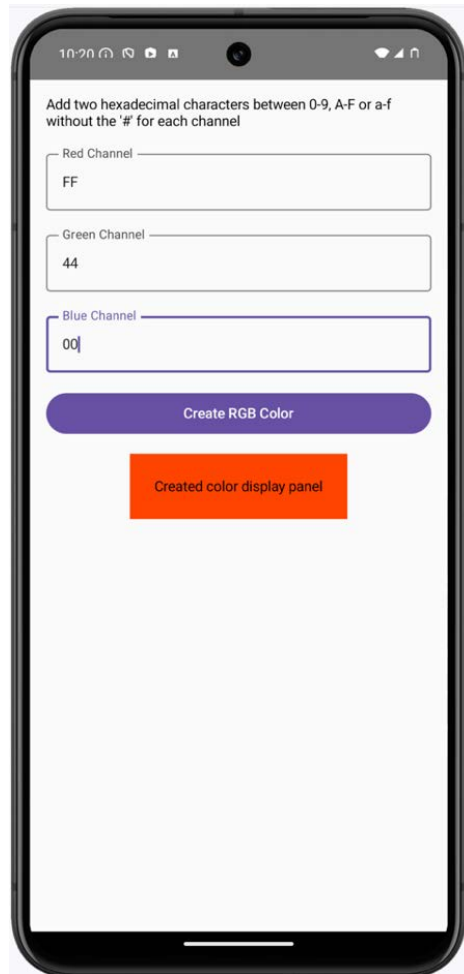


Figure 1.26 – Output when the color is displayed

The solution to this activity can be found at <https://packt.link/Y0uCY>.

Summary

This chapter has covered a lot about the foundations of Android development. You started with how to create Android projects using Android Studio and then created and ran apps on a virtual device.

The chapter then progressed by exploring the `AndroidManifest` file, which details the contents of your app and the permission model, followed by an introduction to Gradle and the process of adding dependencies and building your app.

This was then followed by going into the details of an Android application and the files and folder structure. Jetpack Compose was introduced, and exercises were iterated to illustrate how to construct UIs with an introduction to Google's Material Design.

The next chapter will build on this knowledge by learning about the activity lifecycle, activity tasks, and launch modes, persisting and sharing data between screens, and how to create robust user journeys through your app.

Unlock this book's exclusive benefits now

Scan this QR code or go to packtpub.com/unlock, then search this book by name.

Note: Keep your purchase invoice ready before you start.

UNLOCK NOW



2

Building User Screen Flows

In this chapter, you'll learn how the Android system interacts with your app through the Android lifecycle, how you are notified of changes to your app's state, and how you can use the Android lifecycle to respond to these changes.

You will start developing the UI with Jetpack Compose and use some of the core composables and layout groups to achieve this.

You'll progress to learning how to create user journeys through your app and how to share data between screens. You'll be introduced to different techniques to achieve these goals so that you'll be able to use them in your own apps and recognize them when you see them used in other apps.

You'll also learn how activities and launch modes function and how to save and restore the state of your activity, use logs to report on the flow of the app, and share data between screens.

By the end of the chapter, you will have learned the fundamentals of working with activities and building user screen flows. You will have been shown how to use Jetpack Compose to create a UI, and you will also have learned how to save state and handle changes made by a user's interaction with your apps.

We will cover the following topics in the chapter:

- The activity lifecycle
- Saving and restoring the activity state
- Activity interaction with intents
- Intents, tasks, and launch modes

Technical requirements

The complete code for all the exercises and the activities in this chapter is available on GitHub at <https://packt.link/IGQP7>.

The activity lifecycle

In *Chapter 1, Creating Your First App*, we used the `onCreate(saveInstanceState: Bundle?)` method to display the composable UI for our screen. Now, we'll explore in more detail how the Android system interacts with our application to make this happen. As soon as an activity is launched, it goes through a series of steps to take it through initialization, from preparing to be displayed to being partially displayed and then being fully visible.

There are also steps that correspond with your application being hidden, backgrounded, and then destroyed. This process is called the **activity lifecycle**. For every one of these steps, there is a **callback** that your activity can use to perform actions such as creating and changing the display, saving data when your app has been put into the background, and then restoring that data after your app comes back into the foreground.

These callbacks are made on the parent activity, and it's up to you to decide whether you need to implement them in your own activity to take any corresponding action. Each of these callback functions has the `override` keyword. The `override` keyword in Kotlin means that either this function is providing an implementation of an interface or an abstract method; or, in the case of your activity here, which is a subclass, it is providing the implementation that will override its parent.

Now that you know how the activity lifecycle works in general, let's go into more detail about the principal callbacks you will work with in order, from creating an activity to the activity being destroyed:

- `override fun onCreate(savedInstanceState: Bundle?):` This is the callback that you will use the most for activities that draw a full-sized screen. At this stage, after the method has finished, it is still not displayed to the user, although it will appear that way if you don't implement any other callbacks. You usually set up the UI of your activity here by calling the `setContent{...}` method and carrying out any initialization that is required.

This method is only called once in its lifecycle, unless the activity is created again. This happens by default for some actions (such as rotating the phone from portrait to landscape orientation). The `savedInstanceState` parameter of the `Bundle?` type (`?` means the type can be `null`) in its simplest form is a map of key-value pairs optimized to save and restore data.

It will be `null` if this is the first time that an activity has been run after the app has started, if an activity is being created for the first time, or if an activity is being recreated without any state being saved.

- `override fun onStart()`: When an activity restarts, this is called immediately before `onStart()`. It is important to be clear about the difference between restarting an activity and recreating an activity. When an activity is backgrounded by pressing the **Home** button, when it comes back into the foreground again, `onRestart()` will be called. Recreating an activity is what happens when a configuration change happens, such as the device being rotated. The activity is finished and then created again, in which case, `onRestart()` will not be called.
- `override fun onStart()`: This is the first callback made when an activity is brought from the background to the foreground.
- `override fun onRestoreInstanceState(savedInstanceState: Bundle?)`: If the state has been saved using `onSaveInstanceState(outState: Bundle?)`, this is the method that the system calls after `onStart()`, where you can retrieve the `Bundle` state instead of restoring the state using `onCreate(savedInstanceState: Bundle?)`.
- `override fun onResume()`: This callback is run as the final stage of creating an activity for the first time, and also when the app has been backgrounded and then is brought into the foreground. Upon the completion of this callback, the activity is ready to be used and receive user events.
- `override fun onSaveInstanceState(outState: Bundle?)`: If you want to save the state of an activity, this function can do this for you. You add key-value pairs using one of the convenience functions, depending on the data type. The data will then be available if your activity is recreated in `onCreate(savedInstanceState: Bundle?)` and `onRestoreInstanceState(savedInstanceState: Bundle?)`.
- `override fun onPause()`: This function is called when an activity starts to be backgrounded or another dialog or activity comes into the foreground.
- `override fun onStop()`: This function is called when an activity is hidden, either because it is being backgrounded or another activity is being launched on top of it.
- `override fun onDestroy()`: This is called by the system to kill an activity when system resources are low, when `finish()` is called explicitly on the activity, or, more commonly, when an activity is killed by the user swiping up to close the app from the **Overview** menu.

The flow of callbacks/events is illustrated in the following diagram:

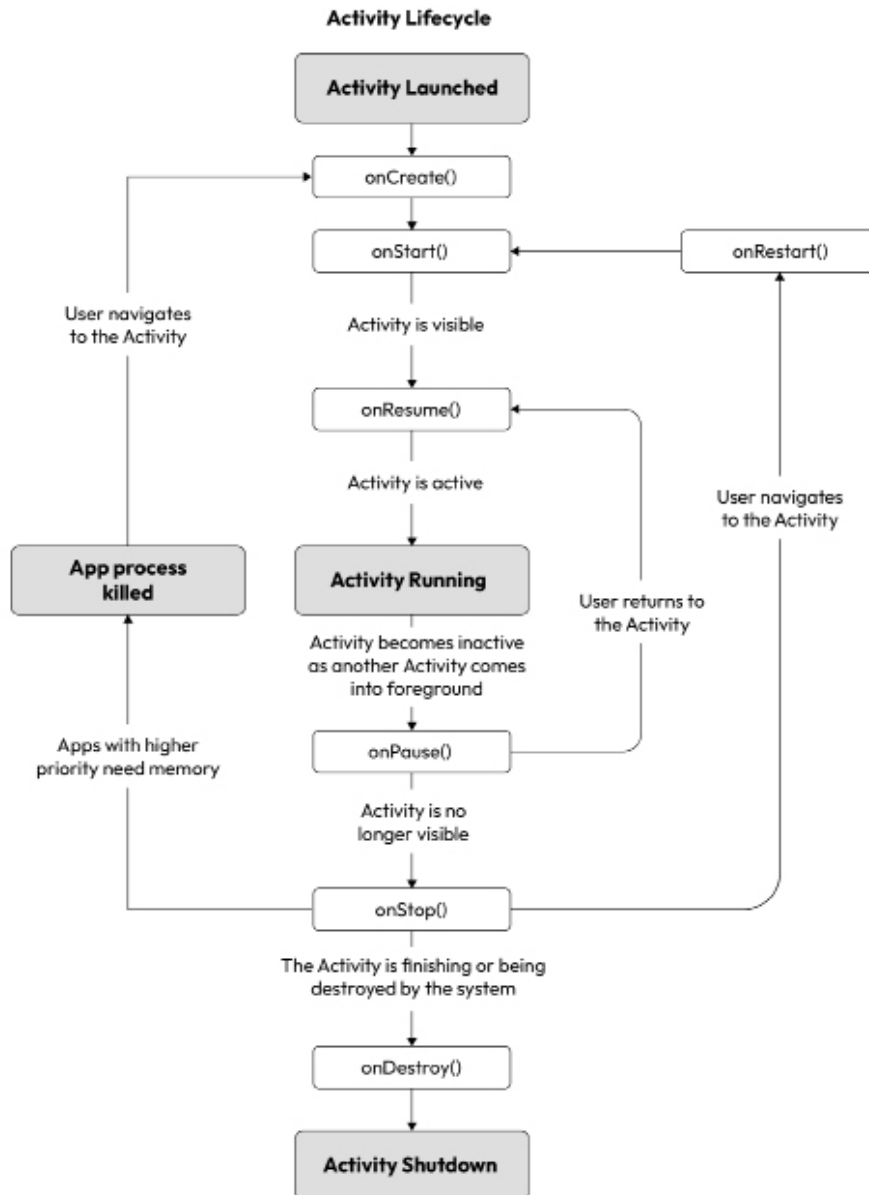


Figure 2.1 – Activity lifecycle

Now that you understand what these common lifecycle callbacks do, let's implement them to see when they are called.

Exercise 2.01 – logging activity callbacks

Create an application called `Activity Callbacks` with an empty activity. The aim of this exercise is to log activity callbacks and the order in which they occur for common operations:

1. To verify the order of callbacks, let's add a log statement at the end of each callback. Open up `MainActivity` and prepare the activity for logging by adding `import android.util.Log` to the import statements. Then, add a constant to the class to identify your activity. Constants in Kotlin are identified by the `const` keyword and can be declared at the top level (outside the class) or in an object within the class.

Top-level constants are generally used if they are required to be public. For private constants, Kotlin provides a convenient way to add static functionality to classes by declaring a companion object. Add the following below `onCreate(savedInstanceState: Bundle?):`

```
companion object {  
    private const val TAG = "MainActivity"  
}
```

2. Then, add a log statement at the end of `onCreate`:

```
Log.d(TAG, "onCreate")
```

Our activity should now have the following code:

```
class MainActivity : ComponentActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        enableEdgeToEdge()  
        setContent {  
            ActivityCallbacksTheme {  
                Scaffold(  
                    modifier = Modifier.fillMaxSize()  
                ) { innerPadding ->  
                    Greeting(  
                        name = "Android",  
                        modifier = Modifier  
                            .padding(innerPadding)  
                    )  
                }  
            }  
        }  
    }  
}
```



```
    }  
    Log.d(TAG, "onCreate")  
}  
  
companion object {  
    private const val TAG = "MainActivity"  
}  
}
```

The `d` instance in the preceding log statement refers to *debug*. Six different log levels can be used to output message information, from the least to most important – `v` for *verbose*, `d` for *debug*, `i` for *info*, `w` for *warn*, `e` for *error*, and `wtf` for *what a terrible failure* (this last log level highlights an exception that should never occur):

```
Log.v(TAG, "verbose message")  
Log.d(TAG, "debug message")  
Log.i(TAG, "info message")  
Log.w(TAG, "warning message")  
Log.e(TAG, "error message")  
Log.wtf(TAG, "what a terrible failure message")
```

3. Now, let's see how the logs are displayed in Android Studio. Open the **Logcat** window. It can be accessed by clicking on the **Logcat** icon in the bottom left of the screen:



Figure 2.2 – Logcat icon

4. You can also access Logcat by going to **View | Tool Windows | Logcat**.
5. Run the app on the virtual device and examine the **Logcat** window output. You should see the log statement you have added formatted like the line shown in *Figure 2.3*:

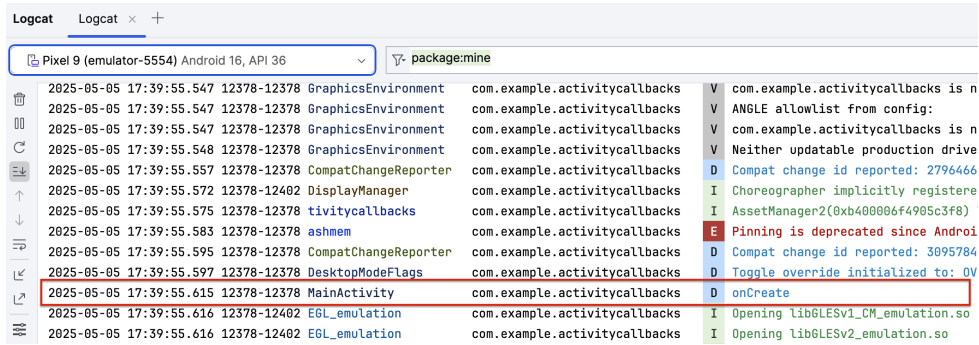


Figure 2.3 – Log output in Logcat

6. Log statements can be quite difficult to interpret at first glance, so let’s break down the following statement into its separate parts:

```
2025-05-05 17:39:55.615 12378-12378 MainActivity com.example.  
activitycallbacks D onCreate
```

7. Let’s examine the elements of the log statement in detail:

Fields	Values
Date	2025-05-05
Time	17:39:55.615
Process identifier and thread identifier (your app process ID and current thread ID)	12378-12378
Class name	MainActivity
Package name	com.example.activitycallbacks
Log level	D (for Debug)
Log message	onCreate

Table 2.1 – Table explaining a log statement

By default, in the log filter (the textbox above the log window), it says package:mime, which is your app logs. You can examine the output of the different log levels of all the processes on the device by changing the log filter from package:mime to level:debug or to other options in the drop-down menu. If you select level:verbose, as the name implies, you will see a lot of output.

8. What's great about the tag option of the log statement is that it enables you to filter log statements that are reported in the **Logcat** window of Android Studio by typing in tag followed by the text of the tag, tag:MainActivity, as shown in *Figure 2.4*:

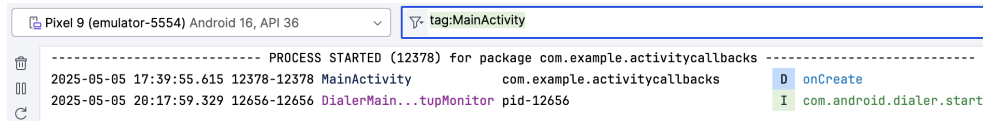


Figure 2.4 – Filtering log statements by the tag name

So, if you are debugging an issue in your activity, you can type in the tag name and add logs to your activity to see the sequence of log statements. This is what you are going to do next by implementing principal activity callbacks and adding a log statement to each one to see when they are run.

9. Place your cursor on a new line after the closing brace of the onCreate(savedInstanceState: Bundle?) function, and then add the onRestart() callback with a log statement. Make sure you call through to super.onRestart() so that the existing functionality of the activity callback works as expected:

```
override fun onRestart() {
    super.onRestart()
    Log.d(TAG, "onRestart")
}
```



In Android Studio, you can start typing the name of a function, and auto-complete options will pop up with suggestions for functions to override. Alternatively, if you go to the top menu and then **Code | Generate | Override methods**, you can select methods to override.

10. Do this for all of the following callback functions:

```
onCreate(savedInstanceState: Bundle?)
onRestart()
onStart()
onRestoreInstanceState(savedInstanceState: Bundle)
onResume()
onPause()
onStop()
onSaveInstanceState(outState: Bundle)
onDestroy()
```

11. Run the app, and then once it has loaded, you should see the following log statements (this is a shortened version):

```
MainActivity onCreate
MainActivity onStart
MainActivity onResume
```

12. Press the round **Home** button in the center of the navigation controls in the emulator window above the virtual device and background of the app. You should now see the following Logcat output:

```
MainActivity onPause
MainActivity onStop
MainActivity onSaveInstanceState
```

13. Now, bring the app back into the foreground by pressing the overview square button in the emulator controls and selecting the app. You should now see the following:

```
MainActivity onRestart
MainActivity onStart
MainActivity onResume
```

The activity has been restarted. You might have noticed that the `onRestoreInstanceState(savedInstanceState: Bundle)` function was not called. This is because the activity was not destroyed or recreated.

14. Press the overview square button again and then swipe the app image upward to kill the activity. This is the output:

```
MainActivity onPause
MainActivity onStop
MainActivity onSaveInstanceState
MainActivity onDestroy
```

15. Launch your app again and then rotate the device using the rotate icons above the emulator, as shown in *Figure 2.5*:

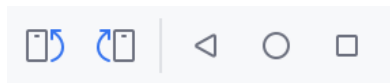


Figure 2.5 – Emulator toolbar with rotate buttons

16. You might find that the phone does not rotate, and the display is sideways. If this happens, drag down the status bar at the very top of the virtual device, look for a button with a rectangular icon with arrows called **Auto-rotate**, and select it:

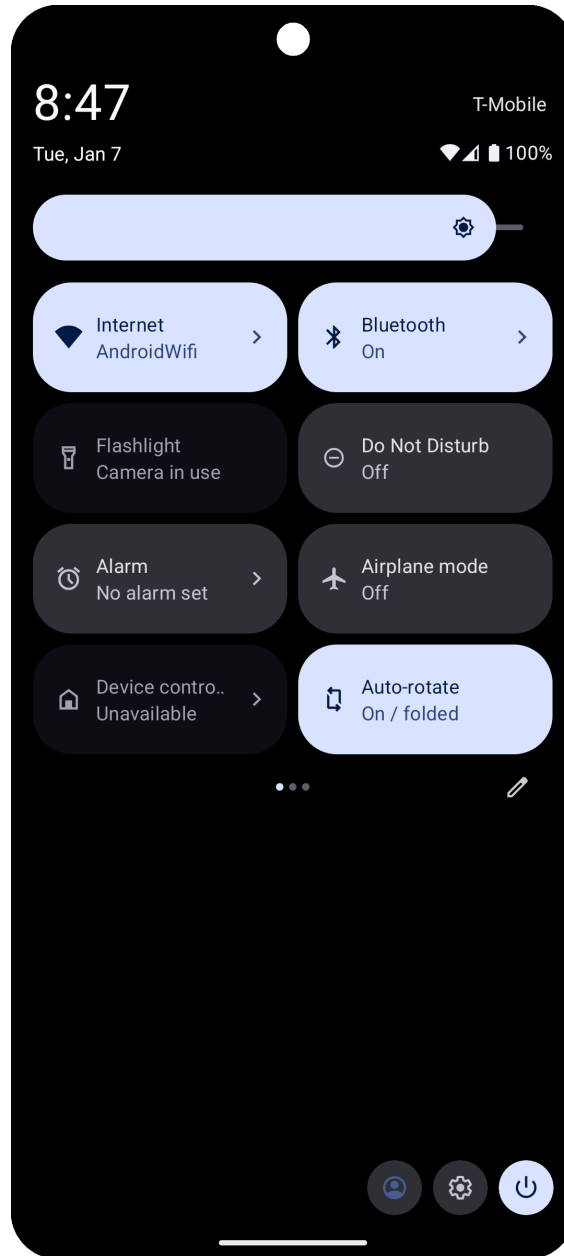


Figure 2.6 – Quick settings bar with the Wi-Fi and Auto-rotate buttons selected

You should see the following callbacks:

```
MainActivity: onPause
MainActivity: onStop
MainActivity: onSaveInstanceState
MainActivity: onDestroy
MainActivity: onCreate
MainActivity: onStart
MainActivity: onRestoreInstanceState
MainActivity: onResume
```

Configuration changes, such as rotating the phone, by default recreate the activity. You can choose not to handle certain configuration changes in the app, which will then not recreate the activity.

17. To not recreate the activity for rotation, add `android:configChanges="orientation|screenSize|screenLayout"` to `MainActivity` in the `AndroidManifest.xml` file. Launch the app and then rotate the phone, and these are the only callbacks you have added to `MainActivity` that you will see:

```
D/MainActivity: onCreate
D/MainActivity: onStart
D/MainActivity: onResume
```

The `orientation` and `screenSize` values have the same function for different Android API levels for detecting screen orientation changes. The `screenLayout` value detects other layout changes that might occur on foldable phones.

These are some of the config changes you can choose to handle yourself (another common one is `keyboardHidden`, to react to changes in accessing the keyboard). The app will still be notified by the system of these changes through the following callback:

```
override fun onConfigurationChanged(
    newConfig: Configuration
) {
    super.onConfigurationChanged(newConfig)
    Log.d(TAG, "onConfigurationChanged")
}
```

If you add this callback function to MainActivity (with `import android.content.res.Configuration`) and you have added `android:configChanges="orientation|screenSize|screenLayout"` to MainActivity in the manifest, you will see it called on rotation. This approach of not restarting the activity is not recommended, as the system will not apply alternative resources automatically. So, rotating a device from portrait to landscape won't apply a suitable landscape layout.

In this exercise, you have learned about principal activity callbacks and how they run when a user carries out common operations with your app through the system's interaction with MainActivity.

In the next section, we will cover saving the state and restoring it, as well as see more examples of how the activity lifecycle works.

Saving and restoring the activity state

In this section, you'll explore how your activity saves and restores the state. As you learned in *Exercise 2.01 – logging activity callbacks*, configuration changes, such as rotating the phone, cause an activity to be recreated.

In such scenarios, it is important to preserve the state of the activity and then restore it. In the next two exercises, you'll work through an example, ensuring that data is restored using a long-established technique to save and restore activity state.

Exercise 2.02 – saving and restoring the state

In this exercise, you are going to create a simple app that generates a random number:

1. Create an application called **Save and Restore** with an empty activity.
2. Open up the `strings.xml` file (located in `app|res|values|strings.xml`) in the **Android** view of the **Project** window or at `app|src|main|res|values|strings.xml` in the **Project** view.
3. Add the following strings, which you'll need for your app:

```
<string name="generate_random_number">
    Generate Random Number
</string>
<string name="random_number_message">
    Random Number: %d
</string>
```

4. Add a property below the class `MainActivity : AppCompatActivity()` header to set the state for the random number and the required imports you'll need in the exercise:

```
import androidx.compose.material3.Button
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.setValue
import androidx.compose.ui.Alignment
import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.ui.res.stringResource
import androidx.compose.ui.unit.dp
import androidx.compose.ui.unit.sp
import kotlin.random.Random

private var randomNumber by mutableStateOf(0)
```

5. Add a function to the class to generate a random number from 0 to 1000:

```
private fun generateRandomNumber(): Int {
    return Random.nextInt(0, 1000)
}
```

6. Replace the Greeting composable with code to generate and display a random number:

```
Column(
    verticalArrangement = Arrangement.spacedBy(10.dp),
    horizontalAlignment = Alignment.CenterHorizontally,
    modifier = Modifier
        .padding(innerPadding)
        .fillMaxWidth(),
) {
    Button(
        onClick = {
            randomNumber = generateRandomNumber()
        }
    ) {
        Text(
            stringResource(
```



```
        id = R.string.generate_random_number
    ),
    fontSize = 18.sp
)
}
Text(
    stringResource(
        id = R.string.random_number_message,
        randomNumber
    ),
    fontSize = 18.sp
)
}
```

The string used to display the random number uses a formatted string resource. This is an Android alternative to using Kotlin's string templates to display variables within text blocks. The reference in the `strings.xml` file is as follows:

```
<string name="random_number_message">
    Random Number: %d
</string>
```

The `%d` instance is a placeholder for a number that is replaced when the string is initialized by passing in the value to be substituted:

```
stringResource(
    id = R.string.random_number_message,
    randomNumber
),
```

To replace a string, you would use a `%s` placeholder.

7. Run the app and click the button. A random number is displayed, as shown in *Figure 2.7*:

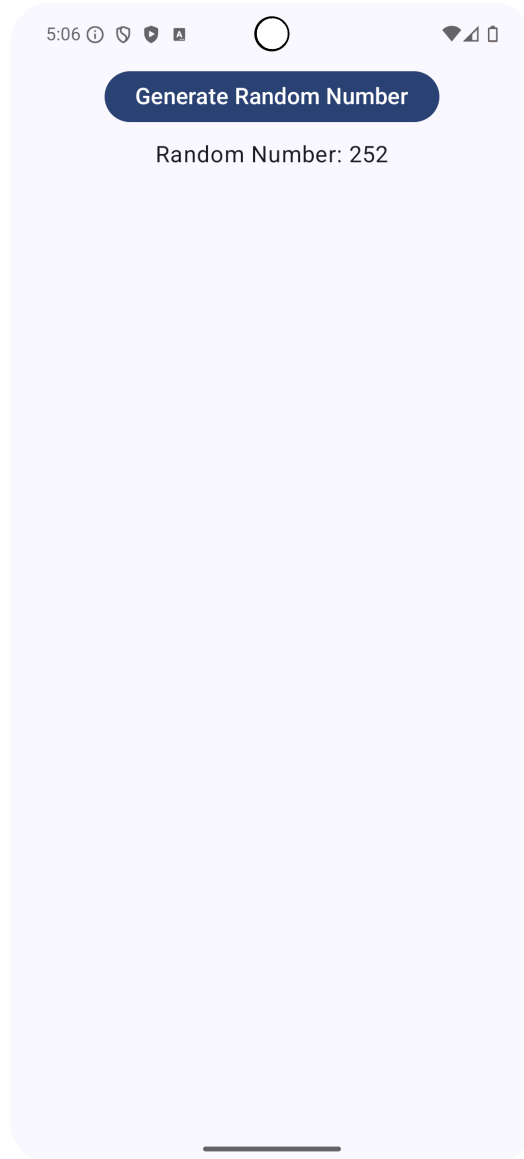


Figure 2.7 – Random number generator initial screen display

You'll notice that if you rotate the device, the number resets to 0. This can be retained by using a combination of `onSaveInstanceState(outState: Bundle)` with either `onCreate(savedInstanceState: Bundle?)` or `onRestoreInstanceState(savedInstanceState: Bundle)`.

8. Add `onSaveInstanceState(outState: Bundle)` and a companion object with a constant:

```
override fun onSaveInstanceState(outState: Bundle) {  
    super.onSaveInstanceState(outState)  
    outState.putInt(RANDOM_NUMBER, randomNumber)  
}  
  
companion object {  
    private const val RANDOM_NUMBER = "RANDOM_NUMBER"  
}
```

The `Bundle` object stores key/value pairs. Here, you are setting the random number value with the key of the `RANDOM_NUMBER` constant.

9. Add the retrieval of the value of the key constant from the `Bundle` object above the `setContent{...}` block:

```
if (savedInstanceState != null) {  
    randomNumber =  
        savedInstanceState.getInt(RANDOM_NUMBER, 0)  
}
```

With this in place, if keys have been set in `Bundle` in `onSaveInstanceState(outState: Bundle)` when the activity was previously finishing, then `Bundle` will not be `null` in `onCreate(savedInstanceState: Bundle?)` and the values can be retrieved using the keys set when the activity is recreated.

In our case, we use the same `RANDOM_NUMBER` constant to get the value and update the `randomNumber` state, which causes the `Text` composable displaying the `randomNumber` state to redraw or recompose. The second argument of `0` (zero) is the default value returned if the key cannot be found. You have two callbacks to retrieve the state once it has been set. If you are doing a lot of initialization in `onCreate(savedInstanceState: Bundle)`, it might be better to use `onRestoreInstanceState(savedInstanceState: Bundle)`. In this way, it's clear which state is being recreated. However, you might prefer to use `onCreate(savedInstanceState: Bundle)` as you are doing here if there is minimal setup required.

10. Run the app again and rotate the device. The activity will be restarted, and the random number will be retained.

You can find the code for the entire exercise at <https://packt.link/LhIjA>.

Exercise 2.03 – saving and restoring state in Compose

Using a `Bundle` object to set key/value pairs that can be saved and restored with activity callbacks is a well-established technique to keep state up to date. It does require a lot of boilerplate code to set up, however. The Jetpack Compose UI toolkit introduces simpler techniques to save and restore state using `rememberSaveable`. In *Chapter 1, Creating Your First App*, you used the `remember` function to remember state when it is updated and recomposed. This function is used to retain state when the data is transient and doesn't need to be restored when the configuration changes. The `rememberSaveable` function differs from `remember` as it can also save and restore state on configuration changes. In the next exercise, we will use it to maintain the state of the form input data on the configuration change of rotating the device.

We will continue using *Exercise 1.05 – adding interactive UI elements to display a bespoke greeting to the user*, from the previous chapter, in this exercise:

1. Open the project and enter values for first name and last name. Click the button labeled **Enter** and then rotate the device. The screen will be blank with no values displayed, as in *Figure 2.8*:

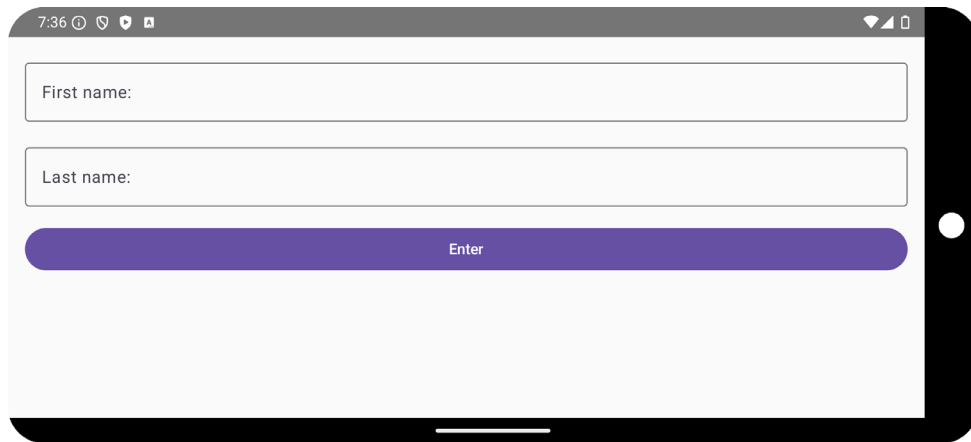


Figure 2.8 – Landscape view of simple form with an empty display

2. Add the `rememberSaveable` import to the list of imports:

```
import androidx.compose.runtime.saveable.rememberSaveable
```

3. Update the `firstName`, `lastName`, and `fullName` properties to use `rememberSaveable`:

```
var firstName by rememberSaveable { mutableStateOf("") }  
var lastName by rememberSaveable { mutableStateOf("") }  
var fullName by rememberSaveable { mutableStateOf("") }
```

4. Launch the app again, complete the form, and then rotate the device. The results should be similar to *Figure 2.9*:

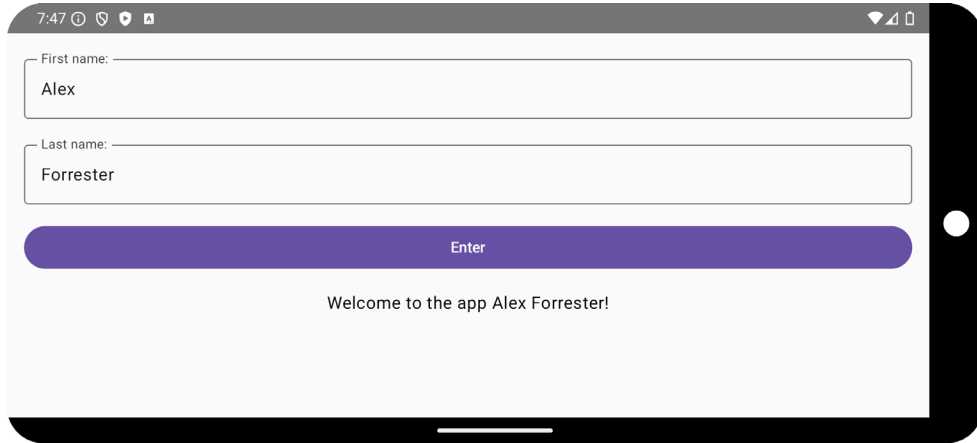


Figure 2.9 – Landscape view of simple form with empty display

The state was preserved when the activity restarted. This single update to use `rememberSaveable` with the `MutableState` property has maintained the correct state. This is a significant improvement on the effort required to use the `Bundle` object in *Exercise 2.02 – saving and restoring the state*. In fact, `rememberSaveable` uses the `Bundle` object under the hood but has the benefit of integrating with Compose’s state management. These functions provide a way to save and restore simple data. The Android framework also provides `ViewModel`, an Android architecture component that is lifecycle-aware. The mechanisms of saving and restoring this state (with `ViewModel`) are managed by the framework, so you don’t have to explicitly manage it as you have done in the exercise using the `Bundle` object. You will learn how to use this component in *Chapter 11, Android Architecture Components*.

The complete code for this exercise can be found here: <https://packt.link/QKQzu>.

In the next section, you will add another activity to an app and navigate between the two activities.

Activity interaction with intents

An **intent** in Android is a communication mechanism between components. Within your own app, you will sometimes want another activity to start when some action happens in the current activity. Specifying exactly which activity will start is called an **explicit intent**. On other occasions, you will want to get access to a system component, such as the camera. As you can't access these components directly, you will have to send an intent, which the system resolves to open the camera. These are called **implicit intents**. An intent filter has to be set up to register to respond to these events. Go to the `AndroidManifest.xml` file from the previous exercise, and you will see an example of two intent filters set within the `<intent-filter>` XML element of `MainActivity`:

```
<intent-filter>
    <action android:name="android.intent.action.MAIN"/>
    <category
        android:name="android.intent.category.LAUNCHER"
    />
</intent-filter>
```

The `<action android:name="android.intent.action.MAIN" />` line means that this is the main entry point into the app. Depending on which category is set, it governs which activity starts first when the app is started. The other intent filter that is specified is `<category android:name="android.intent.category.LAUNCHER" />`, which defines that the app should appear in the launcher. When combined, the two intent filters define that when the app is started from the launcher, `MainActivity` should be started.

For the next exercise, you will see how explicit intents work for navigating around your app.

Exercise 2.04 – an introduction to intents

The goal of this exercise is to create a simple app that uses intents to display text to the user based on their input:

1. Create a new project in Android Studio called `Intents Introduction` and select an empty activity. Once you have set up the project, go to the toolbar and select **File | New | Activity | Gallery | Empty Activity**. Call it `WelcomeActivity` and leave all the other defaults as they are. It will be added to the `AndroidManifest.xml` file, ready to use. The issue you have now that you've added `WelcomeActivity` is knowing how to do anything with it. The `MainActivity` activity starts when you launch the app, but you need a way to launch `WelcomeActivity` and then, optionally, pass data to it, which is when you use intents.

2. In order to work through this example, add the following strings to the `strings.xml` file:

```
<string name="header_text">
    Please enter your name and then we'll get started!
</string>
<string name="welcome_text">
    Hello %s, we hope you enjoy using the app!
</string>
<string name="full_name_label">
    Enter your full name:
</string>
<string name="submit_button_text">SUBMIT</string>
```

3. Open up `MainActivity` and create an empty `@Composable` function called `MainScreen` and another `@Composable` function called `MainScreenPreview` with a `@Preview` annotation to preview the screen you will create:

```
@Composable
private fun MainScreen() {}

@Preview
@Composable
private fun MainScreenPreview() {
    MainScreen()
}
```

4. Add a constant called `FULL_NAME_KEY` in a companion object. (A companion object is a special object that belongs to the class itself rather than instances of the class. It is comparable to Java's static methods and fields.) You will use the `FULL_NAME_KEY` key to set and retrieve the user's name. The `MainActivity` class should now look like this:

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {
            IntentsIntroductionTheme {
                MainScreen()
            }
        }
    }
}
```

```
}  
companion object {  
    const val FULL_NAME_KEY = "FULL_NAME_KEY"  
}  
}
```

5. Add the imports you require for the exercise:

```
import android.content.Intent  
import android.widget.Toast  
import androidx.compose.foundation.layout.Arrangement  
import androidx.compose.foundation.layout.Column  
import androidx.compose.foundation.layout.fillMaxWidth  
import androidx.compose.material3.Button  
import androidx.compose.material3.OutlinedTextField  
import androidx.compose.runtime.getValue  
import androidx.compose.runtime.remember  
import androidx.compose.runtime.mutableStateOf  
import androidx.compose.runtime.setValue  
import androidx.compose.ui.platform.LocalContext  
import androidx.compose.ui.res.stringResource  
import androidx.compose.ui.text.TextStyle  
import androidx.compose.ui.unit.dp  
import androidx.compose.ui.unit.sp  
import com.example.intentsintroduction.MainActivity.Companion  
    .FULL_NAME_KEY
```

6. Next, add the following code to the MainScreen composable, which adds an OutlinedTextField field and a Button component within a Column component:

```
var fullName by remember { mutableStateOf("") }  
val context = LocalContext.current  
  
val welcomeIntent =  
    Intent(context, WelcomeActivity::class.java)  
Scaffold(  
    modifier = Modifier.fillMaxSize()  
) { innerPadding -> Column(  
    verticalArrangement =  
        Arrangement.spacedBy(16.dp),
```



```

        modifier = Modifier
            .padding(innerPadding)
            .padding(16.dp)
            .fillMaxWidth()
    ) {
        OutlinedTextField(
            value = fullName,
            onChange = { fullName = it },
            label = {
                Text(
                    fontSize = 18.sp,
                    text = stringResource(
                        id = R.string.full_name_label
                    )
                )
            },
            textStyle = TextStyle(fontSize = 20.sp),
            modifier = Modifier
                .fillMaxWidth()
        )
        Button(
            onClick = ({
                if (fullName.isNotEmpty()) {
                    welcomeIntent.putExtra(FULL_NAME_KEY, fullName)
                    context.startActivity(welcomeIntent)
                } else {
                    Toast.makeText(
                        context, context.getString(R.string
                            .full_name_label),
                        Toast.LENGTH_LONG
                    ).show()
                }
            })),

            modifier = Modifier
                .fillMaxWidth()
        ) {
            Text(

```

```
        text = stringResource(  
            R.string.submit_button_text  
        )  
    }  
}
```

7. The app, when run, looks as shown in *Figure 2.10*:

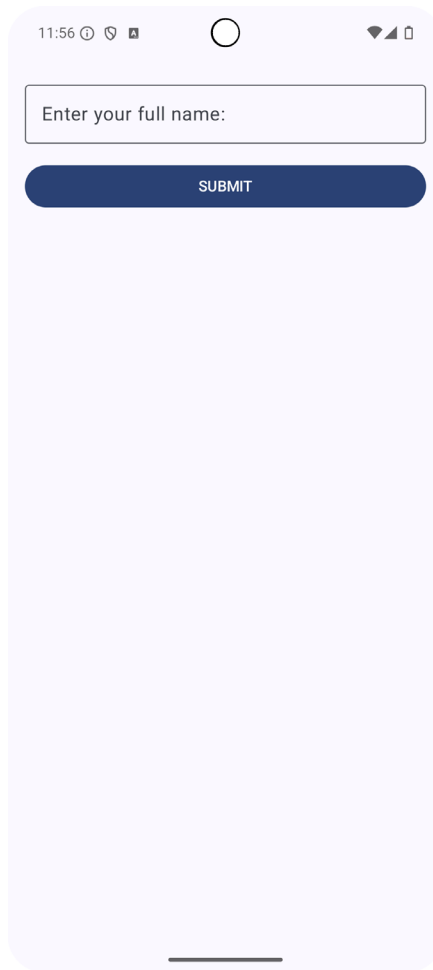


Figure 2.10 – The app display after adding a full name `OutlinedTextField` field and a `SUBMIT` button

8. Let's examine the properties in `MainScreen` and the `onClick` argument:

```
var fullName by remember { mutableStateOf("") }
val context = LocalContext.current
val welcomeIntent =
    Intent(context, WelcomeActivity::class.java)

onClick = ({
    if (fullName.isNotEmpty()) {
        welcomeIntent.putExtra(FULL_NAME_KEY, fullName)
        context.startActivity(welcomeIntent)
    } else {
        Toast.makeText(
            context, context.getString(R.string.full_name_label),
            Toast.LENGTH_LONG
        ).show()
    }
}),
```

A state holder called `fullName` is used to set the value from `OutlinedTextField`, and a `context` is accessed in the `Composable` function with `LocalContext.current`; this is used to create an explicit intent to start `WelcomeActivity`. The `fullName` state holder is verified to ensure that the user has filled this in; a pop-up `Toast` message will be shown if it's blank. Every intent can contain a `Bundle` object, which is an object that can store key-value pairs. The process of adding to and retrieving data from the `Bundle` object uses the term **extra**, which stands for **extended data**. Here, the extra data is set as the `FULL_NAME_KEY` key with the `fullName` value, and then the last step is to use the intent to start `WelcomeActivity`.

9. Now, run the app, enter your name, and press **SUBMIT**, and your screen will load as shown in *Figure 2.11*:

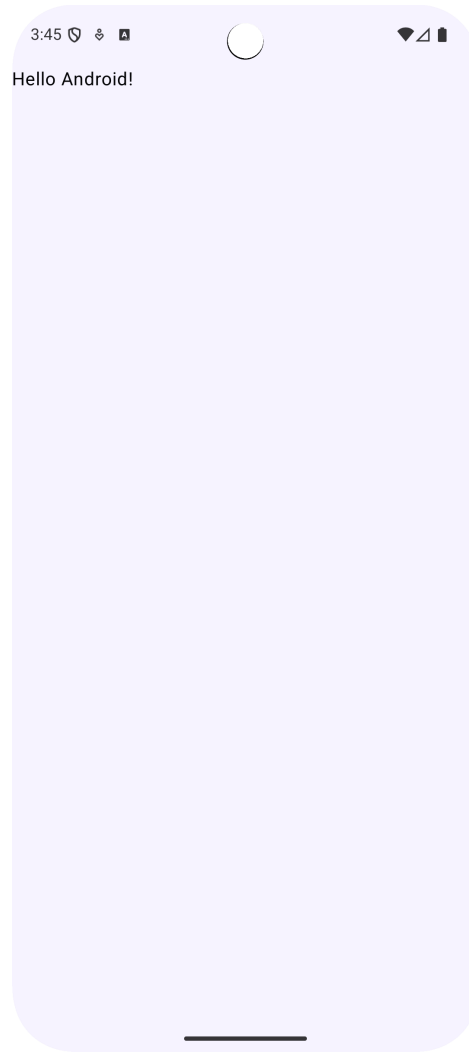


Figure 2.11 – Blank screen displayed when the intent extra data is not processed

Well, that's not very impressive. You've added the logic to send the user's name, but not to display it.

10. Next, add the required imports from *Step 5* into `WelcomeActivity`
11. Next, add a `WelcomeScreen` composable with the following content and replace the Greeting function by calling the `WelcomeScreen` function:

```
IntentsIntroductionTheme {  
    WelcomeScreen(intent)
```

```

    }
    @Composable private fun WelcomeScreen(intent: Intent) {
        Scaffold(
            modifier = Modifier.fillMaxSize()
        ) { innerPadding -> Box( contentAlignment = Alignment.Center,
            modifier = Modifier
                .padding(innerPadding)
                .fillMaxSize()
        ) {
            val fullName =
                intent.getStringExtra(
                    FULL_NAME_KEY
                ) ?: ""
            val welcomeText =
                stringResource(
                    R.string.welcome_text,
                    fullName
                )
            Text(
                textAlign = TextAlign.Center,
                text = welcomeText,
                fontSize = 20.sp,
                fontWeight = FontWeight.Bold,
                modifier = Modifier.padding(12.dp)
            )
        }
    }
}

```

12. We retrieve the intent used to start `WelcomeActivity` and put it into `WelcomeScreen`. We then retrieve the string value that was passed from the intent from the `FULL_NAME_KEY` string. We then format the `<string name="welcome_text">Hello %s, we hope you enjoy using the app!</string>` resource string by getting the string from the resources and passing in the `fullName` value retrieved from the intent. Finally, this is set as the text of `TextView`.

13. Run the app again, and a simple greeting will be displayed, as shown in *Figure 2.12*:

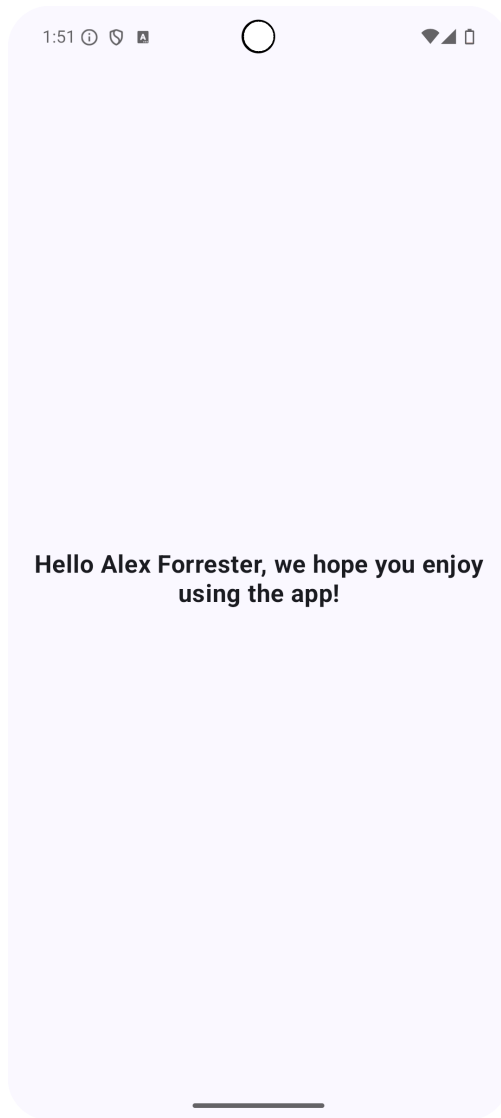


Figure 2.12 – User welcome message displayed

This exercise, although relatively simple in terms of layout and user interaction, demonstrates some of the core principles of intents. You can use them to add navigation and create user flows from one section of your app to another. In the next section, you will see how you can use intents to launch an activity and receive a result from it.

Exercise 2.05 – retrieving a result from an activity

For some user flows, you will only launch an activity for the sole purpose of retrieving a result from it. This pattern is often used to ask permission to use a particular feature, popping up a dialog with a question about whether the user gives their permission to access contacts, the calendar, and so on, and then reporting the result of yes or no to the calling activity. In this exercise, you will ask the user to pick their favorite color of the rainbow, and then, once that is chosen, display the result in the calling activity:



From here on, the imports for the exercises won't be listed. Android Studio has the functionality to add automatic imports by enabling **Editor | General | Auto import | Kotlin | Add unambiguous imports on the fly**. You can also manually trigger imports by placing the cursor on the class or function name and selecting *Option + Enter* on a Mac and *Alt + Enter* on Windows/Linux to automatically add the import. For any issues, you can refer to the full source code of the exercises in the GitHub repository.

Two problematic imports that don't automatically import are those for the `by` keyword. Add these two imports to fix this issue:

- `import androidx.compose.runtime.getValue`
- `import androidx.compose.runtime.setValue`

1. Create a new project named `Activity Results` with an empty activity and add the following strings to the `strings.xml` file:

```
<string name="header_text_main">
    Please click the button below to choose your favorite
    color of the rainbow!
</string>
<string name="header_text_picker">Rainbow Colors</string>
<string name="footer_text_picker">
    Click the button above which is your favorite color
    of the rainbow.
</string>
<string name="color_chosen_message">%s is your favorite color!
</string>
<string name="submit_button_text">CHOOSE COLOR</string>
<string name="red">RED</string>
```

```

<string name="orange">ORANGE</string>
<string name="yellow">YELLOW</string>
<string name="green">GREEN</string>
<string name="blue">BLUE</string>
<string name="indigo">INDIGO</string>
<string name="violet">VIOLET</string>
<string name="title_activity_color_picker">
    ColorPickerActivity</string>

```

2. Create a new activity called `ColorPickerActivity` following the steps at the start of *Exercise 2.04 – an introduction to intents*.
3. Go back to `MainActivity` and add a `Text` composable that displays a background color and the name of the background color, and a `@Preview` composable:

```

@Composable fun TextWithBackgroundColor(backgroundColor: Color,
    colorMessage: String) {
    Row(
        verticalAlignment = Alignment.CenterVertically,
        modifier = Modifier
            .padding(horizontal = 20.dp)
            .height(50.dp)
            .background(backgroundColor)
            .fillMaxWidth()
    ) {
        Text(
            text = colorMessage,
            fontSize = 22.sp,
            color = Color.White,
            textAlign = TextAlign.Center,
            modifier = Modifier
                .background(backgroundColor)
                .fillMaxWidth()
        )
    }
}

@Preview @Composable fun TextWithBackgroundColorPreview() {

```



```
TextWithBackgroundColor(Color(0xFF00FF00),
    "Chosen color appears here")
}
```

This will display the panel of the color the user has chosen as their favorite color of the rainbow.

4. Next, add a `MainScreen` composable that adds a button to navigate to `ColorPickerActivity` and `TextWithBackgroundColor` for the display of the chosen color:

```
@Composable
fun MainScreen(
    backgroundColor: Color,
    colorMessage: String,
    context: Context,
    startForResult: ActivityResultLauncher<Intent>
) {
    Scaffold(
        modifier = Modifier.fillMaxSize()
    ) { innerPadding ->
        Column(
            verticalArrangement = Arrangement.Top,
            modifier = Modifier
                .fillMaxSize()
                .padding(innerPadding),
        ) {
            Text(
                stringResource(
                    id = R.string.header_text_main
                ),
                fontSize = 18.sp,
                textAlign = TextAlign.Center,
                modifier = Modifier.padding(16.dp)
            )
            Button(
                onClick = {
                    val intent = Intent(
                        context,
                        ColorPickerActivity::class.java
                    )
                }
            )
        }
    }
}
```

```

        )
        startForResult.launch(intent)
    },
    modifier = Modifier
        .fillMaxWidth()
        .padding(20.dp)
    ) {
        Text(
            text = stringResource(
                id =
                    R.string.submit_button_text
            )
        )
    }
    TextWithBackgroundColor(
        backgroundColor, colorMessage
    )
}
}
}

```

5. Add the following constants in a companion object: `RAINBOW_COLOR_NAME` to set the rainbow color name in the intent, `RAINBOW_COLOR` to set the hexadecimal value of the color, and `TRANSPARENT` to set a default:

```

companion object {
    const val RAINBOW_COLOR_NAME = "RAINBOW_COLOR_NAME"
    const val RAINBOW_COLOR = "RAINBOW_COLOR"
    const val TRANSPARENT = 0x00FFFFFFL
}

```

6. Add `rainbowColor`, `colorName`, and `colorMessage` properties below the class header:

```

private var rainbowColor by
    mutableStateOf(Color(TRANSPARENT))
private var colorName by mutableStateOf("")
private var colorMessage by mutableStateOf("")

```

These properties are the state that governs the recomposition of `MainScreen`. When one of these properties changes, then `MainScreen` will be recomposed.

7. Add another property, `startForResult`, which is the final part of preparing the activity to receive the result from `ColorPickerActivity`:

```
private val startForResult =
    registerForActivityResult(
        ActivityResultContracts.StartActivityForResult()
    ) { activityResult ->
        val data = activityResult.data

    rainbowColor = Color(
        data?.getLongExtra(
            RAINBOW_COLOR,
            TRANSPARENT
        ) ?: TRANSPARENT)
    colorName =
        data?.getStringExtra(RAINBOW_COLOR_NAME) ?: ""
    colorMessage = getString(
        R.string.color_chosen_message,
        colorName
    )
    data?.getStringExtra(RAINBOW_COLOR_NAME) ?: ""
}
```

8. Finally, update `onCreate` to use the `MainScreen` composable:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    enableEdgeToEdge()
    setContent {
        val context = LocalContext.current
        ActivityResultsTheme {
            MainScreen(
                rainbowColor,
                colorMessage,
                context,
                startForResult
            )
        }
    }
}
```

```

    }
  }
}

```

The `startForResult` property is used to both launch the new activity and return a result from it. The `activityResult` property returned from `ColorPickerActivity` has the intent in the `activityResult.data` property.

You can proceed to query the intent data for the values you are expecting. For this exercise, we want to get the background color name (`colorName`) and the hexadecimal value of the color (`rainbowColor`) so that we can display it. The `?` operator checks whether the value is null (that is, not set in the intent), and, if so, the Elvis operator (`?:`) sets the default value. Also, `colorMessage` uses string formatting to set a message, replacing the placeholder in the resource value with the color name. If any of the data has changed value, then `MainScreen`, which depends on it, will recompose, and the chosen color will be set as the background of the Text field.

9. Now, let's look at what `ColorPickerActivity` does. Firstly, create a `RainbowColor` composable with a preview that adds a button with the name of the color and sets the background value of the color:

```

@Composable
fun RainbowColor(color: Long, colorName: String, onClick:
(Long, String) -> Unit) {
    Button(
        onClick = { onClick(color, colorName) },
        colors = ButtonDefaults.buttonColors(
            containerColor = Color(color), // Background color
            contentColor = Color.Black
        ),
        modifier = Modifier
            .padding(horizontal = 20.dp)
            .fillMaxWidth()
    )
    {
        Text(
            text = colorName,
            color = Color.White,

```

```

        fontSize = 22.sp,
        textAlign = TextAlign.Center,
        fontWeight = FontWeight.Bold
    )
}
}

@Preview
@Composable
fun RainbowColorPreview() {
    RainbowColor(0xFF00FF00, "GREEN") { color, name -> }
}

```

10. The `RainbowColor` composable has `color`, `colorName`, and an `onButtonClick` lambda, which takes the `color` and `colorName` parameters and passes them into the `onButtonClick` function. This is the first time we have used a higher-order function, which is a function that can be passed into or returned from another function. We could achieve the same functionality by directly referencing the function by name in the `Rainbow` composable, but this makes the composable bound to another function and makes it less reusable because the state needs to be passed into the composable. The pattern of elevating the state that controls recomposition higher up in the composable hierarchy and then passing it down as parameters is called *state hoisting*, and it makes composables more reusable as they are either stateless or depend on less state. It also separates the state from the UI logic so that they can be tested more easily. These and other benefits of the pattern focus on the **separation of concerns (SoC)** between the state and the UI composables.
11. Next, add the colors of the rainbow in a companion object in `ColorPickerActivity`:

```

companion object {
    const val RED = 0xFFFF0000L
    const val ORANGE = 0xFFFFA500L
    const val YELLOW = 0xFFFFEE00L
    const val GREEN = 0xFF00FF00L
    const val BLUE = 0xFF0000FFL
    const val INDIGO = 0xFF4B0082L
    const val VIOLET = 0xFF8A2BE2L
}

```

12. Create a `setRainbowColor` function, which will be passed into `RainbowColor` through the `onButtonClick` parameter and set the intent with color values back to `MainActivity`:

```
private fun setRainbowColor(color: Long, colorName: String) {
    Intent().let { pickedColorIntent ->
        pickedColorIntent.putExtra(RAINBOW_COLOR_NAME, colorName)
        pickedColorIntent.putExtra(RAINBOW_COLOR, color)
        setResult(RESULT_OK, pickedColorIntent)
        finish()
    }
}
```

13. The final step is to create a `ColorPickerScreen` composable in the `ColorPickerActivity` class that populates the background color of the `RainbowColor` buttons and passes in a `clickHandler` property that is run on the button click, to set the color value to be returned to `MainActivity` in `setRainbowColor`:

```
@Composable private fun ColorPickerScreen() {
    Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding ->
        Column(
            verticalArrangement = Arrangement.Top,
            modifier = Modifier
                .fillMaxSize()
                .padding(innerPadding),
        ) {

            val clickHandler = { color: Long, colorName: String ->
                setRainbowColor(color, colorName)
            }

            Text(
                stringResource(id = R.string.header_text_picker),
                textAlign = TextAlign.Center,
                fontSize = 24.sp,
                modifier = Modifier
                    .padding(20.dp)
                    .fillMaxWidth()
            )

            RainbowColor(RED, getString(R.string.red), clickHandler)
            RainbowColor(ORANGE, getString(R.string.orange),
```

```

        clickHandler)
    RainbowColor(YELLOW, getString(R.string.yellow),
        clickHandler)
    RainbowColor(GREEN, getString(R.string.green),
        clickHandler)
    RainbowColor(BLUE, getString(R.string.blue),
        clickHandler)
    RainbowColor(INDIGO, getString(R.string.indigo),
        clickHandler)
    RainbowColor(VIOLET, getString(R.string.violet),
        clickHandler)
    Text(
        stringResource(id = R.string.footer_text_picker),
        textAlign = TextAlign.Center,
        fontSize = 18.sp,
        modifier = Modifier
            .padding(12.dp)
            .fillMaxWidth()
    )
}
}
}

```

14. Then, update onCreate by adding a ColorPickerScreen() method:

```

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    enableEdgeToEdge()
    setContent {
        ActivityResultsTheme {
            ColorPickerScreen()
        }
    }
}
}

```

15. Run the app and see it in action. The layout should now look as shown in the following screenshot:

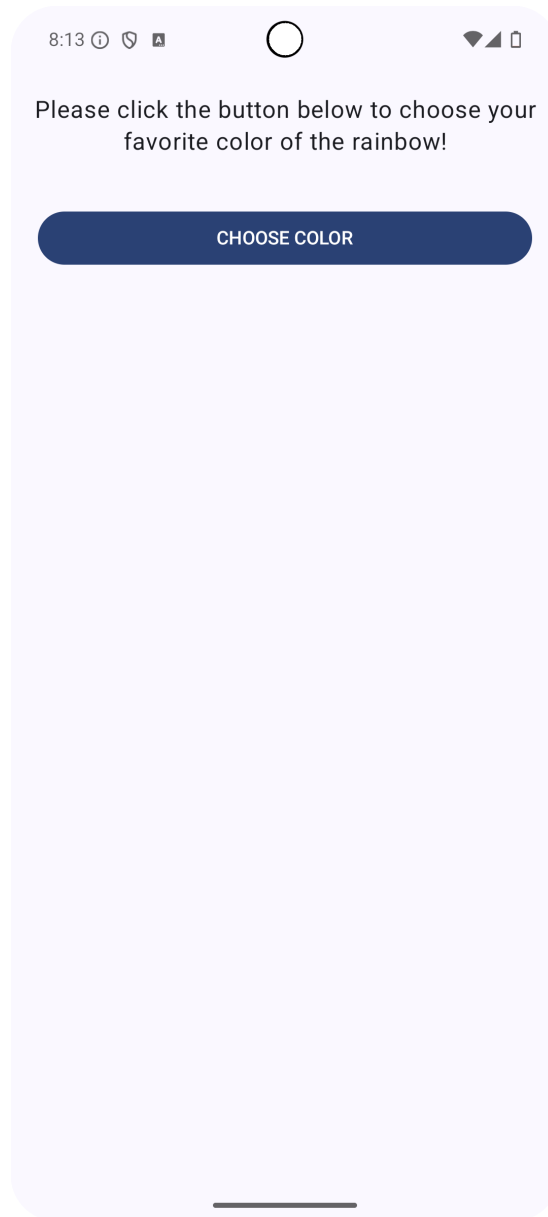


Figure 2.13 – Initial rainbow colors screen

16. Press the **CHOOSE COLOR** button, as shown in *Figure 2.13*:

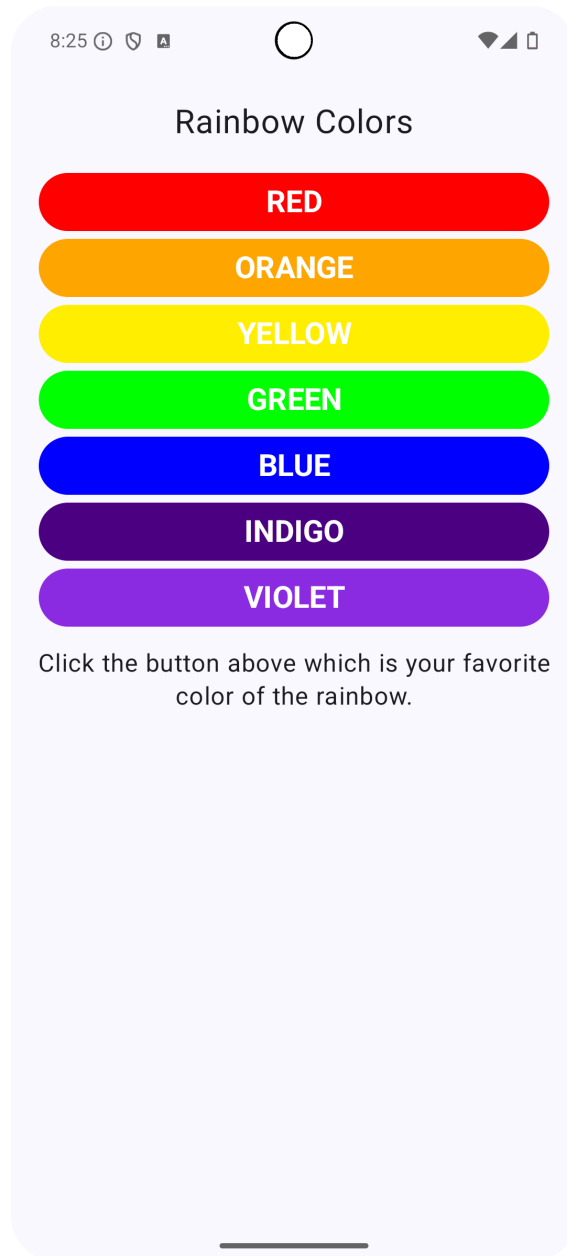


Figure 2.14 – The rainbow colors selection screen

17. If you choose **BLUE**, a screen with this color will be displayed, as shown in *Figure 2.15*:

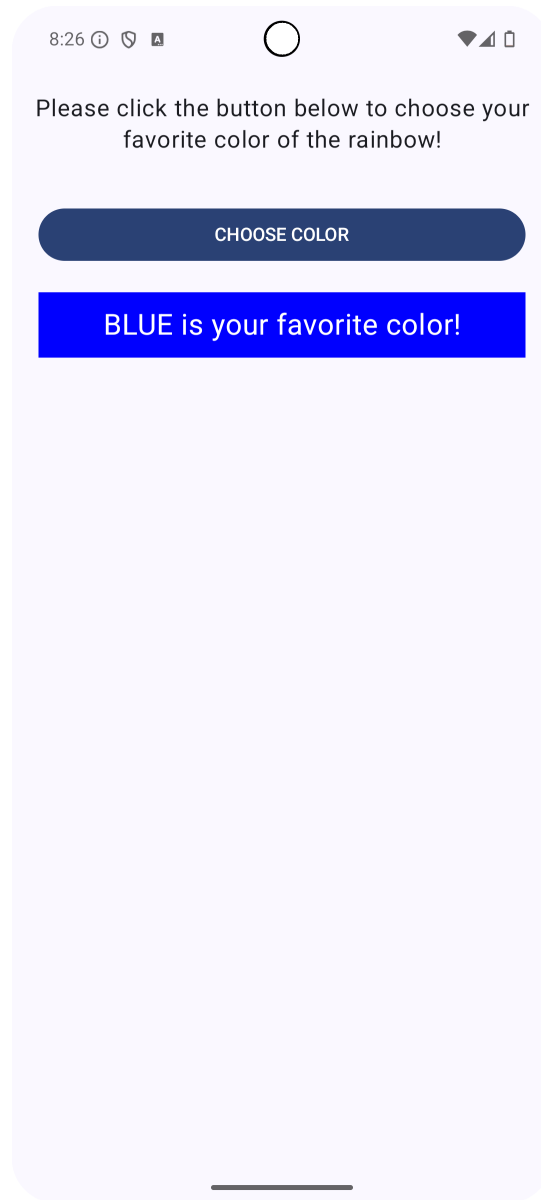


Figure 2.15 – The app displaying the selected color

The full source code for this exercise can be found here: <https://packt.link/Et7nn>.

This exercise introduced you to another way of creating user flows using `registerForActivityResult`. This can be very useful for carrying out a dedicated task where you need a result before proceeding with the user's flow through the app. Next, you will explore launch modes and how they impact the flow of user journeys when building apps.

Intents, tasks, and launch modes

Up until now, you have been using the standard behavior for creating activities and moving from one activity to the next. When you open the app from the launcher with the default behavior, it creates its own task, and each activity you create is added to a back stack, so when you open three activities one after the other as part of your user's journey, pressing the back button three times will move the user back through the previous screens/activities and then go back to the device's home screen, while still keeping the app open.

The launch mode for this type of activity is called `Standard`; it is the default and doesn't need to be specified in the activity element of `AndroidManifest.xml`. Even if you launch the same activity three times, one after the other, there will be three instances of the same activity.

For some apps, you may want to change this behavior so that the same instance is used. The launch mode that can help here is called `singleTop`. If a `singleTop` activity is the most recently added, when the same `singleTop` activity is launched again, it uses the same activity.

There are three other launch modes to be aware of, called `singleTask`, `singleInstance`, and `singleInstancePerTask`. These are not for general use and are only used for special scenarios. Detailed documentation of all launch modes can be viewed here: <https://packt.link/UCY7x>.

You'll explore the differences in the behavior of the `Standard` and `singleTop` launch modes in the upcoming exercise.

Exercise 2.06 – setting the launch mode of an activity

This exercise illustrates the behavior of two of the most commonly used launch modes. Please download the code from here: <https://packt.link/MiUDz>. Let's get started:

1. Open up `AndroidManifest.xml` and examine it. In the activity section of the file, three activities have been added:

```
<activity
    android:name=".StandardActivity"
    android:exported="false"
    android:label="@string/title_activity_standard"
    android:launchMode="standard"
    android:theme="@style/Theme.LaunchModes" /> <activity
    android:name=".SingleTopActivity"    android:exported="false"
    android:label="@string/title_activity_single_top"
    android:launchMode="singleTop"
    android:theme="@style/Theme.LaunchModes" /> <activity
    android:name=".MainActivity"    android:exported="true"
    android:label="@string/app_name"
    android:theme="@style/Theme.LaunchModes">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

The `android:launchMode` instance is what fundamentally distinguishes between `SingleTopActivity` and `StandardActivity`.

2. Run the app, and you will see that **Main Activity** has two buttons to launch the standard activity and single top activity in *Figure 2.16*:

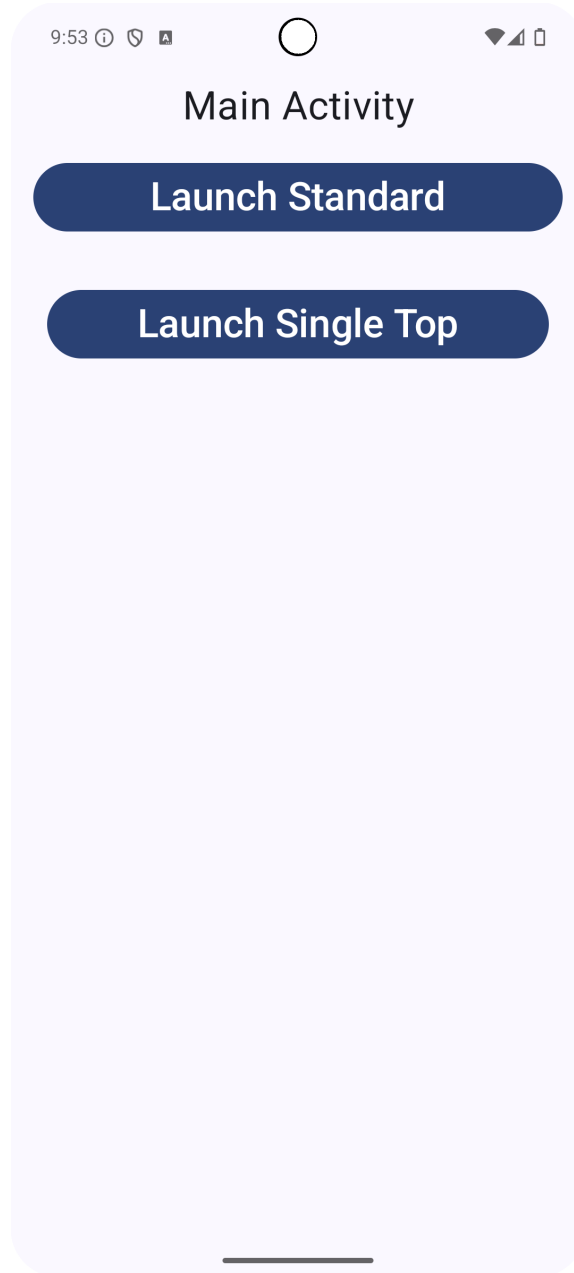


Figure 2.16 – App displaying buttons to launch activities with different launch modes

3. Click the **Launch Standard** button, and a new activity will always be loaded:

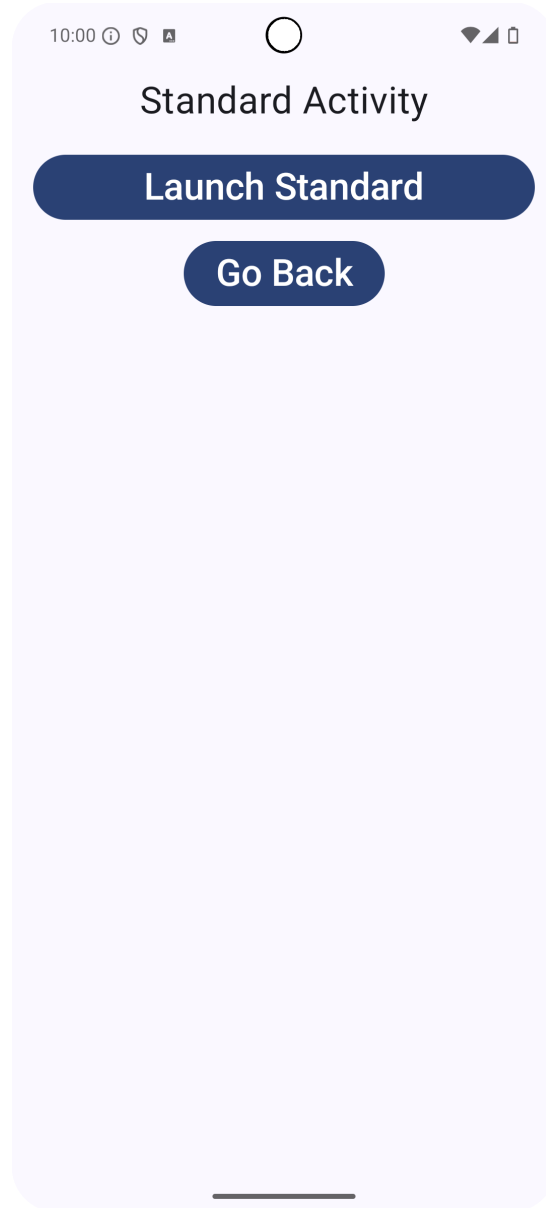


Figure 2.17 – App displaying a standard activity

4. Clicking back will take you back through the number of activities you have launched before you get back to MainActivity.

5. Click the **Launch Single Top** button, and the same activity is always loaded:

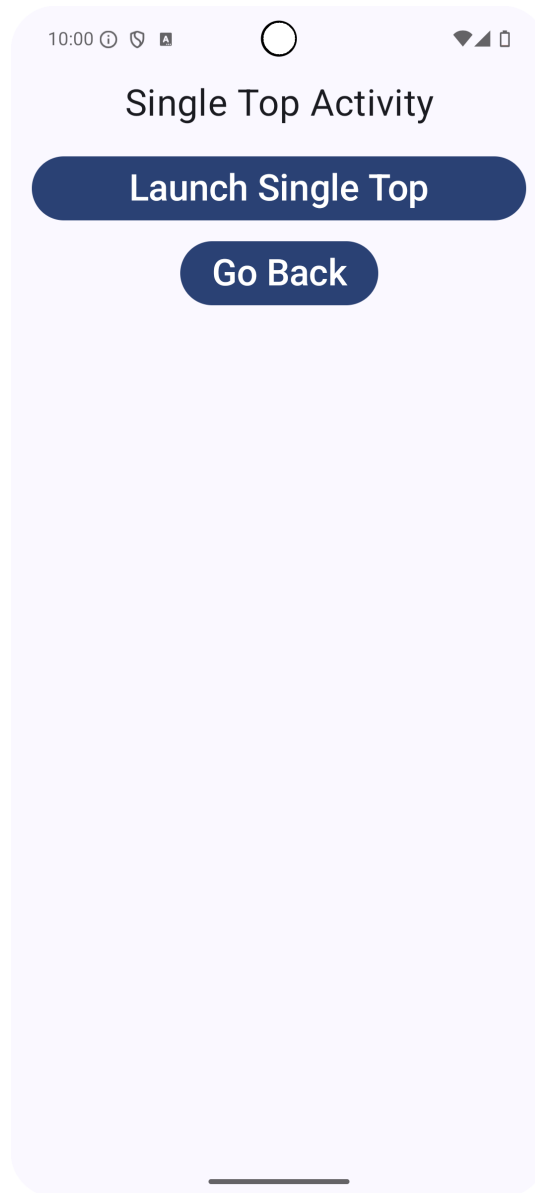


Figure 2.18 – App displaying a single top activity

If a single top activity is at the top of the back stack, then a new activity will not be launched. This can be useful if you don't need to pass any data to the activity as extras in an intent, as you don't have to launch another activity that increases the memory usage of the app.

Activity 2.01 and/or Activity 2.02 – creating a login form

The aim of this activity is to create a login form with username and password fields. Once the values in these fields have been submitted, check these entered values against the hardcoded values and display a welcome message if they match, or an error message if they don't, and return the user to the login form. The steps needed to achieve this are the following:

1. Create a form with username and password `TextField` composables and a login button.
2. Add a `ClickListener` composable to the button to react to a button press event.
3. Validate that the form fields are filled in.
4. Check the submitted username and password fields against the hardcoded values.
5. Display a welcome message with the username if the check is successful and hide the form.
6. Display an error message if the check is not successful and redirect the user back to the form.

There are a few possible ways that you could go about trying to complete this task using activities. Here are two ideas for approaches you could adopt:

- Use a standard activity to pass a username and password to another activity and validate the credentials used in *solution 2.01*.
- Use `registerForActivityResult` to carry out the validation in another activity and then return the result used in *solution 2.02*.



The solutions (2.01 and 2.02) to this activity can be found at <https://packt.link/IGQP7>.

Summary

In this chapter, we covered a lot of the groundwork of how our application interacts with the Android framework, from activity lifecycle callbacks to retaining the state in our activities, navigating from one screen to another, and how intents and launch modes make this happen, all within the context of the Jetpack Compose UI framework. These are core concepts that you need to understand in order to move on to more advanced topics.

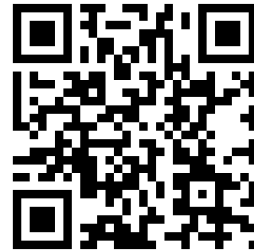
In the next chapter, you will be introduced to the core Jetpack Compose UI components and layout groups, and how they fit into the architecture of your application.

Unlock this book's exclusive benefits now

Scan this QR code or go to packtpub.com/unlock, then search this book by name.

Note: Keep your purchase invoice ready before you start.

UNLOCK NOW



3

Developing the UI with Jetpack Compose

A composable function is any function marked with the `@Compose` annotation. You've created and used composable functions, also known as composables, in the previous chapters to achieve tasks within the exercises. Some of the composables used were built-in composables such as `Text` and `Button`, and built-in layout groups of `Row` and `Column`. You then created your own composable functions using these building blocks.

This chapter provides a more in-depth look at basic composable functions and layout groups in Jetpack Compose. It demonstrates how to use them to build the UI and respond to state changes by recomposing the UI.

By the end of this chapter, you will be able to use all the main composable functions to create UI elements and will have learned how to lay out these elements using the major composable layout groups.

In this chapter, you will cover the following topics:

- Transitioning from XML layouts to Jetpack Compose
- Essential composable functions
- Jetpack Compose layout groups

To start with, you'll learn about the legacy process of using XML to create layouts and compare its imperative approach to the declarative process of using Jetpack Compose.

Technical requirements

The complete code for all the exercises and the activity in this chapter is available on GitHub at <https://packt.link/9GoAL>.

Transitioning from XML layouts to Jetpack Compose

Before Jetpack Compose became the established UI framework, Android apps used XML for the UI. These XML files contained a schema and a root element with embedded XML elements for the content. The XML had to be loaded when the activity started, and by a process called **inflation**, the XML was transformed into a hierarchy of Android views. The views needed to be retrieved before they could be interacted with to set data and perform any operations.

It's important to know the basics of how this works, as many existing apps were built using just this pattern, and most established Google Play apps will have some legacy XML code after they adopt Compose.

Exercise 3.01 – Creating a counter app with legacy views

For the first exercise, you will create a simple counter app that uses views using a new project template:

1. Open Android Studio and select **New Project** on the Android welcome screen. Select **Empty Views Activity** and call it Counter Views.
2. Add the following strings to the `value/strings.xml` file:

```
<string name="zero">0</string>
<string name="plus">+</string>
<string name="minus">-</string>
<string name="counter_text">Counter</string>
```

3. Replace the `TextView` XML elements in the `activity_main.xml` file, which is within the `res/layout` folder, with these new `TextView` elements, which display a label and the counter value:

```
<TextView
    android:id="@+id/counter_text"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/counter_text"
    android:paddingTop="10dp"
```

```
        android:textSize="44sp"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintBottom_toTopOf="@id/counter_value"/>
<TextView
    android:id="@+id/counter_value"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/zero"
    android:textSize="54sp"
    android:textStyle="bold"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintBottom_toBottomOf="parent"/>
```

4. Next, add the Button elements that will increase and decrease the counter value below the TextView elements:

```
<Button
    android:id="@+id/plus"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/plus"
    android:textSize="40sp"
    android:textStyle="bold"
    app:layout_constraintEnd_toStartOf="@id/minus"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@id/counter_value"
    app:layout_constraintBottom_toBottomOf="parent"/>
<Button
    android:id="@+id/minus"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/minus"
    android:textSize="40sp"
    android:textStyle="bold"
    app:layout_constraintStart_toEndOf="@id/plus"
```

```
app:layout_constraintEnd_toEndOf="parent"  
app:layout_constraintTop_toBottomOf="@id/counter_value"  
app:layout_constraintBottom_toBottomOf="parent"/>
```

5. Run the app on the emulator, and you will see the following screen:

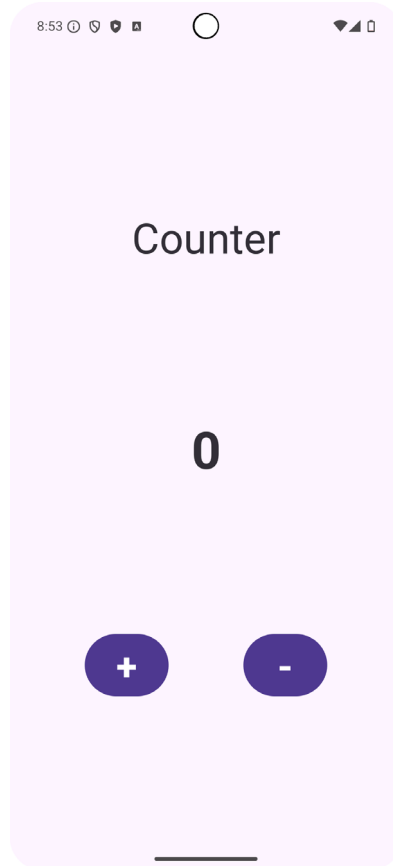


Figure 3.1 – A simple counter app

The layout is fairly simple. There are four views, which are contained within a parent layout group called `ConstraintLayout`. This is the most common XML layout group (or `ViewGroup`, as they are called within the XML view hierarchy). All the embedded views have an `id` identifier with which they can be retrieved in the activity, and the embedded views use the constraint values of top, bottom, start, and end to constrain themselves against their parent or the other views with their `id` identifier value. Guidelines and barriers can also be added to position the views more precisely. Complex layouts can be achieved with these constraints, but the XML required then becomes verbose and less readable.

6. Return to MainActivity and add this code to the bottom of onCreate():

```
var counter = 0
val mainView = findViewById<ConstraintLayout>(R.id.main)
val counterValue = mainView.findViewById<TextView>(
    R.id.counter_value
)
val plusButton = mainView.findViewById<Button>(
    R.id.plus
)
val minusButton = mainView.findViewById<Button>(
    R.id.minus
)
plusButton.setOnClickListener {
    counter++
    counterValue.text = counter.toString()
}
minusButton.setOnClickListener {
    if (counter > 0) {
        counter--
        counterValue.text = counter.toString()
    }
}
```

A counter variable is created, which, as it has a `var` type, is mutable and can be updated. All the required views are then retrieved from the view hierarchy with `findViewById`, and then an `OnClickListener` callback is added to each `Button` view with the `setOnClickListener` function to be invoked when the `Button` object is pressed. This updates the counter integer and then displays the result in the `counterValue` view.

In this simple example, the approach works well to create basic functionality. There are some downsides, however. The UI is still partitioned between the source and the XML rather than building the UI in code. It can also require switching between code and the XML editor, which can slow down development. The overriding disadvantage in comparison with Compose is the imperative nature of building the UI. Every view that requires updating must be retrieved and then set. Each change to the UI requires step-by-step instructions on how to handle state changes. It's a manual approach that can lead to lots of boilerplate code and embedded logic in views, which can be error-prone.

Exercise 3.02 – Creating an Android app with Compose

Let's examine creating the same example app implemented in Compose:

1. Open Android Studio and select **New Project** on the Android welcome screen. Select **Empty Activity** and call it Counter Compose.
2. Add the same strings required to the value/strings.xml file:

```
<string name="zero">0</string>
<string name="plus">+</string>
<string name="minus">-</string>
<string name="counter_text">Counter</string>
```

3. Create a composable called MainScreen below the MainActivity class:

```
@Composable
fun MainScreen(modifier: Modifier) {
    var counter by remember { mutableStateOf(0) }
    Column(
        verticalArrangement = Arrangement.SpaceEvenly,
        horizontalAlignment = Alignment.CenterHorizontally,
        modifier = modifier.fillMaxHeight()
    ) {
        Text(
            text = stringResource(id = R.string.counter_text),
            fontSize = 44.sp,
            modifier = Modifier.align(Alignment.CenterHorizontally)
        )
        Text(
            text = counter.toString(),
            fontSize = 54.sp,
            fontWeight = FontWeight.Bold,
        )
        Row(
            horizontalArrangement = Arrangement.SpaceEvenly,
            modifier = Modifier.fillMaxWidth()
        ) {
            Button(
                onClick = { counter++ }
            ) {
```

```

        Text(
            text = stringResource(id = R.string.plus),
            fontSize = 44.sp,
            fontWeight = FontWeight.Bold,
            modifier = Modifier.padding(horizontal = 8.dp),
        )
    }
    Button(
        onClick = {
            if (counter > 0) {
                counter--
            }
        }
    ) {
        Text(
            text = stringResource(id = R.string.minus),
            fontSize = 44.sp,
            modifier = Modifier.padding(horizontal = 8.dp),
            fontWeight = FontWeight.Bold,
        )
    }
}
}
}

```

It is advisable to create separate composables when building your UI, where possible, starting with the smallest composable functions. This is called the **bottom-up approach**. Creating small composables encourages reusability, and as it minimizes the state within them (as the state can be passed down as arguments), it makes the code more robust. Creating smaller composables also provides explanatory usage through the naming, rather like general refactoring to separate large trunks of code into smaller functions.

In a similar fashion to the `Counter Views` app, a counter is created to hold the state. What is different in the `Compose` example is that it is not set as an integer; it is set as a state holder with `var counter by remember { mutableStateOf(0) }`. As it uses the `remember` function, the value will be retained when the UI is recomposed. In the `Counter`

Views example, after the counter value was incremented or decremented, you needed to explicitly set the value with the following code, which is no longer required:

```
counterValue.text = counter.toString()
```

Another advantage is that the structure and behavior of the UI are clear. You do not have to go back and forth between the code and an XML layout file to understand it. It consists of two Text composables inside a Row composable, which is inside a Column composable. The counter is incremented and decremented by clicking on the plus (counter++) and minus (counter--) buttons. These actions trigger a recomposition as the counter read-only Text value depends on the state.

4. To set the content of the activity to display MainScreen, add a call to the composable within onCreate():

```
Scaffold(  
    modifier = Modifier.fillMaxSize()  
) { innerPadding ->  
    MainScreen(  
        Modifier.padding(innerPadding)  
    )  
}
```

Passing innerPadding down into MainScreen ensures that the composable uses the maximum amount of space available, taking system UI elements into consideration, such as the status bar. Scaffold is a higher-level composable that enables you to position content such as TopAppBar and bottom navigation to provide a structured layout for your app. You will learn about this in *Chapter 4, Building App Navigation*.

5. As a final step, add a composable preview to the file by adding a composable function with the @Preview annotation and adding the MainScreen composable within it:

```
@Preview  
@Composable  
fun MainScreenPreview() {  
    MainScreen(modifier = Modifier.padding(20.dp))  
}
```

The @Preview annotated functions allow viewing the composable in the editor with live changes once the app has been built.

The preceding example is declarative because it describes what the UI should look like depending on the state. The approach relies on reacting to changes in the state, rather than the imperative approach, which requires adding steps to update the state.

In the next section, you'll explore the variety of built-in composables that are available to create rich and engaging UIs.

Essential composable functions

Android's UI design toolkit uses a combination of bare-bones foundational composables and those that incorporate Material Design. There is a wealth of composable functions to create all kinds of Material UI elements, referred to in the Material Design language as **components**. In this section, we will concentrate on the fundamental building blocks of Compose that are crucial for display, handling user interaction, and collecting user input. To see the full list of Material components, go to <https://packt.link/h0mk0>.

Full details of the Google Material Design 3 system can be found here: <https://m3.material.io/>.

If you ever need to create a composable from the lowest level without Material theming, the composables to do so can be found in this package: <https://packt.link/kU10z>.

Text

You've already used the Text composable in many of the examples. At its simplest, you only need to set the text parameter:

```
Text(text = "Hello world!")
```

It then displays the text in the default style:

Hello world!

Figure 3.2 – A basic Text composable

The Text composable does, however, come with a lot of other functionality. If you select the source of the Text composable (by positioning the cursor in the function name and then pressing *command-click* on a Mac or *Ctrl-click* on Windows or Linux), you will see all its properties:

```
// This code is part of the Android SDK and is licensed under the Apache  
// License, Version 2.0.  
// Copyright (C) 2024 Google Inc.  
@Composable
```

```
fun Text(  
    text: String,  
    modifier: Modifier = Modifier,  
    color: Color = Color.Unspecified,  
    fontSize: TextUnit = TextUnit.Unspecified,  
    fontStyle: FontStyle? = null,  
    fontWeight: FontWeight? = null,  
    fontFamily: FontFamily? = null,  
    letterSpacing: TextUnit = TextUnit.Unspecified,  
    textDecoration: TextDecoration? = null,  
    textAlign: TextAlign? = null,  
    lineHeight: TextUnit = TextUnit.Unspecified,  
    overflow: TextOverflow = TextOverflow.Clip,  
    softWrap: Boolean = true,  
    maxLines: Int = Int.MAX_VALUE,  
    minLines: Int = 1,  
    onTextLayout: ((TextLayoutResult) -> Unit)? = null,  
    style: TextStyle = LocalTextStyle.current  
)
```

Let's look at the most commonly used of these parameters:

- **text:** The text to be displayed.
- **modifier:** Used to customize the layout and behavior of composables. For the layout, it allows setting padding, size, alignment, background, border, and other display options. For behavior, it can be used to make a composable clickable and set other interactions, such as making an element draggable or focusable.
- **color:** Enables setting the color of the text.
- **fontSize:** Sets the font size in scalable pixels.
- **fontStyle:** Sets the font style to either Normal or Italic.
- **fontWeight:** Sets the font weight or thickness, which is typically used to set text to Bold, but the values it can be set to range from Thin for the lightest to Black for the heaviest.
- **textAlign:** Sets how the text should be aligned horizontally.
- **overflow:** Specifies how the app handles overflowing text, using either Clip or Ellipsis.
- **style:** Enables setting a style that can set multiple properties (and be reused for other composables) of Text rather than setting them individually.

(You can look at the other parameters in more detail in the Kotlin documentation, Kdoc, above the function name.) The corresponding low-level Text composable without Material styling has fewer parameters and is called `BasicText`.

Button

A Button composable triggers an action when the user interacts with it. When you use XML for the UI, the button is one XML element:

```
<Button
    android:id="@+id/buttonSubmit"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Save" />
```

You then need to add a `clickListener` object on the button itself to create the interaction:

```
val button: Button = findViewById(R.id.buttonSubmit)
button.setOnClickListener {
    // Button click handled here
}
```

You can see the imperative nature of using Android views to build the UI. You must manually update the UI. In Jetpack Compose, the Button composable is clickable by simply adding the `onClick` argument and wrapping the content:

```
Button(onClick = { /* Handle click */ }) {
    Text("Save")
}
```

The following figure shows the button that is displayed as a result of the preceding code:

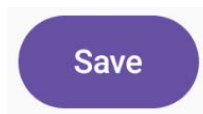


Figure 3.3 – Material styled button

The Button composable is the base type, and there are other buttons that use Button as their base and add outlines, elevation, and fills. See the documentation here for details of all these: <https://packt.link/cgeN4>.

Icon

Icon displays an icon, which is a small image used to give contextual meaning to UI elements and optionally add behavior.

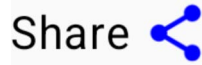


Figure 3.4 – A blue-tinted Share composable icon

The icon in the preceding figure was created with the following code:

```
Row {  
    Text(  
        text = "Share",  
        Modifier.padding(end = 4.dp)  
    )  
    Icon(  
        modifier = Modifier.clickable { /* Handle click */ },  
        imageVector = Icons.Default.Share,  
        tint = Color.Blue,  
        contentDescription = "Share"  
    )  
}
```

In this example, Text is displayed with a Share icon, which sets the `imageVector` parameter to use a built-in vector asset. The `contentDescription` parameter is used for accessibility to provide a description of the icon for visually impaired users. A blue tint is applied with the `tint` parameter, and behavior is added using a `Modifier` parameter to make the icon clickable.

Image

Here, the Text composable is displayed alongside an image using a `painter` argument to set an image resource. It is more general-purpose than Icon as it can load both bitmap images, such as PNG, JPEG, and WebP, and vector assets. It can also apply different scaling using the `contentScale` parameter:

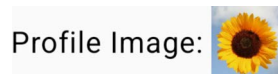


Figure 3.5 – An Image composable

This display is achieved using an Image composable:

```
Row(verticalAlignment = Alignment.CenterVertically) {  
    Text(  
        text = "Profile Image:",  
        Modifier.padding(end = 4.dp)  
    )  
    Image(  
        modifier = Modifier.height(20.dp),  
        painter = painterResource(id = R.drawable.sunflower),  
        contentDescription = "Sunflower",  
        contentScale = ContentScale.Inside  
    )  
}
```

The image is constrained within the bounds imposed by the height to constrain itself within the available space using `ContentScale.Inside`.

Text input fields

These elements correspond to the `EditText` view in XML. They form the core of applications to enter editable text. The base element of these composables is `BasicTextField`. The `TextField` composable is a more fully featured composable that uses Material Design styling, while `OutlinedTextField`, as the name implies, provides the border and Material Design styling. `BasicTextField` is a minimal text input element that allows complete customization. The use of these elements with initial text is shown here:

```
BasicTextField(value = "Enter text:", onValueChange = {  
    // Update Text  
})  
TextField(value = "Enter text:", onValueChange = {  
    // Update Text  
})  
OutlinedTextField(value = "Enter text:", onValueChange = {  
    // Update Text  
})
```

The following is a default `BasicTextField` display with center-aligned text:

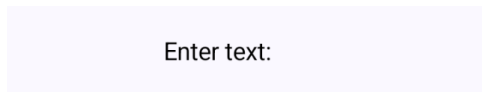


Figure 3.6 – `BasicTextField` with placeholder text

The same content is displayed here with `TextField`:



Figure 3.7 – `TextField` with placeholder text

The same content is displayed here with `OutlinedTextField`:



Figure 3.8 – `OutlinedTextField` with placeholder text

You can see that there is no styling at all on `BasicTextField`. The background color displayed is the app's theme background. `TextField` adds default Material styling, and `OutlinedTextField` adds a border.

Checkbox

Checkbox is used as a binary option, typically where a default is assumed, and allows opting in or out of a particular choice. It is often displayed in lists and forms asking for cookies consent or agreeing to terms and conditions:

```
Checkbox(  
    checked = false, /* or true */  
    onCheckedChange = { /* action when changed */ }  
)
```

The following is an example of a default Checkbox display:

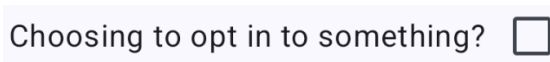


Figure 3.9 – Checkbox in the unchecked state

`Switch` is similar in scope.

Switch

Switch is similar to Checkbox in that it is used for a binary choice, but it is typically used for feature-driven options that can be switched on or turned off, affecting the entire app state or behavior:

```
Switch(  
    checked = false, /* or true */  
    onCheckedChange = { /* action when changed */ }  
)
```

The following is an example of a Switch display:

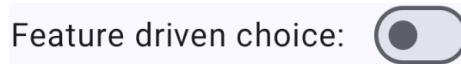


Figure 3.10 – Switch in the off state

Non-binary selections can be achieved with Slider.

Slider

Slider is used to set or respond to a scaling set of values or the percentage of a state on a linear scale. It's typically used for features such as the brightness of a screen setting, the volume of a device, or showing the progress of a song or film. It has a steps parameter that enables restricting the values to a defined number of values.

The following is the code to display a slider set to zero with two steps, excluding the start and end:

```
Slider(  
    value = 0f, /*Set to zero */  
    onValueChange = { /* action when changed */ },  
    steps = 2  
)
```

This produces the following display with the Slider display set to 0:



Figure 3.11 – Slider set to 0 or the smallest scale value

RadioButton

`RadioButton` is used to set a single value from a fixed number of predefined choices. Examples are multiple choice with a set number of answers, a favorite color from a list, and a payment method at the end of an online purchase journey:

```
RadioButton(  
    selected = true, /* or false */  
    onClick = { /* action when clicked */ },  
)
```

The following figure displays three `RadioButton` elements with one selected:

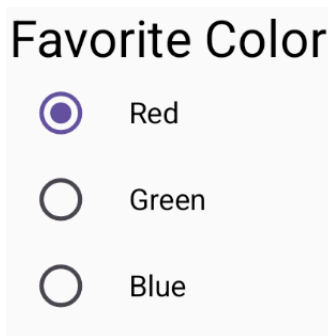


Figure 3.12 – Radio button options with labels

The selected state of each button needs to be manually handled so that when one is selected, the others are deselected.

Progress indicators

A common pattern to inform the user that a page is loading instead of a blank display is to use a progress indicator. The two built-in progress indicators that can be customized by color and stroke size are spinning and linear progress bars:

```
// Spinner Progress Bar  
CircularProgressIndicator()  
  
// Linear Progress Bar  
LinearProgressIndicator(  
    modifier = Modifier.fillMaxWidth()  
)
```

The image at the top is the circular progress indicator. The image at the bottom is the linear progress indicator:

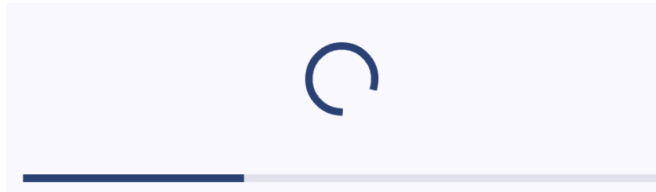


Figure 3.13 – Progress bars

Next, let's understand AlertDialog.

AlertDialog

Dialogs are elevated UI elements that take complete focus of the screen. They are often used to inform the user of a condition, such as an error message. AlertDialog typically displays two options of confirming or dismissing the choice described in the title and message:

```
AlertDialog(  
    onDismissRequest = {  
        /* Handle click outside the dialog window */  
    },  
    title = { Text("Dialog Title") },  
    text = { Text("Dialog Message") },  
    confirmButton = {  
        Button(onClick = {/* Action on confirm */}) {  
            Text(text = "OK")  
        }  
    },  
    dismissButton = {  
        Button(onClick = {/* Action on dismiss */}) {  
            Text(text = "Cancel")  
        }  
    }  
)
```

This code results in the following display with the **Cancel** and **OK** buttons:

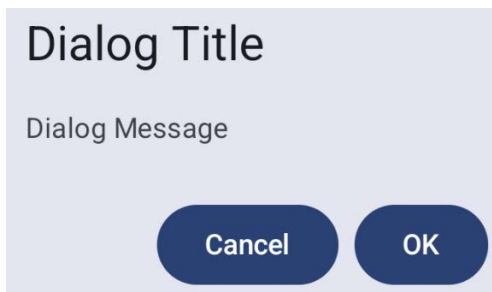


Figure 3.14 – AlertDialog with title, message, and buttons

The `dismissButton` callback is invoked when the **Cancel** button is pressed, and the `confirmButton` callback is invoked when the **OK** button is pressed. Clicking anywhere outside the dialog or the *back* button dismisses it, and the `onDismissRequest` callback is invoked. All of these actions remove the dialog from appearing on the screen.

Exercise 3.03 – Creating a Settings screen

Most apps have a **Settings** screen where app-wide settings can be configured. The action of updating the settings within this screen requires all the components we have just examined:

1. Open Android Studio and select **New Project** on the Android welcome screen. Select **Empty Activity** and call it **Settings**.
2. Create a `SettingsContainer` composable with a column to display the page and a preview composable so that, as you incrementally build the page, you can see it taking shape:

```
@Composable
fun SettingsContainer(modifier: Modifier = Modifier) {
    Column(
        verticalArrangement = Arrangement.Center,
        horizontalAlignment =
            Alignment.CenterHorizontally,
        modifier = modifier
    ) {
    }
}

@Preview
@Composable
fun SettingsContainerPreview() {
```

```
SettingsContainer()
}
```

3. Add SettingsContainer as the content of Scaffold:

```
Scaffold(
    modifier = Modifier.fillMaxSize()
) { innerPadding ->
    SettingsContainer(
        modifier = Modifier.padding(innerPadding)
    )
}
```

4. Add the following strings to the strings.xml file:

```
<string name="settings_icon_description">
    Settings
</string>
<string name="settings_consent">
    Accept non-essential cookies?
</string>
<string name="settings_mobile_data">
    Download using mobile data?
</string>
<string name="settings_text_size">Text size:</string>
<string name="settings_name">Name:</string>
<string name="sign_out">Sign out</string>
<string name="alert_title">Sign out</string>
<string name="alert_message">Are you sure?</string>
<string name="ok">OK</string>
<string name="cancel">Cancel</string>
<string name="payment_method">
    Preferred payment method
</string>
<string name="settings_profile_image">Profile Image</string>
```

5. Add a TextStyle at the bottom of Theme.kt:

```
val HeaderTextStyle = TextStyle(
    fontSize = 28.sp,
```

```
fontWeight = FontWeight.ExtraBold
)
```

We examined the parameters of the Text composable before and saw that style is one of them. We'll use the style parameter to apply HeaderTextStyle to the header when we create it.

6. Add a SettingsHeader composable to display the page title with a settings icon:

```
@Composable
fun SettingsHeader() {
    Row(
        modifier = Modifier
            .fillMaxWidth()
            .padding(vertical = 14.dp),
        verticalAlignment = Alignment.CenterVertically,
        horizontalArrangement = Arrangement.Center
    ) {
        Text(
            text = stringResource(
                id = R.string.app_name),
            style = HeaderTextStyle,
            modifier = Modifier.padding(end = 10.dp)
        )
        Icon(
            imageVector = Icons.Default.Settings,
            contentDescription = stringResource(
                id = R.string.settings_icon_description
            ),
        )
    }
}
```

Each of the items on the **Settings** screen will consist of a Row composable with two composables inside it. In the header, we are setting the Row composable to be full width using fillMaxWidth() on the Modifier parameter, and adding some vertical space above and below the Row composable with padding(vertical = 14.dp). The contents of the Row composable are centered vertically and arranged in the center by setting the verticalAlignment = Alignment.CenterVertically and horizontalArrangement =

`Arrangement.Center` parameters. The style we added in *step 5*, `HeaderTextStyle`, is now used on the `Text` composable to display an appropriate title. Finally, a `Settings` cog `Icon` composable is added to the `Row` composable.

7. Add the composable you have just created to `SettingsContainer`, and do it for all of the following composables we create:

```
Column(
    verticalArrangement = Arrangement.Center,
    horizontalAlignment = Alignment.CenterHorizontally,
    modifier = modifier
) {
    // Header
    SettingsHeader()
}
```

8. Add a composable to display a profile image. You can download the profile image from <https://packt.link/fxKe0> or use your own:

```
@Composable
fun SettingsImage() {
    Row(
        modifier = Modifier
            .fillMaxWidth()
            .padding(vertical = 8.dp)
            .padding(start = 16.dp),
        verticalAlignment = Alignment.CenterVertically,
        horizontalArrangement =
            Arrangement.SpaceBetween
    ) {
        Text(
            text = stringResource(
                id = R.string.settings_profile_image),
            fontSize = 18.sp,
        )
        Image(
            modifier = Modifier.padding(
                end = 10.dp).height(34.dp)
            .clickable {
```

```

        /* Handle changing the profile image */
    },
    painter = painterResource(
        id = R.drawable.sunflower),
    contentDescription = stringResource(
        id = R.string.settings_profile_image),

)
}
}

```

The appropriate padding has been added to offset the Row composable from the start (left) edge with `padding(start = 16.dp)`, and the composables within the Row composable are arranged with `Arrangement.SpaceBetween`, so any available space is spaced between the composables. The profile image is added and made clickable by setting `.clickable` on the `Modifier` parameter.

9. Next, add a `Checkbox` composable to enable the user to opt in or out of accepting non-essential cookies:

```

@Composable
fun SettingsCheckbox() {
    var isChecked by remember { mutableStateOf(false) }
    Row(
        modifier = Modifier
            .fillMaxWidth()
            .padding(vertical = 8.dp)
            .padding(start = 16.dp),
        verticalAlignment = Alignment.CenterVertically,
        horizontalArrangement =
            Arrangement.SpaceBetween
    ) {
        Text(
            text = stringResource(
                id = R.string.settings_consent),
            fontSize = 18.sp,
        )
        Checkbox(
            checked = isChecked,

```

```

        onCheckedChange = { isChecked = it },
    )
}
}

```

10. As this is the first composable that contains a value that is subject to change, we have to create a state to hold the checked value of Checkbox. It is initially set to false with the following:

```

var isChecked by remember { mutableStateOf(false) }

```

11. When the checkbox is changed, then the onCheckedChange function block is run:

```

onCheckedChange = { isChecked = it },

```

Here, it is the updated value of the checkbox. As the isChecked state is being used to set the state of the checkbox in checked = isChecked, when it is updated, the checkbox has to be recomposed, displaying the updated state, either checked or not.

12. Add a Row composable for the user to switch to download over mobile data, with a Switch composable to represent the yes and no states:

```

@Composable
fun SettingsSwitch() {
    var isChecked by remember { mutableStateOf(false) }
    Row(
        modifier = Modifier
            .fillMaxWidth()
            .padding(vertical = 8.dp)
            .padding(start = 16.dp),
        verticalAlignment = Alignment.CenterVertically,
        horizontalArrangement =
            Arrangement.SpaceBetween
    ) {
        Text(
            text = stringResource(
                id = R.string.settings_mobile_data),
            fontSize = 18.sp,
        )
        Switch(
            modifier = Modifier.padding(end = 10.dp),

```



```

        checked = isChecked,
        onCheckedChange = { isChecked = it },
    )
}
}

```

The behavior is almost the same as in the previous Checkbox example. The only difference is that the composable element has now been changed to a Switch composable.

13. Add a Slider composable to configure the text size:

```

@Composable
fun SettingsSlider() {
    var sliderValue by remember { mutableStateOf(0f) }

    Row(
        modifier = Modifier
            .fillMaxWidth()
            .padding(vertical = 8.dp)
            .padding(start = 16.dp),
        verticalAlignment = Alignment.CenterVertically,
        horizontalArrangement =
            Arrangement.SpaceBetween
    ) {
        Text(
            modifier = Modifier.padding(end = 16.dp),
            text = stringResource(
                id = R.string.settings_text_size),
            fontSize = 18.sp,
        )
        Slider(
            value = sliderValue,
            onValueChange = { sliderValue = it },
            steps = 2
        )
    }
}

```

Again, the manner of updating `Slider` is very similar to `Checkbox` and `Switch`. In this case, however, the state of the slider is available in `sliderValue` as a float (`0f`), which is a floating-point number (which can contain a decimal place) and is updated in `onValueChange {}`, which reflects that the slider is a scale and doesn't have a binary state of *on* or *off*. We've specified it in this example to have two steps, which, when added to the start and end settings, means there are four text sizes available for the user to choose.

14. Add `RadioButton` composables to select the preferred payment method:

```
@Composable fun SettingsRadioButtons() {
    var selectedPaymentMethod by remember { mutableStateOf("PayPal")
}

Column(modifier = Modifier
    .fillMaxWidth()
    .padding(16.dp)) {
    Text(text = stringResource(id = R.string.payment_method),
        modifier = Modifier.padding(bottom = 8.dp))
    listOf("PayPal", "Credit Card", "Bank Transfer").forEach {
        paymentMethod -> Row(
            verticalAlignment = Alignment.CenterVertically,
            modifier = Modifier.padding(vertical = 4.dp)
        ) {
            RadioButton(
                selected = (selectedPaymentMethod ==
                    paymentMethod),
                onClick = { selectedPaymentMethod =
                    paymentMethod },
                colors = RadioButtonDefaults.colors()
            )
            Text(text = paymentMethod, modifier =
                Modifier.padding(start = 8.dp))
        }
    }
}
```

Three items are created in a list with `listOf`. They are then looped through to create a `RadioButton` composable for each of the payment methods. `PayPal` is set to the default with `var selectedPaymentMethod` by `remember { mutableStateOf("PayPal") }`, and then the `RadioButton` composables are evaluated to check whether they are set as selected with `selected = (selectedPaymentMethod == paymentMethod)`. Then, `onClick` is added to select them if they are clicked with `selectedPaymentMethod = paymentMethod`.

15. Finally, add an `AlertDialog` composable to sign the user out with a confirmation to verify that it is their intention to sign out:

```
@Composable fun SettingsAlertDialog() {
    var showDialog by remember { mutableStateOf(false) }

    Button(onClick = { showDialog = true }) {
        Text(text = stringResource(id = R.string.sign_out))
    }

    if (showDialog) {
        AlertDialog(
            onDismissRequest = { showDialog = false },
            title = { Text(text = stringResource(id =
                R.string.alert_title)) },
            text = { Text(text = stringResource(id =
                R.string.alert_message)) },
            confirmButton = {
                Button(onClick = { showDialog = false }) {
                    Text(text = stringResource(id = R.string.ok))
                }
            },
            dismissButton = {
                Button(onClick = { showDialog = false }) {
                    Text(text = stringResource(id =
                        R.string.cancel))
                }
            }
        )
    }
}
```

Whether or not to show the dialog is governed by `showDialog`, which is set to `false` by `mutableStateOf(false)`. When the button is clicked, `showDialog` gets flipped to `true`, and `AlertDialog` displays with a title and message. `confirmButton` and `dismissButton` enable actions to be taken when either one is pressed, and also close the `AlertDialog` composable.

When you have added the preceding composables, the screen should appear as in the following figure:

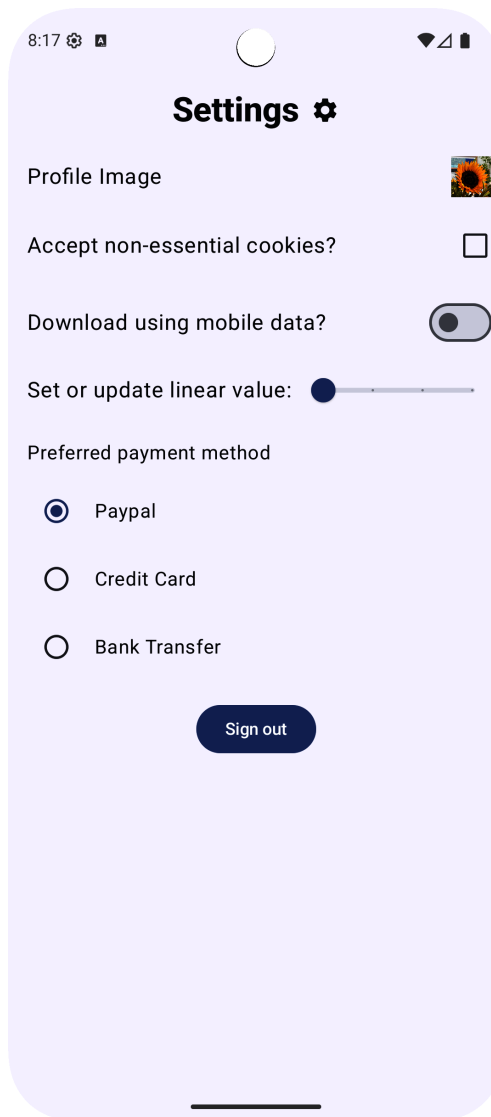


Figure 3.15 – Settings screen

This exercise has shown how easy it is to display Compose UI elements, style them, and add behavior. Together with composable text input fields, these are the principal building blocks of a Jetpack Compose UI. They are the leaf nodes in which behavior is captured and actions are propagated from.

Next, we'll explore how these building block composables can be positioned in composable layout groups.

Jetpack Compose layout groups

You've already used the key composable layout groups of `Row` and `Column` from the first chapter onward. With these composables, you can create many complex layouts. There are, however, several other layout groups that are at the core of building attractive UIs. We will first cover the `Box` composable layout, which enables the positioning of items in a stack with the alignment of its child composables. Then, we will look at the `Surface` composable layout group, which is a Material Design-based container layout group that enables you to build on your app's theme to customize and enhance the look of your UI by adding background and elevation. Following on from this, we'll look at the `Card` composable layout group, which is a specialized surface customized with Material Design for card-style layouts. Then, we will take a more in-depth look at `Column` and `Row` composables.

Box

The `Box` layout group positions composables within it using a stack that is aligned to the parent's edges. So, the last composable added is displayed on top.

Adding three `Text` composables of "RED", "GREEN", and "BLUE" contained within a `Box` composable displays them one on top of the other:

```
Box(modifier = Modifier.fillMaxSize()){
    Text(text = "RED",
        fontSize = 20.sp,
    )
    Text(text = "GREEN",
        fontSize = 20.sp,
    )
    Text(text = "BLUE",
        fontSize = 20.sp,
    )
}
```

The following is a Box composable display showing each Text composable displayed on top of each other:

GREEN

Figure 3.16 – Box displaying stacked composables



Quick tip: Need to see a high-resolution version of this image? Open this book in the next-gen Packt Reader or view it in the PDF/ePub copy.

The next-gen Packt Reader and a **free PDF/ePub copy** of this book are included with your purchase. Scan the QR code OR visit packtpub.com/unlock, then use the search bar to find this book by name. Double-check the edition shown to make sure you get the right one.



By default, a Box composable positions its elements in the top-left corner (`contentAlignment: Alignment = Alignment.TopStart`), but this can be overridden by setting the `contentAlignment` parameter on the Box composable itself:

```
Box(contentAlignment = Alignment.Center)
```

However, the contents would still be stacked on top of each other. The alternative is to set the alignment of the composables within the Box composable:

```
@Composable fun BoxDisplay() {
    Box(Modifier.fillMaxSize(), contentAlignment = Alignment.Center) {
        Text(
            text = "RED",
            fontSize = 20.sp,
            fontWeight = FontWeight.Bold,
            modifier = Modifier.align(Alignment.TopStart)
        )
        Text(
            text = "GREEN",
```

```
        fontSize = 20.sp,  
        fontWeight = FontWeight.Bold,  
    )  
    Text(  
        text = "BLUE",  
        fontSize = 20.sp,  
        fontWeight = FontWeight.Bold,  
        modifier = Modifier.align(Alignment.BottomEnd)  
    )  
}  
}
```

This code produces the following display:

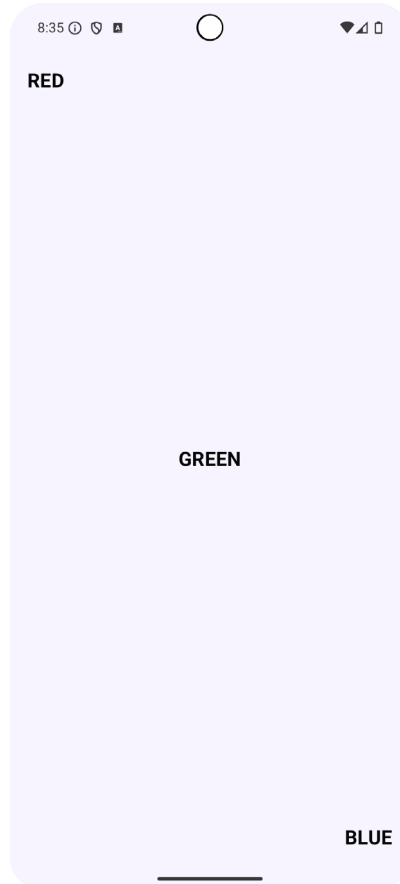


Figure 3.17 – Box displaying different alignment values

In this example, modifiers are used to set the Box composable itself and the Box composable contents. This enables behavior and appearance to be set on the composable group and composables within it.

Boxes are very useful when you need simple positioning of composables that use one of the nine alignment positions of TopStart, TopCenter, TopEnd, CenterStart, Center, CenterEnd, BottomStart, BottomCenter, and BottomEnd.

The next composable layout group to explore is Surface.

Surface

Surface is a core layout composable that lets you provide a base for an app's content. You can add style elements such as a background, shape, elevation, and borders, wrapping a single embedded composable:

```
@Composable fun SurfaceExample(){
    Surface(
        modifier = Modifier
            .padding(40.dp)
            .width(300.dp),
        color = Color.LightGray,
        contentColor = Color.Yellow,
        shape = RoundedCornerShape(22.dp),
        tonalElevation = 8.dp,
        shadowElevation = 8.dp,
        border = BorderStroke(2.dp, SolidColor(Color.Blue))
    ) {
        Text(
            text = "Hello, Surface!",
            textAlign = TextAlign.Center,
            modifier = Modifier
                .padding(16.dp).fillMaxWidth(),
            fontSize = 22.sp
        )
    }
}
```


The preceding Surface composable has a background color of `Color.Gray` and the default styling of the contents within it will be the `contentColor` `Color` of `Color.Yellow`. The shape is set to `RoundedCornerShape` with a corner size of `22.dp`, and other shapes, such as `Circle`, are available. Specific material theming can be added using `tonalElevation` and `shadowElevation`. There is a range of border options that can use linear and radial gradients. Here, a solid color is applied:

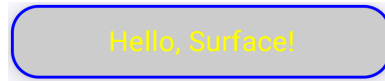


Figure 3.18 – Surface layout group with Text content

Closely related to Surface is the Card composable.

Card

The Card composable is a filled Surface that has predefined styling for padding, elevation, and borders. The following code demonstrates how to create a very similar-looking composable, but setting fewer parameters:

```
Card(
    modifier = Modifier.padding(40.dp)
        .width(300.dp),
    colors = CardColors(
        Color.LightGray,
        Color.Yellow,
        Color.LightGray,
        Color.Yellow
    ),
    border = BorderStroke(2.dp, SolidColor(Color.Blue))
) {
    Text(
        text = "Hello, Card!",
        textAlign = TextAlign.Center,
        modifier = Modifier.padding(16.dp).fillMaxWidth(),
        fontSize = 22.sp
    )
}
```

tonalElevation and shadowElevation are set with defaults, and the colors parameter is set to CardColors, which sets the color and contentColor values, as well as the corresponding colors for when the button is disabled.



Figure 3.19 – Card layout group with Text content

A more structured layout for multiple items can be achieved using the Column and Row composables.

Column

You’ve used this layout composable from the very first chapter. Its function is to lay out composables in a column, controlling the start-to-end alignment of the content with the `horizontalAlignment` parameter and how the content is laid out from top to bottom with the `verticalArrangement` parameter. The `horizontalAlignment` options are `Start`, `End`, or `CenterHorizontally` to position composables along the *x* axis. The arrangement of the *y* axis allows for placing content in different ways. Content can be spaced apart by a fixed amount using the `spacedBy` parameter, or if either `Modifier.fillMaxSize` or `Modifier.fillMaxHeight` is used, then content can be spaced using the following values: `Top`, `Bottom`, `Center`, `SpaceEvenly`, `SpaceAround`, or `SpaceBetween`.

The following code uses three `Text` composables to illustrate how the content is laid out with a `verticalArrangement` parameter, which is `SpaceEvenly`:

```
Column(  
    modifier = Modifier.fillMaxSize(),  
    horizontalAlignment = Alignment.CenterHorizontally,  
    verticalArrangement = Arrangement.SpaceEvenly  
) {  
    Text(  
        text = "RED",  
        color = Color.White,  
        fontSize = 24.sp,  
        modifier = Modifier  
            .background(Color.Red)  
            .width(100.dp)  
            .padding(4.dp),
```

```
        textAlign = TextAlign.Center
    )
    Text(
        text = "GREEN",
        color = Color.White,
        fontSize = 24.sp,
        modifier = Modifier
            .background(Color.Green)
            .width(100.dp)
            .padding(4.dp),
        textAlign = TextAlign.Center
    )
    Text(
        text = "BLUE",
        color = Color.White,
        fontSize = 24.sp,
        modifier = Modifier
            .background(Color.Blue)
            .width(100.dp)
            .padding(4.dp),
        textAlign = TextAlign.Center
    )
}
```

The following figure shows how `SpaceEvenly` distributes space evenly between contents, and also before the first and after the last item:

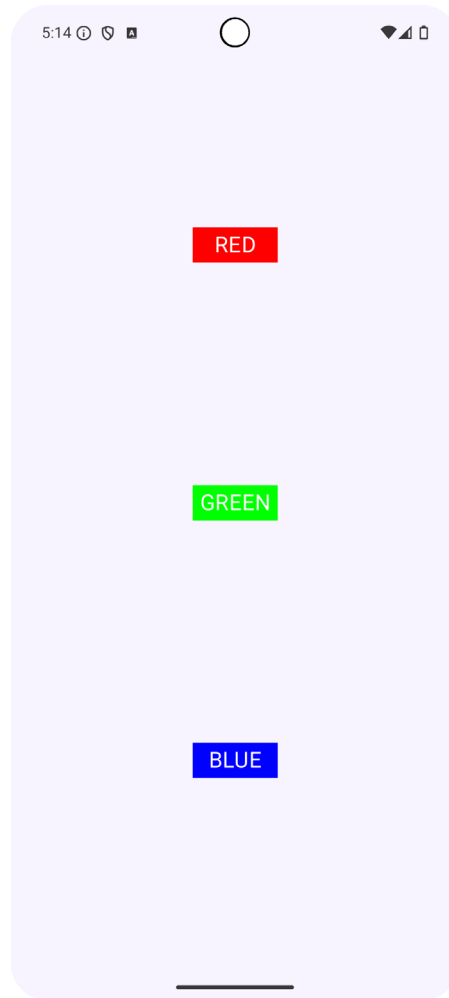


Figure 3.20 – Column contents spaced evenly

This illustration from the official documentation shows how all the options display: <https://packt.link/6RJve>.

The accompanying layout composable to `Column` is `Row`.

Row

Arranging composables along the x axis is what `Row` does. It uses parameters for the arrangement and alignment of content in a similar way to `Column`. The `verticalAlignment` parameter sets the content alignment to one of `Top`, `CenterVertically`, or `Bottom`, and `horizontalArrangement`

sets how the content of the area the row occupies will be distributed from start to end, if `Modifier.fillMaxSize()` or `Modifier.fillMaxWidth` is set. These options are `Start`, `End`, `Center`, `SpaceEvenly`, `SpaceAround`, and `SpaceBetween`.

You can also use `spacedBy` to add space after each row item. The following code shows how the Row content is `SpaceAround`, so space is added around each item, and the space before and after the last item is half the space between the children:

```
Row(
    modifier = Modifier.fillMaxSize(),
    horizontalArrangement = Arrangement.SpaceAround,
    verticalAlignment = Alignment.CenterVertically
) {
    Text(
        text = "RED",
        color = Color.White,
        fontSize = 24.sp,
        modifier = Modifier
            .background(Color.Red)
            .width(100.dp)
            .padding(4.dp),
        textAlign = TextAlign.Center
    )
    Text(
        text = "GREEN",
        color = Color.White,
        fontSize = 24.sp,
        modifier = Modifier
            .background(Color.Green)
            .width(100.dp)
            .padding(4.dp),
        textAlign = TextAlign.Center
    )
    Text(
        text = "BLUE",
        color = Color.White,
        fontSize = 24.sp,
        modifier = Modifier
```

```
        .background(Color.Blue)  
        .width(100.dp)  
        .padding(4.dp),  
        textAlign = TextAlign.Center  
    )  
}
```

This Row composable will be displayed like this:

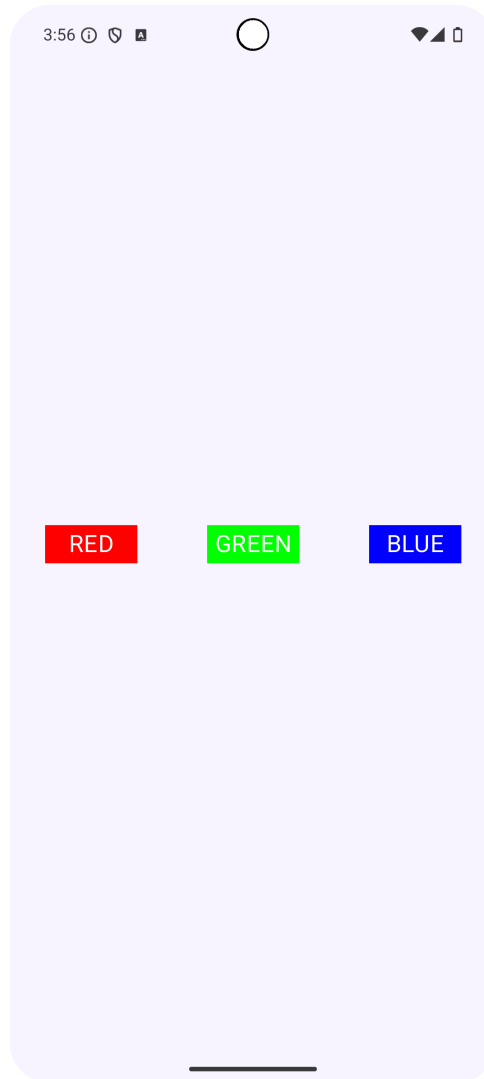


Figure 3.21 – Row contents spaced around

This illustration from the official documentation shows how all the options display: <https://packt.link/jYNX6>.

The core composable layout groups provide the fundamental building blocks that you will use to build the UI. To see these in action, follow the next exercise to create a **Profile** page.

Exercise 3.04 – Creating a Profile page

Apps that depend on interaction between users usually have a **Profile** page. You will create one of these **Profile** pages in Jetpack Compose:

1. Open Android Studio and select **New Project** on the Android welcome screen. Select **Empty Activity** and call it **Profile**.
2. Create a composable called **Profile** with a preview:

```
@Composable
fun Profile(modifier: Modifier) {
}

@Preview(showBackground = true)
@Composable
fun ProfilePreview() {
    Profile(modifier = Modifier.padding(20.dp))
}
```

You add a **Profile** composable and the preview so that as you start building up the screen, you can see how it displays. By default, Android Studio shows a transparent background on previews. As you will create a **Surface** composable to display the card, to distinguish it as a card, you need elevation. This produces a shadow, which is difficult to see on a transparent background.

3. Add a **Surface** composable elevation with rounded corners to create the basic panel that the card contents will be displayed on:

```
Surface(
    modifier = modifier
        .fillMaxWidth()
        .padding(16.dp),
    shape = RoundedCornerShape(16.dp),
    shadowElevation = 8.dp,
```

```
        color = Color.White
    ) {}
```

As there is no content added to `Surface` at this stage, the preview will just display the background. Once content is added, you will see `RoundedCornerShape` take effect. This is specified as `16.dp`, which sets the same size for all corners. `shadowElevation` sets how raised the surface is above the main content, which it does through applying shadow. The background color is specified as `Color.White`.

4. As the content is laid out from top to bottom, we need to create a `Column` composable. We want there to be padding inside and all the contents to be aligned horizontally by default:

```
Column(
    horizontalAlignment = Alignment.CenterHorizontally,
    modifier = Modifier
        .padding(16.dp)
        .fillMaxWidth()
) {}
```

As you have some content, albeit an empty layout group, you can see the elevation:

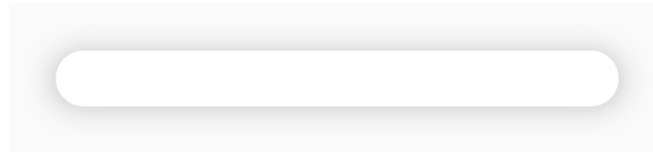


Figure 3.22 – Empty `Column` composable displaying within a `Surface` composable

5. The content will start by showing a profile image and a way to edit it. We want the **Edit** button to be overlaid slightly over the top of the image, so create a `Box` composable with a defined size:

```
Box (modifier = Modifier.size(116.dp)) {
}
```

6. Add a profile image within the box:

```
Image(
    // Replace with your image resource
    painter = painterResource(id = R.drawable.cat),
    contentDescription = "Profile Picture",
    contentScale = ContentScale.Inside,
```



```
        modifier = Modifier  
            .padding(16.dp)  
            .size(100.dp)  
            .clip(RoundedCornerShape(50.dp))  
    )
```

You can add your own image or download one from the repository: <https://packt.link/efhEs>.

For accessibility, add a `contentDescription` parameter and then the `Modifier` parameter with `clip(RoundedCornerShape(50.dp))` to clip the image with a rounded corner shape with a size of `50.dp`. The `contentScale` value is specified as `ContentScale.Inside`, so the image will be scaled down to size if required in the `Image` composable. The image is clickable to allow updating the profile image. The order of the padding is important here. Currently, `Image` has space applied between its border and the content inside. So, the cat image can take up the whole `100.dp` size.

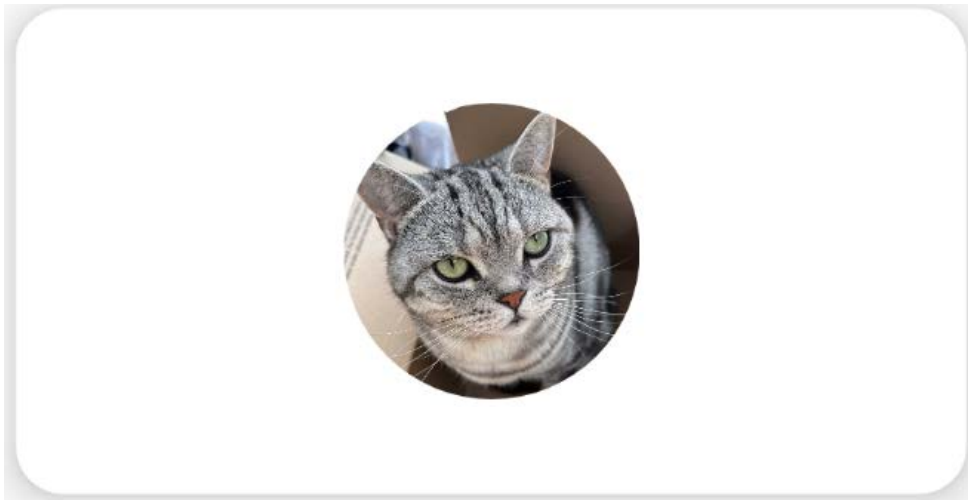


Figure 3.23 – Image padding applied before specifying the height

7. Now, swap these values around:

```
        .size(100.dp)  
        .padding(16.dp)
```

There is only the size value, `100.dp`, minus the padding value of `16.dp`, so `84.dp` in total to display the image.

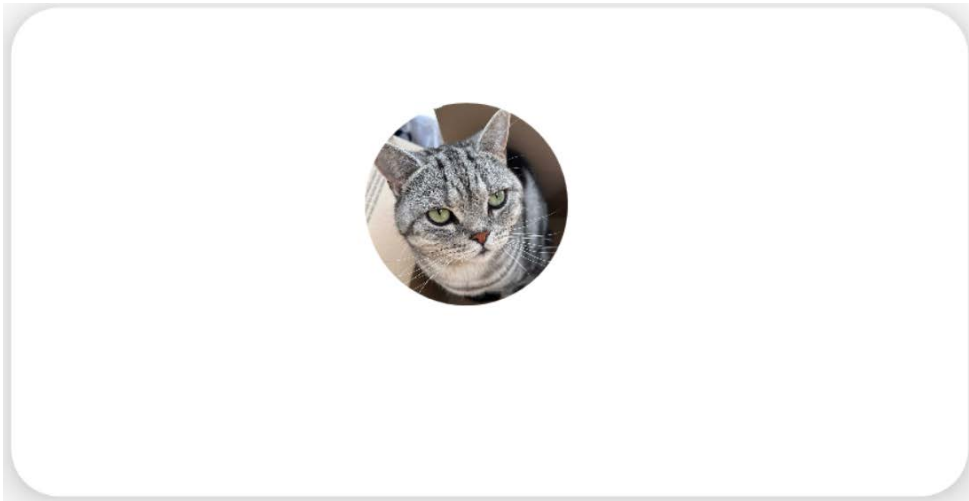


Figure 3.24 – Image padding applied after specifying the size

Thus, the image is displayed smaller.

8. Swap the values back again and add a clickable Card composable with a Text composable below the Image composable:

```
Card(  
    onClick = {/**/},  
    modifier = Modifier.align(BottomEnd),  
    border = BorderStroke(  
        1.dp, SolidColor(Color.Blue))  
) {  
    Text(  
        text = "Edit",  
        modifier = Modifier.padding(horizontal = 4.dp),  
        fontSize = 12.sp)  
    }
```

As the Card composable is aligned using the BottomEnd option of the Box composable that wraps it, the Text composable appears at the bottom of the image aligned to the end:

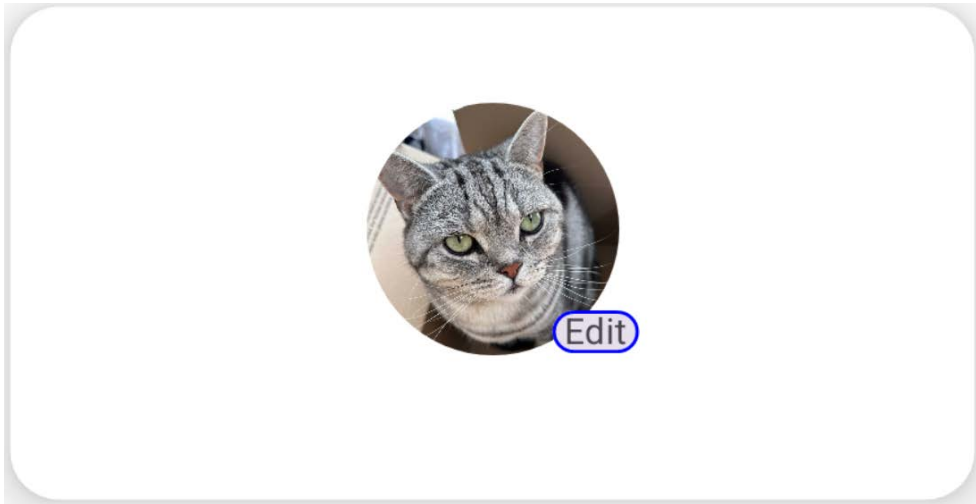


Figure 3.25 – Image padding applied after specifying the height

Next, we'll add some personal details for the profile.

9. Add two Text composables for the name and description (below the Box composable), one above the other, and then a Row composable with two buttons with **calls to action (CTAs)** to follow and message the user:

```
Spacer(modifier = Modifier.height(16.dp))
Text(
    text = "Jane Doe",
    fontSize = 24.sp,
    fontWeight = FontWeight.Bold,
    textAlign = TextAlign.Center,
    modifier = Modifier.padding(top = 16.dp)
)
Text(
    text = "Mobile Developer | Tech Enthusiast",
    fontSize = 16.sp,
    color = Color.Gray,
    modifier = Modifier.padding(vertical = 16.dp)
```

```
)

Row(
    horizontalArrangement = Arrangement.SpaceEvenly,
    modifier = Modifier.fillMaxWidth()
) {
    Button(onClick = { /* TODO: Add follow action */ }) {
        Text(text = "Follow")
    }
    Button(onClick = { /* TODO: Add message action */ }) {
        Text(text = "Message")
    }
}
```

You are using a `Spacer` composable to add some vertical whitespace between the column items. This is often very useful for customizing your display, as you don't then need to add padding to other items to achieve the look you want. `HorizontalDivider` and `VerticalDivider` achieve similar effects with a dividing line. You are individually specifying the `Text` look using parameters instead of styles. You are horizontally spacing the composables with a `horizontalArrangement` value of `SpaceEvenly`, so any available space will be positioned equally with the buttons along the horizontal axis.

10. The last step is to add the `Profile` composable to `setContent` within the `Theme` composable and the `Scaffold` composable:

```
setContent {
    ProfileTheme {
        Scaffold(
            modifier = Modifier.fillMaxSize()
        ) { innerPadding ->
            Profile(
                modifier = Modifier
                    .padding(innerPadding)
            )
        }
    }
}
```

The final display should look like the following:

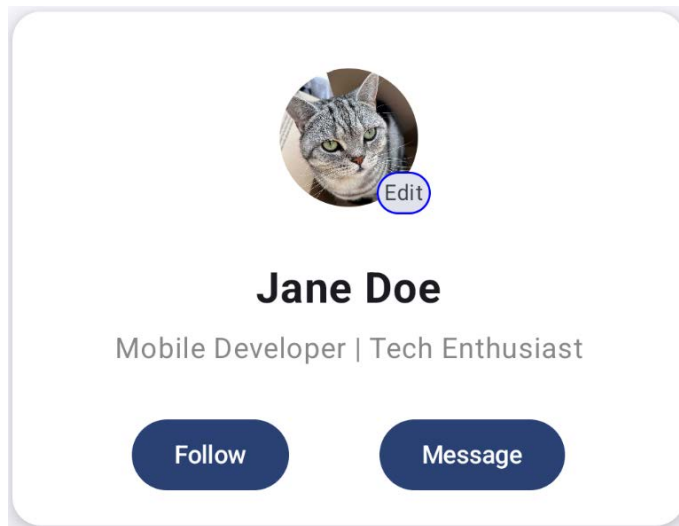


Figure 3.26 – Profile page

This exercise has demonstrated the use of most of the core composable layout groups. There are more layout groups to discover, however. In *Chapter 4, Building App Navigation*, you will use `Scaffold` to lay out app content into more manageable sections and also use navigation layout groups to navigate around your apps. You'll learn how to create lists and grids in *Chapter 6, Building Lists with Jetpack Compose*. For more precise positioning of composables, take a look at the Jetpack Compose version of the legacy XML view-based layout called `ConstraintLayout`. It provides finer positional control of UI elements relative to each other, as well as to guidelines and barriers: <https://packt.link/u8toL>.

Now that you have acquired the knowledge of core composables and layout groups, complete the activity to create a business dashboard that displays key financial metrics.

Activity 3.01 – Creating a business metrics dashboard

The object of this exercise is to use some of the core composables to show four key business metrics in a dashboard display. It should have four tiles displaying a headline metric and a detailed display area below the tiles that displays more detail about the selected metric:

1. Open Android Studio and select **New Project** on the Android welcome screen. Select **Empty Activity** and call it **Dashboard**.
2. Add a `Text` composable with `Business Dashboard` as the header of the page.

3. Create four states for the four headline metrics. These can be strings, integers, or another data type.
4. Create a `MutableState` object with the `remember` function that will hold the value of the selected metric.
5. Create a single dashboard tile composable that will be reused to display a headline company metric. It should take a title text and a click handler for the action when the tile is clicked.
6. Create a composable that takes in the selected metric and evaluates the value before displaying one of the key metrics in detail. This can be a few lines of text.
7. Create the main body of the dashboard, which will display the four dashboard tiles with a key metric headline title and a click handler that updates the selected metric to one of the four states.
8. Verify that each of the four tiles populates the company details for each of the corresponding four states.
9. For an extra task, change the background color of the selected headline title to show which company metric is being displayed.



The solution to this activity can be found at <https://packt.link/PAeqs>.

Summary

This chapter has covered core composables and layout groups in depth. We started with a comparison between the legacy XML creation of a UI compared with composables. We then moved on to studying the foundational composables, before finishing with a study of the major composable layout groups. These are the fundamental building blocks of creating Android UIs. You can build upon these concepts to progress to create increasingly more advanced UIs.

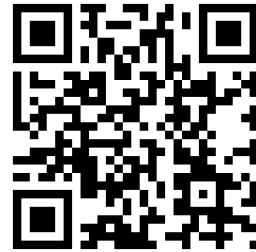
The next chapter will expand on this knowledge by using established UI patterns to build clear and consistent navigation in your apps.

Unlock this book's exclusive benefits now

Scan this QR code or go to packtpub.com/unlock, then search this book by name.

Note: Keep your purchase invoice ready before you start.

UNLOCK NOW



4

Building App Navigation

In *Chapter 3, Developing the UI with Jetpack Compose*, you explored core composables and layout groups to create the **user interface (UI)**. In this chapter, you will build user-friendly app navigation through three primary patterns: the *navigation drawer*, *bottom navigation*, and *tab navigation*. Through guided theory and practice, you will learn how each of these patterns works so that users can easily access your app's content.

You will start by learning how to create the structure of your app with top and bottom bars and body content. Then, you'll learn about the fundamentals of how navigation in Jetpack Compose works, how it is built around routes, and how routes govern navigation within apps. Moving on, you'll implement the navigation drawer, the earliest widely adopted navigational pattern used in Android apps, before exploring bottom navigation and tab navigation.

By the end of this chapter, you will know how to use these three primary navigation patterns and understand how they work to support navigation.

We will cover the following topics in this chapter:

- Creating a screen structure with a Scaffold composable and slots
- Building a navigation graph
- Implementing a navigation drawer
- Adding bottom navigation
- Introducing tabbed navigation

Technical requirements

The complete code for all the exercises and the activity in this chapter is available on GitHub at <https://packt.link/DuNcM>.

Creating a screen structure with a Scaffold composable and slots

So far, the content of all the exercises you have been working on starts with creating an app template with a Scaffold composable. This is a high-level layout group that enables building a skeletal app layout structure to display top and bottom bars and body content, simplifying the process of creating an app's UI. So far, you have used it to provide body content, as the default option is to only display one composable. However, Scaffold contains slots, which are specific placeholders that accept other composables. The main slots Scaffold has are `topBar`, `bottomBar`, and `content`.

The following code block shows how Scaffold can be used with slots to create this structure:

```
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun MyScaffoldApp() {
    Scaffold(
        topBar = {
            CenterAlignedTopAppBar(
                title = { Text("TOP BAR") },
                modifier = Modifier.statusBarsPadding(),
                colors = TopAppBarDefaults.centerAlignedTopAppBarColors(
                    containerColor =
                        MaterialTheme.colorScheme.surfaceContainer
                )
            )
        },
        bottomBar = {
            BottomAppBar(
                modifier = Modifier.fillMaxWidth().height(56.dp)
            ) {
                Box(
                    modifier = Modifier.fillMaxWidth(),
                    contentAlignment = Alignment.Center
                ) {
                    Text(text = "BOTTOM BAR")
                }
            }
        }
    ),
}
```

```
content = { innerPadding ->
    Box(
        modifier = Modifier
            .fillMaxSize()
            .padding(innerPadding),
        contentAlignment = Alignment.Center
    ) {
        Text(
            text = "BODY CONTENT",
            modifier = Modifier.padding(16.dp),
            textAlign = TextAlign.Center
        )
    }
}
```

💡 **Quick tip:** Enhance your coding experience with the **AI Code Explainer** and **Quick Copy** features. Open this book in the next-gen Packt Reader. Click the **Copy** button (1) to quickly copy code into your coding environment, or click the **Explain** button (2) to get the AI assistant to explain a block of code to you.

```
function calculate(a, b) {
    return {sum: a + b};
};
```

Copy

Explain

1

2



🔒 **The next-gen Packt Reader** is included for free with the purchase of this book. Scan the QR code OR go to packtpub.com/unlock, then use the search bar to find this book by name. Double-check the edition shown to make sure you get the right one.



The output can be seen in *Figure 4.1*.

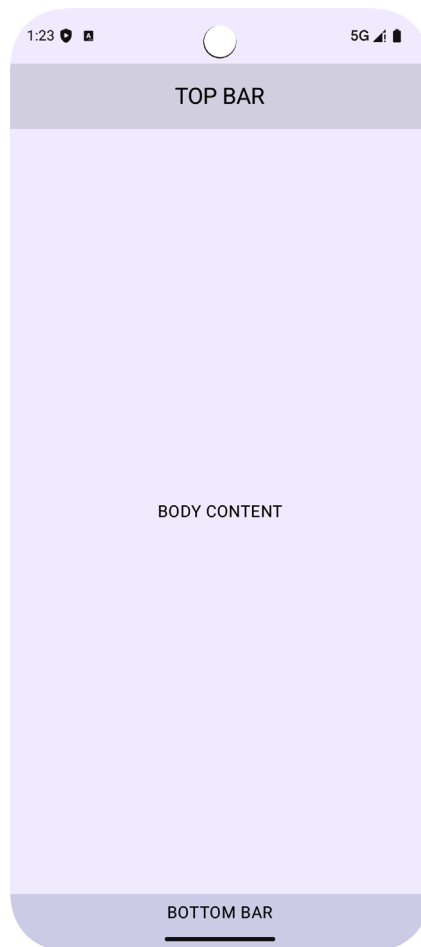


Figure 4.1 – Scaffold with topBar, bottomBar, and content

Here, you can see how Scaffold makes it easy to position content using recognized areas of an app. The topBar composable has CenterAlignedTopAppBar, which positions a title so that it's aligned in the center. To ensure it doesn't cover the status bar, it uses Modifier.statusBarsPadding(). The container color of the topBar composable is set to the default surface color with the centerAlignedTopAppBarColors composable. The bottomBar composable is set with BottomAppBar, which provides sensible defaults for placing a bottomBar composable containing a Text composable within a Box composable. Finally, the content composable is also set with a Text composable within a Box composable. Normally, the content argument would be left out as it's the last parameter of Scaffold, and its type can be inferred without explicitly naming it.

Now that we have the screen structure in place, let's learn how to add navigation.

Building a navigation graph

In Jetpack Compose, navigation is created using a `NavHost` composable. This hosts `NavGraph`, which defines your composable destinations; these are defined by routes. Navigating to and from these routes is done using `NavController`, which is passed into `NavHost` as its parameter name, `navController`. The builder parameter of `NavHost` (`builder: NavGraphBuilder.() -> Unit`) creates `NavGraph`. Look at the following example:

```
@Serializable
data object Home

@Serializable
data object Detail

@Composable
fun NavigationApp() {
    val navController = rememberNavController()
    NavHost(
        navController = navController,
        startDestination = Home,
        builder = (
            {
                composable<Home> {
                    HomeScreen(navController)
                }
                composable<Detail> {
                    DetailScreen(navController)
                }
            }
        ))
}

@Composable
fun HomeScreen(navController: NavController) {
```

```

Box(modifier = Modifier.fillMaxSize()) {
    Text(text = "Home Screen", fontSize=28.sp,
        modifier = Modifier.align(Alignment.TopCenter))

    Button(
        modifier = Modifier.align(Alignment.Center),
        onClick = { navController.navigate(Detail) },
    ) {
        Text(text = "Go to Detail Screen", fontSize= 22.sp)
    }
}

@Composable
fun DetailScreen(navController: NavController) {
    Box(modifier = Modifier.fillMaxSize()) {

        Text(text = "Detail Screen", fontSize=28.sp,
            modifier = Modifier.align(Alignment.TopCenter))

        Button(
            modifier = Modifier.align(Alignment.Center),
            onClick = { navController.navigate(HOME) },
        ) {
            Text(text = "Go to Home Screen", fontSize= 22.sp)
        }
    }
}

```

Here, you can see that `NavHost` sets up two screens, with `HOME` and `DETAIL` as route identifiers that are passed in for the builder argument. Since `builder` is the last parameter, it can be left out; this is the syntax that the upcoming exercises will use. Here, `navController` is passed into each composable and is used to manage navigation between the screens. Note that `rememberNavController` ensures that the navigation state is remembered over recompositions. The navigation graph is a structure that defines the relationships between composable destinations (nodes) within `NavHost`.

Each composable destination is a node in the graph, and `NavController` is responsible for navigating between these nodes. Note that these nodes are identified by routes. The navigation pattern used here is called **type-safe navigation**. Types are set as the routes, with the type providing the route identifier of the composable. In the preceding navigation graph, the `Home` object is used as the `Type` composable with `composable<Home>`. The declaration of the `Home` object looks like this:

```
@Serializable
data object Home
```

The `@Serializable` annotation creates a serializer that will convert your object into a `String` (usually JSON) before navigation and then convert the `String` back into an object when you arrive.

The basic navigation graph display that appears when the app is run can be seen in *Figure 4.2*.



Figure 4.2 – Screens from the navigation graph displayed

Next, we'll learn how to pass arguments between screens.

Exercise 4.1 – Building simple navigation

Now, you are going to combine Scaffold with NavHost to display a simple RGB color picker.

For all the exercises and activities you'll be completing in this chapter, you will need to add the Kotlinx Serialization plugin and the related dependencies. Open the `libs.versions.toml` file and follow along.

In the `[versions]` section, add the following:

```
kotlinxSerializationJson = "1.8.1"
navigationCompose = "2.9.0"
```

In the `[libraries]` section, add the following dependencies as two single lines:

```
kotlinx-serialization-json = { group = "org.jetbrains.kotlin", name =
    "kotlinx-serialization-json", version = "kotlinxSerializationJson" }
androidx-navigation-compose = { group = "androidx.navigation", name =
    "navigation-compose", version.ref = "navigationCompose" }
```

In the `[plugins]` section, add the following dependency as one single line:

```
kotlin-serialization = { id = "org.jetbrains.kotlin.plugin.serialization",
    version.ref = "kotlin" }
```

Once you've done this, select **Sync Now** from the top toolbar to download these dependencies.

In the `build.gradle` app, in the `plugins` block, add the following as one single line:

```
alias(libs.plugins.kotlin.serialization)
```

Finally, in the `build.gradle` app, in the `dependencies` block, add the following:

```
implementation(libs.kotlin.serialization.json) implementation(libs.
androidx.navigation.compose)
```



The app you are going to create in this exercise will display three different color tiles: red, green, and blue. Clicking on one of these tiles will navigate you to a screen that shows the body of the screen, where the background is of the selected color. For all the exercises in this chapter, to have a clear separation of concerns, you'll need to create a `Routes.kt` file and a `Screens.kt` file, both of which `MainActivity` will use.

Perform the following steps to complete this exercise:

1. Open Android Studio and select **New Project** on the Android welcome screen. Select **Empty Activity** and call the project `Simple Navigation`.
2. Create a separate file in the same package as `MainActivity.kt`, call it `Routes.kt`, and add a data object and a data class for `Home` and `Color`:

```
@Serializable
data object Home

@Serializable
data class Color(val name: String, val value: Long)
```

The preceding code contains the two routes (destination identifiers) you will use in the app. In addition to the name parameter, which will display the name of the color, there is also a value parameter with a `Long` data type, which is used to create the color. Colors can be represented as `Long` numeric values that consist of two hexadecimal characters for the alpha/transparency of the color, followed by two hexadecimal characters each for the red, green, and blue channels, which are defined from left to right. The value of `00` denotes completely transparent, while `FF` denotes completely opaque. For this exercise, we will use the following values, where `0x` represents the hexadecimal format:

```
RED: 0xFFFF0000 - Fully opaque, only red channel
GREEN: 0xFF00FF00 - Fully opaque, only blue channel
BLUE: 0xFF0000FF - Fully opaque, only green channel
```

By default, only simple data types can be used in type-safe navigation, which is why we are using the numeric values of colors rather than a complex `Color` type.

3. Create a new file called `Screens.kt` and add a `ColorScreen` screen. This will be used to display whichever color value is passed into the screen:

```
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun ColorScreen(navController: NavController, colorName: String,
    colorValue: Long) {
    Scaffold(
        topBar = {
            CenterAlignedTopAppBar(
                modifier = Modifier.fillMaxWidth(),
```



```

        title = {
            Text(stringResource(R.string.app_name))
        },
        navigationIcon = {
            IconButton(onClick = {
                navController.navigateUp() }) {
                Icon(Icons.AutoMirrored.Filled.ArrowBack,
                    contentDescription = "Back")
            }
        }
    )
}
) { padding ->
    Box(
        modifier = Modifier
            .padding(padding)
            .fillMaxSize()
            .background(Color(colorValue)),
        contentAlignment = Alignment.Center
    ) {
        Text(text = "$colorName SCREEN", fontSize = 24.sp)
    }
}
}

```

The preceding code adds a top bar that's populated with the name of the color. There is also a back arrow icon, which, combined with the passed-in `navController`, navigates back up to the screen it was launched from using `navController.navigateUp()`. As `topBar` is an experimental feature, you have to annotate it with `@OptIn(ExperimentalMaterial3Api::class)` at the top of the composable. The body of the screen has a `Box` composable, which occupies the maximum size of the screen with `Modifier.fillMaxSize()`, and the background is set to the passed-in hexadecimal color value, which is created through the `Color` type constructor, `Color(colorValue)`. The name of the color is added as a `Text` composable, and it is centered in the box using `contentAlignment` set to `Alignment.Center`.

4. Add the `HomeScreen` composable, which displays the three color tiles in a column:

```

@OptIn(ExperimentalMaterial3Api::class)
@Composable

```

```
fun HomeScreen(navController: NavController) {
    Scaffold(
        topBar = {
            CenterAlignedTopAppBar(
                title = { Text(stringResource(R.string.app_name)) },
                modifier = Modifier.statusBarsPadding(),
                colors =
                    TopAppBarDefaults.centerAlignedTopAppBarColors(
                        containerColor =
                            MaterialTheme.colorScheme.surfaceContainer
                    )
            )
        }
    ) { padding ->
        Column(
            modifier = Modifier
                .padding(padding)
                .fillMaxSize(),
            verticalArrangement = Arrangement.Center,
            horizontalAlignment = Alignment.CenterHorizontally
        ) {
            Box(
                modifier = Modifier
                    .size(150.dp)
                    .border(4.dp, Color(0xFFFF0000), shape =
                        RoundedCornerShape(16.dp))
                    .clickable { navController.navigate(Color("RED",
                        0xFFFF0000))}
                    .padding(16.dp),
                contentAlignment = Alignment.Center
            ) {
                Text("RED", color = Color(0xFFFF0000),
                    fontSize = 24.sp)
            }

            Spacer(modifier = Modifier.height(24.dp))

            Box(
                modifier = Modifier
```

```

        .size(150.dp)
        .border(4.dp, Color(0xFF00FF00),
            shape = RoundedCornerShape(16.dp))
        .clickable { navController.
            navigate(Color("GREEN", 0xFF00FF00))}
        .padding(16.dp),
        contentAlignment = Alignment.Center
    ) {
        Text("GREEN", color = Color(0xFF00FF00),
            fontSize = 24.sp)
    }

    Spacer(modifier = Modifier.height(24.dp))

    Box(
        modifier = Modifier
            .size(150.dp)
            .border(4.dp, Color(0xFF0000FF),
                shape = RoundedCornerShape(16.dp))
            .clickable { navController.
                navigate(Color("BLUE", 0xFF0000FF))}
            .padding(16.dp),
        contentAlignment = Alignment.Center
    ) {
        Text("BLUE", color = Color(0xFF0000FF),
            fontSize = 24.sp) // Large font size
    }
}
}
}

```

5. Finally, create NavHost in a separate composable and add it to the onCreate function of MainActivity to create the source composables and assign them to a route:

```

@Composable
fun NavigationApp() {
    val navController = rememberNavController()

```

```

        NavHost(navController = navController,
            startDestination = Home) {
            composable<Home> { HomeScreen(navController) }
            composable<Color> { navBackStackEntry ->
                val color = navBackStackEntry.toRoute<Color>()
                ColorScreen(navController, color.name, color.value) }
        }
    }
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            SimpleNavigationTheme {
                NavigationApp()
            }
        }
    }
}

```

In this example, the name of the builder argument of `NavGraphBuilder` is not used as it can be inferred by its position as the last argument. The interesting point to observe here is the clickable handlers on each of the `Box` composable tiles:

```

Box(
    modifier = Modifier
        .size(150.dp)
        .border(4.dp, Color(0xFF00FF00),
            shape = RoundedCornerShape(16.dp))
        .clickable { navController.navigate(Color("GREEN", 0xFF00FF00))}
        .padding(16.dp),
    contentAlignment = Alignment.Center)

```

Here, `navcontroller` is used to navigate to `composable<Color>`. Once the destination has been navigated to, the `Color` type and its arguments can be retrieved from `NavBackStackEntry`, which is a single entry point on your app's back stack of destinations. As the latest entry is the `Color` route, we can assign a property with `navBackStackEntry.toRoute<Color>()` and use the arguments to populate the `ColorScreen` composable.

Run the app; you should see the following Home screen and all three Color screens upon selecting each of the tiles:

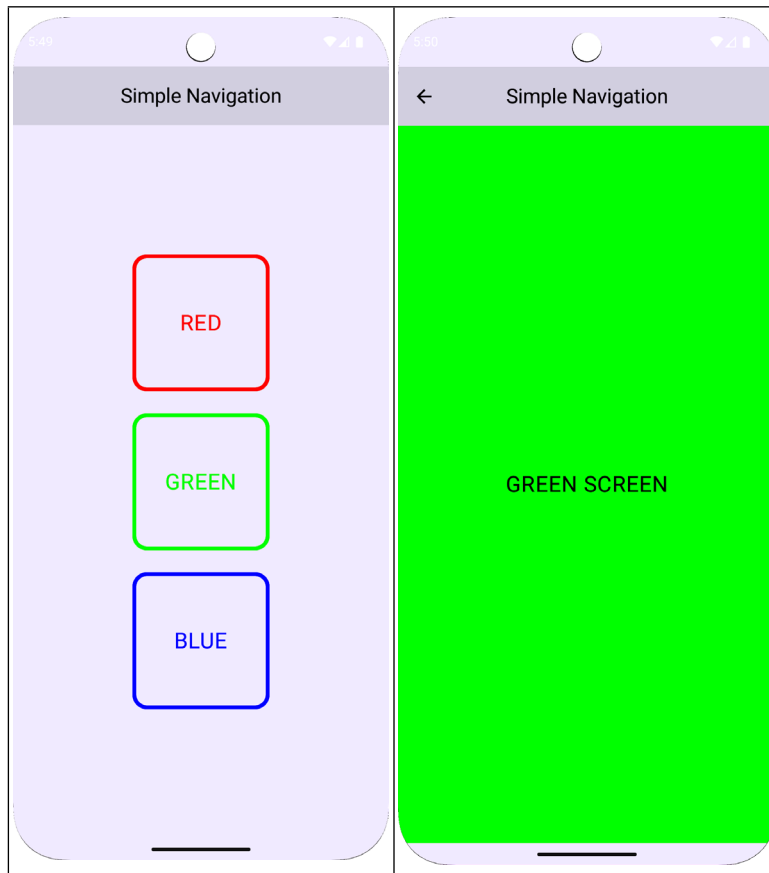


Figure 4.3 – The Simple Navigation app with NavHost and routes

The legacy method of creating routes and then navigating to and from them involves using plain String Objects as routes. To illustrate this, let's look at a similar example of using a String route to define a color destination in NavHost and then navigate to it.

The destination can be set up like so:

```
composable(  
    route = "color/{name}/{value}",  
    arguments = listOf(  
        navArgument("name") { type = NavType.StringType },  
        navArgument("value") { type = NavType.StringType }
```

```
    )  
  ) { navBackStackEntry ->  
      val colorName = navBackStackEntry.arguments?.getString("name") ?: ""  
      val colorValue = navBackStackEntry.arguments?.getString("value") ?: ""  
      ColorScreen(navController, colorName, colorValue)  
  }  
}
```

We can navigate to the destination with a route:

```
navController.navigate("color/Red/FF0000")
```

This way of both defining destinations and navigating to and from them with routes is very verbose and particularly error-prone. This is because the arguments in the route need to be encoded correctly in navigating to a destination, and the receiving arguments when the destination has been navigated to need to be typed correctly. This explains why type-safe navigation is a distinct improvement and the preferred way to set up navigation in your apps.

Now that you have got to grips with the fundamentals of navigation via navigation graphs, navigation hosts, navigation controllers, routes, and screens, and the structure of an app using Scaffold, you are equipped to study the principal Android navigation patterns.

A guiding principle of each of the three primary navigation patterns is to contextually provide information about the main section of the app the user is in at any point in time.

This can take the form of a label in the top app bar of the destination, optionally displaying an arrow hint indicating that the user is not at the top level, and/or providing highlighted text and icons in the UI that indicate the section the user is in. In your app, navigation should be fluid and natural, intuitively guiding the user.

Each of the three navigation patterns you will explore accomplishes this goal in varying ways. Some of these navigational patterns are more suitable for use with a higher number of top-level primary destinations to display, while others are suitable with fewer.

Let's look at the first pattern to be widely adopted.

Implementing a navigation drawer

The **navigation drawer** is one of the most common navigation patterns used in Android apps. The following screenshot shows the culmination of the next exercise, which shows a simple navigation drawer in its closed state:

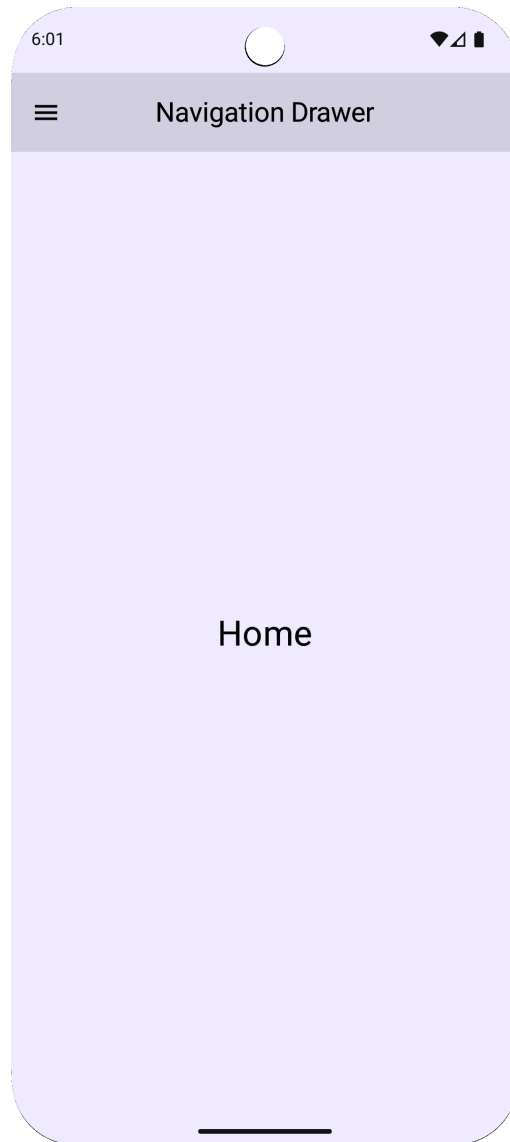


Figure 4.4 – App with the navigation drawer closed

The navigation drawer can be accessed through what has become commonly known as the **hamburger menu**, which is the icon with three horizontal lines in the top-left corner of *Figure 4.4*.

Upon selecting the hamburger menu, the navigation drawer slides out from the left, with the current section highlighted. This can be displayed with or without an icon. Due to the nature of the navigation occupying the height of the screen, it is best suited to five or more top-level destinations:

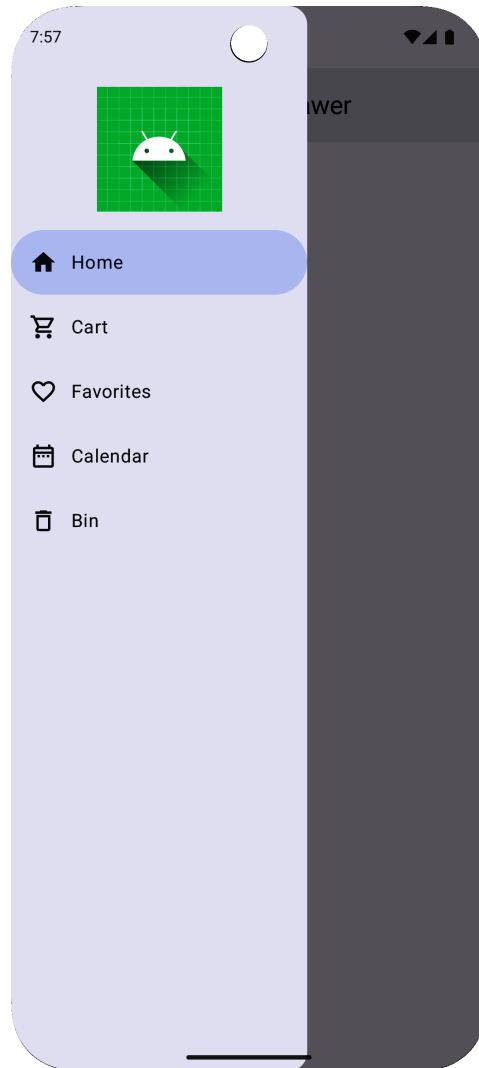


Figure 4.5 – App with the navigation drawer open

A weakness of the navigation drawer is that it requires the user to select the **hamburger menu** for the destinations to become visible. In contrast, with bottom navigation, they are always visible, and with tabbed navigation, they are partially seen, depending on how many tabs there are and whether they are scrollable. However, this is also a strength of the navigation drawer as more screen space can be used for the app's content.

Now that you have been introduced to the key concepts of navigation, let's put it all together and build an app with a navigation drawer.

Exercise 4.2 – Creating an app with a navigation drawer

Let's create a simple app that showcases the features of the navigation drawer:

1. Open Android Studio and select **New Project** on the Android welcome screen. Select **Empty Activity** and call it **Navigation Drawer**.
2. Create a `Screens.kt` file that contains a simple screen with a name:

```
@Composable
fun ContentScreen(name: String) {
    Box(
        modifier = Modifier.fillMaxSize(),
        contentAlignment = Alignment.Center
    ) {
        Text(name, fontSize = 28.sp)
    }
}
```

3. Create a `Routes.kt` file:

```
@Serializable
sealed class Destination(val label: String) {
    @Serializable
    data object Home : Destination("Home")

    @Serializable
    data object Shopping : Destination("Cart")

    @Serializable
    data object Favorites : Destination("Favorites")

    @Serializable
    data object Calendar : Destination("Calendar")

    @Serializable
    data object Bin : Destination("Bin")
}

sealed class NavigationDrawer(val label: String, val selectedIcon:
    ImageVector, val unselectedIcon: ImageVector, val route:
```

```

        Destination) {

    data object Home : NavigationDrawer("Home", Icons.Filled.Home,
        Icons.Outlined.Home, Destination.Home)

    data object Shopping : NavigationDrawer("Cart",
        Icons.Filled.ShoppingCart, Icons.Outlined.ShoppingCart,
        Destination.Shopping)

    data object Favorites : NavigationDrawer("Favorites",
        Icons.Filled.Favorite, Icons.Outlined.FavoriteBorder,
        Destination.Favorites)

    data object Calendar : NavigationDrawer("Calendar",
        Icons.Filled.DateRange, Icons.Outlined.DateRange,
        Destination.Calendar)

    data object Bin : NavigationDrawer("Bin", Icons.Filled.Delete,
        Icons.Outlined.Delete, Destination.Bin)
}

```

Notice that, as well as the routes in the sealed class of `Destination` being defined, another sealed class called `NavigationDrawer` is defined. This will be used for each of the navigation drawer items to display a label, selected and unselected icons, and the route `Destination` it will link to.

4. Create a composable that contains `Scaffold` and `NavHost` in `MainActivity`:

```

@Composable
@OptIn(ExperimentalMaterial3Api::class)
fun NavigationDrawerHost(coroutineScope: CoroutineScope,
    drawerState: DrawerState, navController: NavHostController) {
    Scaffold(
        topBar = {
            CenterAlignedTopAppBar(
                title = { Text(stringResource(R.string.app_name)) },
                modifier = Modifier.statusBarsPadding(),
                colors = TopAppBarDefaults.
            centerAlignedTopAppBarColors(

```

```

        containerColor = MaterialTheme.colorScheme.
surfaceContainer
    ),
    navigationIcon = {
        IconButton(onClick = {
            coroutineScope.launch {
                drawerState.open()
            }
        }) {
            Icon(Icons.Default.Menu,
                contentDescription = "Menu")
        }
    }
)
}
) { innerPadding ->
    NavHost(
        navController = navController,
        modifier = Modifier.padding(innerPadding),
        startDestination = Destination.Home
    ) {
        composable<Destination.Home> {
            ContentScreen(Destination.Home.label) }
        composable<Destination.Shopping> {
            ContentScreen(Destination.Shopping.label) }
        composable<Destination.Favorites> {
            ContentScreen(Destination.Favorites.label) }
        composable<Destination.Calendar> {
            ContentScreen(Destination.Calendar.label) }
        composable<Destination.Bin> {
            ContentScreen(Destination.Bin.label) }
    }
}
}

```

This is similar to what you've seen previously; it is the full screen of the app. There is a Scaffold composable with a topBar composable and a NavHost composable that contains all the destinations. What's new is navigationIcon, defined in topBar. Here,

`Icons.Default.Menu` is the hamburger menu. When `onClick` is pressed, it opens the navigation drawer with `drawState`, which holds the `Open` or `Closed DrawerValue` value of the navigation drawer. Note that here, `coroutineScope` is used for asynchronous operations that do not block the main thread. You will learn about coroutines in *Chapter 10, Coroutines and Flow*. For now, you can think of it as a lightweight thread.

5. Now, you can start to create the `MainApp` composable, which will hold the navigation drawer and the navigation graph:

```
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun MainApp() {
    val navController: NavHostController = rememberNavController()
    val navBackStackEntry by
        navController.currentBackStackEntryAsState()

    val currentDestination = navBackStackEntry?.destination
    val navigationDrawerItems = listOf(
        NavigationDrawer.Home,
        NavigationDrawer.Shopping,
        NavigationDrawer.Favorites,
        NavigationDrawer.Calendar,
        NavigationDrawer.Bin
    )
    val drawerState =
        rememberDrawerState(initialValue =
            androidx.compose.material3.DrawerValue.Closed)
    val coroutineScope = rememberCoroutineScope()
    // Add the ModalNavigationDrawer here in the next step
}
```

A key concept in navigation is determining which destination in `NavGraph` is currently selected, and which is the route that identifies it. You can do this by retrieving the back stack from `navController` and then getting this destination:

```
val navController: NavHostController = rememberNavController()
val navBackStackEntry by
    navController.currentBackStackEntryAsState()
val currentDestination = navBackStackEntry?.destination
```



```

        contentDescription = "Logo",
        modifier = Modifier.padding(16.dp)
    )
}

navigationDrawerItems.forEach { item ->
    val isSelected = currentDestination?.hasRoute(
        item.route::class) == true

    NavigationDrawerItem(
        icon = {
            Icon(
                imageVector = if (isSelected) item.
                    selectedIcon else item.unselectedIcon,
                contentDescription = item.label
            )
        },
        label = { Text(item.label) },
        selected = isSelected,
        onClick = {
            navController.navigate(item.route) {
                launchSingleTop = true
                restoreState = true
                popUpTo(
                    navController.graph.startDestinationId) {
                    saveState = true
                }
            }
            coroutineScope.launch {
                drawerState.close()
            }
        }
    )
}
}
}
) {

```

```

        NavigationDrawerHost(coroutineScope, drawerState, navController)
    }

```

Here, `ModalNavigationDrawer` is the container for `NavigationDrawerHost`. Note that in this case, `Modal` refers to the state of the drawer blocking interaction. There's content beneath this that creates an overlay that dims the content beneath it. Additionally, `drawerContent` is set to `ModalDrawerSheet`, which slides in and out from the left-hand side; this is where you place the content. An image is typically shown at the top, with the `Image` composables set in the `Box` composable. This acts as a launcher icon. The `navigationDrawerItemList` is looped through to add all the `NavigationDrawerItem`s and set the label text. The selected state of the drawer item is checked and set using `currentDestination?.hasRoute(item.route::class) == true`, which is also used to display the selected or not selected icon in the navigation drawer.

Finally, `onClick{}` is set to navigate to the selected route, which is set to the current index with `drawerItemIndex = index`, after which it closes the navigation drawer with `drawerState.close()`. Note that `launchSingleTop = true` ensures that no matter how many times one of the navigation items is repeatedly clicked, it is treated as only being clicked once for navigation purposes. We also have `popUpTo(navController.graph.startDestinationId)`. This sets the back navigation to go back to `startDestinationId`, which is the Home route, `AppDestination.Home`.

7. Finally, add the `MainApp` composable to the `onCreate` function:

```

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    enableEdgeToEdge()
    setContent {
        NavigationDrawerTheme {
            MainApp()
        }
    }
}

```

8. Run the app and open the navigation drawer. You should see the screen shown in *Figure 4.5*.

The next navigational pattern you'll explore is bottom navigation. This has become the most popular navigational pattern in Android, largely because it makes all the main sections of the app easily accessible.

Adding bottom navigation

Bottom navigation is used when there are a limited number of top-level destinations, and these can range from three to five primary destinations that are not related to each other. Each item on the bottom navigation usually displays an icon and a text label, although using solely icons or text labels is possible.

This navigation allows quick access as the items are always available, no matter which secondary destination of the app the user navigates to.

You will build an app with bottom navigation using the same navigation items that you did in *Exercise 4.2 – Creating an app with a navigation drawer*, to compare these two navigation patterns.

Exercise 4.3 – Adding bottom navigation to your app

The steps for creating a NavGraph composable with routes and screens are similar to those for implementing the navigation drawer:

1. Open Android Studio and select **New Project** on the Android welcome screen. Select **Empty Activity** and call it Bottom Navigation.
2. Create the Screens.kt file and add ContentScreen as you did in *Exercise 4.2 – Creating an app with a navigation drawer*.
3. Create a Routes.kt file:

```
@Serializable
sealed class Destination(val label: String) {
    @Serializable
    data object Home: Destination("Home")

    @Serializable
    data object Shopping: Destination("Cart")

    @Serializable
    data object Favorites: Destination("Favorites")

    @Serializable
    data object Calendar: Destination("Calendar")

    @Serializable
    data object Bin: Destination("Bin")
}
```



```

    }

    sealed class BottomNavigation(val label: String, val selectedIcon:
        ImageVector, val unselectedIcon: ImageVector, val badgeCount:
            Int, val route: Destination)
    {
        data object Home : BottomNavigation("Home", Icons.Filled.Home,
            Icons.Outlined.Home, 0, Destination.Home )

        data object Shopping : BottomNavigation("Cart",
            Icons.Filled.ShoppingCart, Icons.Outlined.ShoppingCart, 0,
            Destination.Shopping )

        data object Favorites : BottomNavigation("Favorites",
            Icons.Filled.Favorite, Icons.Outlined.FavoriteBorder, 0,
            Destination.Favorites )

        data object Calendar : BottomNavigation("Calendar",
            Icons.Filled.DateRange, Icons.Outlined.DateRange, 1,
            Destination.Calendar )

        data object Bin : BottomNavigation("Bin", Icons.Filled.Delete,
            Icons.Outlined.Delete, 0, Destination.Bin )
    }

```

This is very similar to the previous `Routes.kt` file for the navigation drawer. The main difference is that here, there is an `Int` property called `badgeCount` that will be used to display dynamic information about the bottom navigation items.

4. Create a `MainApp` composable so that you can add the app's `Scaffold` composable and the `NavHost` composable:

```

@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun MainApp() {
    val navController: NavHostController = rememberNavController()
    val navBackStackEntry by
        navController.currentBackStackEntryAsState()
    val currentDestination = navBackStackEntry?.destination

    val bottomNavigationItems = listOf(

```

```
        BottomNavigation.Home,
        BottomNavigation.Shopping,
        BottomNavigation.Favorites,
        BottomNavigation.Calendar,
        BottomNavigation.Bin
    )

    Scaffold(
        topBar = {
            CenterAlignedTopAppBar(
                title = { Text(stringResource(R.string.app_name)) },
                modifier = Modifier.statusBarsPadding(),
                colors =
                    TopAppBarDefaults.centerAlignedTopAppBarColors(
                        containerColor =
                            MaterialTheme.colorScheme.surfaceContainer
                    )
            )
        },
        bottomBar = {
            // Add Navigation Bar
        }
    ) { innerPadding ->
        NavHost(
            navController = navController,
            modifier = Modifier.padding(innerPadding),
            startDestination = Home
        ) {
            composable<Home> { ContentScreen(Home.label) }
            composable<Shopping> { ContentScreen(Shopping.label) }
            composable<Favorites> { ContentScreen(Favorites.label) }
            composable<Calendar> { ContentScreen(Calendar.label) }
            composable<Bin> { ContentScreen( Bin.label) }
        }
    }
}
```

This follows very similar syntax to what was shown in the navigation drawer exercise. Here, you set `currentDestination` from the current `NavBackStackEntry` class. Then, you created a list of navigation items with `bottomNavigationItems`. The `topBar` and `NavHost` composables have been set in the same way as before. All that remains to tie everything together is to set `bottomBar`.

5. Replace the `Add Navigation Bar` comment with the following and add `import 'androidx.navigation.NavDestination.Companion.hasRoute'` to the imports list.

```
NavigationBar {
    bottomNavigationItems.forEach { item ->

        val isSelected =
            currentDestination?.hasRoute(item.route::class) == true
        NavigationBarItem(
            selected = isSelected,
            icon = {
                BadgedBox(
                    badge = {
                        if (item.badgeCount > 0) {
                            Badge { Text(item.badgeCount.toString())
                        }
                    }
                ) {
                    Icon(
                        imageVector = if (isSelected)
                            item.selectedIcon else
                            item.unselectedIcon,
                        contentDescription = item.label
                    )
                }
            },
            label = { Text(item.label) },
            onClick = {
                if (!isSelected) {
```

```

                                navController.navigate(item.route) {
                                    launchSingleTop = true
                                    restoreState = true
                                    popUpTo(
                                        navController.graph.startDestinationId) {
                                        saveState = true
                                    }
                                }
                            }
                        }
                    }
                )
            }
        }
    }
}

```

Each `NavigationBarItem` is added by looping through `bottomNavigationItems`. As you can see, `currentDestination` is evaluated to determine whether the current route is the same as the route for the navigation item, after which it sets the selected state of the item and icons. All the text that's displayed and the navigation provided are the same as in the navigation drawer exercise. The difference is that the icon is set with `BadgedBox`, which allows dynamic information to be set. It can be set to display an icon, but it is more commonly used to show a short piece of text. In this case, this text displays a number in the Calendar item to represent a notification.

6. Now, add the `MainApp` composable to the `onCreate` function:

```

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    enableEdgeToEdge()
    setContent {
        BottomNavigationTheme {
            MainApp()
        }
    }
}

```

7. Run the app. You should see the screen shown in *Figure 4.6*.

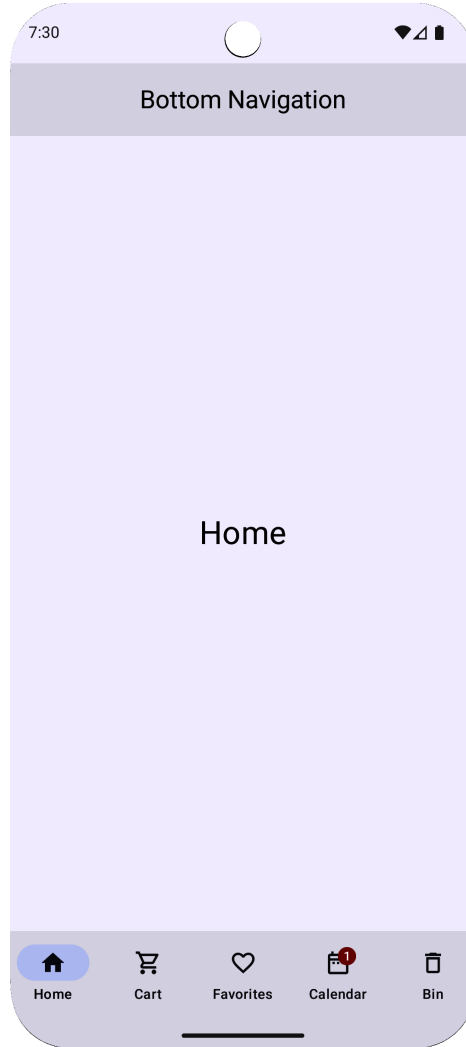


Figure 4.6 – Bottom navigation with Home selected

In this exercise, you learned that setting up bottom navigation is quite straightforward and significantly less complex than setting up the navigation drawer. If you are developing an app that has very well-defined top-level destinations and switching between them is important, then the visibility of these destinations makes bottom navigation an ideal choice.

The final primary navigation pattern that you will explore is tabbed navigation. This is a versatile pattern as it can be used as an app's primary navigation and as secondary navigation with the other navigation patterns we've studied.

Introducing tabbed navigation

Tabbed navigation is mostly used when you want to display related items. It is common to have fixed tabs if there are only a few of them (typically between three and five tabs) and scrolling horizontal tabs if you have more than five. They are used mostly for grouping destinations that are at the same hierarchical level.

This might be the case if the app you are developing has a narrow or specific subject field where the primary destinations are related, such as a news app. More commonly, it is used with bottom navigation to present secondary navigation that's available within a primary destination.

Exercise 4.4 – Using tabs for app navigation

The following exercise demonstrates using scrollable tabbed navigation to create a news app:

1. Open Android Studio and select **New Project** on the Android welcome screen. Select **Empty Activity** and call it **Tab Navigation**.
2. Create the `Screens.kt` file and add the `ContentScreen` as you did in *Exercise 4.2 – Creating an app with a navigation drawer*.
3. Create a `Routes.kt` file and add the following to it:

```
@Serializable
sealed class Destination(val label: String,) {
    @Serializable
    data object TopStories : Destination("Top Stories")

    @Serializable
    data object UKNews : Destination("UK News")

    @Serializable
    data object Politics : Destination("Politics")

    @Serializable
    data object Business : Destination("Business")

    @Serializable
    data object WorldNews : Destination("World News")

    @Serializable
```

```

    data object Sport : Destination("Sport")

    @Serializable
    data object Other : Destination("Other")
}

sealed class TabNavigation(val label: String,
    val route: Destination) {

    data object TopStories : TabNavigation("Top Stories",
        Destination.TopStories )

    data object UKNews : TabNavigation("UK News",
        Destination.UKNews )

    data object Politics : TabNavigation("Politics",
        Destination.Politics )

    data object Business : TabNavigation("Business",
        Destination.Business )

    data object WorldNews : TabNavigation("World News",
        Destination.WorldNews )

    data object Sport : TabNavigation("Sport", Destination.Sport )

    data object Other : TabNavigation("Other", Destination.Other )
}

```

Here, you are only setting the label and the route. You can add icons with tab navigation, but due to space constraints, it is not common to do so.

4. Create a MainApp composable:

```

@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun MainApp() {
    val navController = rememberNavController()
    val navBackStackEntry by

```

```
        navController.currentBackStackEntryAsState()
    val currentDestination = navBackStackEntry?.destination

    val tabNavigationItems = listOf(
        TabNavigation.TopStories,
        TabNavigation.UKNews,
        TabNavigation.Politics,
        TabNavigation.Business,
        TabNavigation.WorldNews,
        TabNavigation.Sport,
        TabNavigation.Other
    )

    var tabIndex by rememberSaveable { mutableIntStateOf(0) }

    Scaffold(
        topBar = {
            Column {
                CenterAlignedTopAppBar(
                    title = {
                        Text(stringResource(R.string.app_name)) },
                    modifier = Modifier.statusBarsPadding(),
                    colors =
                        TopAppBarDefaults.centerAlignedTopAppBarColors(
                            containerColor =
                                MaterialTheme.colorScheme.surfaceContainer
                        )
                )
            }
        },
        // Add Tab Bar
    ) {
        }
    ) { paddingValues ->
        NavHost(
            navController = navController,
```



```

        startDestination = TopStories,
        modifier = Modifier.padding(paddingValues)
    ) {
        composable<TopStories> { ContentScreen(TopStories.label) }

        composable<UKNews> { ContentScreen(UKNews.label) }
        composable<Politics> { ContentScreen(Politics.label) }
        composable<Business> { ContentScreen(Business.label) }
        composable<WorldNews> { ContentScreen(WorldNews.label) }
        composable<Sport> { ContentScreen(Sport.label) }
        composable<Other> { ContentScreen(Other.label) }
    }
}

```

The preceding code is very similar to what was provided for the previous bottom navigation exercise and sets up the current destination, creates a list of navigation items, and adds the Scaffold and NavHost composables. The key difference is the following line:

```
var tabIndex by rememberSaveable { mutableIntStateOf(0) }
```

Tabbed navigation sets `selectedTabIndex` on the `TabRow` composable, as well as the individual item, so you will need to evaluate and update this index within each navigation item.

5. Remove the `Add Tab Bar` comment and replace it with the following and add import `'androidx.navigation.NavDestination.Companion.hasRoute'` to the imports list.

```

PrimaryScrollableTabRow(selectedTabIndex = tabIndex,
    edgePadding = 0.dp) {
    tabNavigationItems.forEachIndexed { index, item ->
        val isSelected =
            currentDestination?.hasRoute(item.route::class) == true
        if (isSelected) tabIndex = index

        Tab(

```

```

        selected = isSelected ,
        text = { Text(item.label) },
        onClick = {
            if (!isSelected) {
                navController.navigate(item.route) {
                    popUpTo(navController.graph.startDestinationId)
                    launchSingleTop = true
                }
            }
        }
    )
}
}
}

```

Here, the `tabNavigationItems` list is used to create the tabs. On this occasion, they are looped around using `forEachIndexed`, which provides the index of the item so that it can be set as the current index when it is displayed:

```
if (isSelected) tabIndex = index
```

Note that `PrimaryScrollableTabRow` is used here with `edgePadding = 0.dp` to lay out the tabs from the start of the Tab row with no inset.

6. Finally, add the `MainApp` composable to `onCreate`:

```

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    enableEdgeToEdge()
    setContent {
        TabNavigationTheme {
            MainApp()
        }
    }
}
}

```

7. Run the app. It should display the screen shown in *Figure 4.7*.

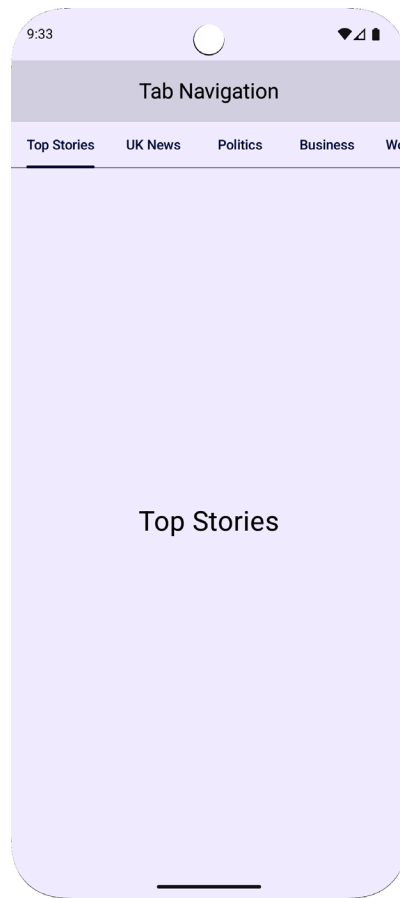


Figure 4.7– Scrollable tabs with Top Stories selected

The list of tabs continues to be displayed offscreen. More advanced features, such as swiping left and right between the tabs, can be achieved using `PagerState` with a `HorizontalPager` composable. For more details, go to <https://packt.link/xy9Cx>.

To set fixed-width tabs, change `PrimaryScrollableTabRow` to `PrimaryTabRow`:

```
PrimaryTabRow(selectedTabIndex = tabIndex)
```

Then, comment out the last three `tabNavigationItems`:

```
val tabNavigationItems = listOf(  
    TabNavigation.TopStories,  
    TabNavigation.UKNews,  
    TabNavigation.Politics,  
    TabNavigation.Business,  
    TabNavigation.Wo
```

```
TabNavigation.Politics,  
TabNavigation.Business,  
// TabNavigation.WorldNews,  
// TabNavigation.Sport,  
// TabNavigation.Other  
)
```

Upon running the app, it should look like this:

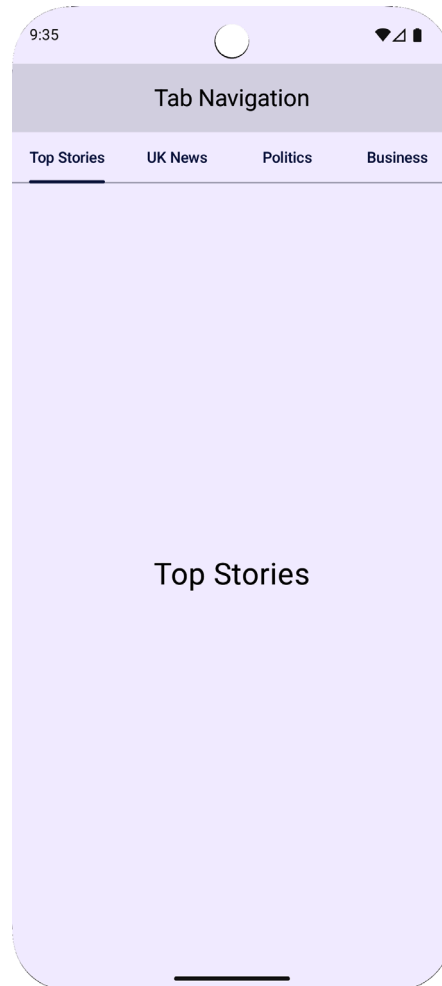


Figure 4.8 – Fixed tabs with Top Stories selected

The tabs can be swiped and selected, and the body content can also be swiped so that you can go left and right through the tab pages.

Now that you've learned about the most fundamental navigational patterns, let's use them to create an app.

Activity 4.1 – Building primary and secondary app navigation

You have been tasked with creating a sports app. It can have three or more top-level destinations. One of the primary destinations, however, must be called *My Sports* and should link to one or more secondary destinations, which are individual sports. You can choose whichever navigational pattern you think is most suitable, or a combination of them, and you can also introduce any customizations that you feel are appropriate.

There are different ways of attempting this activity. One approach would be to use bottom navigation and add the individual secondary sports destinations to the `NavHost` composable so that you can easily link to these destinations. It is fairly simple and delegates to the navigation graph using routes. Here is what the home screen should look like after implementing this approach:

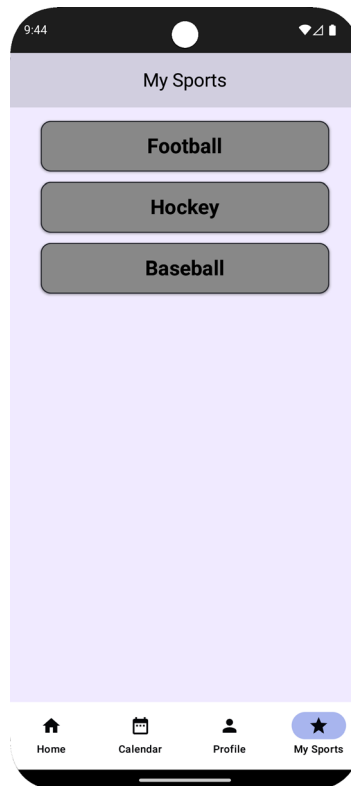


Figure 4.9 – Bottom navigation for the *My Sports* app



A solution to this activity can be found at <https://packt.link/tTKZr>.

Summary

This chapter covered the most important navigation techniques you need to know about to create clear and consistent navigation in your apps. You started by learning how to build screen structure with Scaffold and slots, after which you proceeded to create a navigation graph to navigate between destination composables using routes. Here, you were introduced to the navigation drawer pattern so that you can connect navigation menu items to composables.

Then, you implemented bottom navigation to display primary navigation destinations that are always visible on the screen. We followed this by looking at tabbed navigation, where you learned how to display scrollable tabs. For each navigational pattern, you were shown when it might be more suitable, depending on the type of app you are building. You finished this chapter by building your own app using one or more of these navigational patterns.

In the next chapter, you will build on this knowledge by using Android libraries to make network requests and process the responses to display textual data and images.

Unlock this book's exclusive benefits now

Scan this QR code or go to packtpub.com/unlock, then search this book by name.

Note: Keep your purchase invoice ready before you start.

UNLOCK NOW



Part 2

App Components

This part provides you with tools to develop dynamic Android apps. It starts by teaching you how to make network requests and display images from remote URLs. It then demonstrates how to present users with lists of items and explains how to make these lists interactive. Moving on, you will learn about the permission system in Android and use your newly acquired knowledge to implement an interactive map. Finally, you will be exposed to the tools provided by Android for performing background tasks, even when the app is not visible to the user.

This part of the book includes the following chapters:

- *Chapter 5, Essential Libraries – Ktor, Kotlin Serialization, and Coil*
- *Chapter 6, Building Lists with Jetpack Compose*
- *Chapter 7, Android Permissions and Google Maps*
- *Chapter 8, Services, WorkManager, and Notifications*

5

Essential Libraries – Ktor, Kotlin Serialization, and Coil

In the previous chapter, we learned how to implement navigation in our app. In this chapter, we will learn how to present dynamic content to the user as they navigate our app. Here, we will cover the steps needed to present app users with dynamic content fetched from remote servers. You will also be introduced to the different libraries required to retrieve and handle this dynamic data.

By the end of this chapter, you will be able to fetch data from a network endpoint using **Ktor**, parse **JavaScript Object Notation (JSON)** payloads into Kotlin data objects using **Kotlin Serialization**, and load images into **AsyncImage** composables using **Coil**.

We will cover the following topics in this chapter:

- Introducing REST, API, JSON, and XML
- Fetching data from a network endpoint
- Parsing a JSON response
- Loading images from a remote URL

Technical requirements

The complete code for all the exercises and the activity in this chapter is available on GitHub at <https://packt.link/1f0a2>.

Introducing REST, API, JSON, and XML

Data presented to users can come from different sources. It can be hardcoded into an app (static), but that comes with limitations. To change hardcoded data, we have to publish an update to our app. Some data (such as currency exchange rates, the real-time availability of assets, and the current weather) cannot be hardcoded due to its nature (it's dynamic). Other data may become outdated, such as the terms of use of an app.

In such cases, you usually fetch the relevant data from a server. One of the most common architectures for serving such data is the **Representational State Transfer (REST)** architecture.

REST and APIs

The REST architecture is defined by a set of six constraints: *client-server architecture*, *statelessness*, *cacheability*, a *layered system*, *code on demand* (optional), and a *uniform interface*.



To learn more about REST, visit <https://packt.link/S01UB>.

When applied to a web service **application programming interface (API)**, we get a **Hypertext Transfer Protocol (HTTP)**-based RESTful API. The HTTP protocol is the foundation of data communication for the World Wide Web, hosted on and accessible via the internet. It is the protocol used by servers all around the world to serve websites to users in the form of HTML documents, images, style sheets, and so forth.



An interesting article on this topic can be found at <https://packt.link/Lz5km>.
You can learn more about HTTP here: <https://packt.link/7qifp>.

RESTful APIs rely on the standard HTTP methods—GET, POST, PUT, DELETE, and PATCH—to fetch and transform data. These methods allow us to fetch, store, delete, and update data entities on remote servers.

The Android platform includes an `HttpURLConnection` client to execute these HTTP methods. However, it has limited capabilities, so in modern Android projects, we commonly rely on third-party libraries instead. Common libraries for this purpose are **Retrofit** and **Ktor**. In this chapter, we will use Ktor. Ktor is an HTTP client and server library that was developed by JetBrains using Kotlin. We will use it to fetch data from web APIs.

Most commonly, such data is represented by JSON.

JSON

JSON is a text-based data transfer format. As the name implies, it was derived from JavaScript. In fact, a JSON string can be used as-is as a valid data structure in JavaScript. JSON has become one of the most popular standards for data transfer, and its most modern programming languages have libraries that encode or decode data to or from JSON.

A simple JSON payload may look something like this:

```
{
  "employees": [
    {
      "name": "James",
      "email": "james.notmyemail@gmail.com"
    },
    {
      "name": "Lea",
      "email": "lea.dontemailme@gmail.com"
    }
  ]
}
```

XML

Another common data structure used by RESTful services is **Extensible Markup Language (XML)**, which encodes documents in a human and machine-readable format. XML is considerably more verbose than JSON. The same data structure that was shown in the previous example, but in XML, would look something like this:

```
<employees>
  <employee>
    <name>James</name>
    <email>james.notmyemail@gmail.com</email>
  </employee>
  <employee>
    <name>Lea</name>
    <email>lea.dontemailme@gmail.com</email>
  </employee>
</employees>
```

In this chapter, we will focus on JSON.

Processing JSON payloads

When obtaining a JSON payload, we essentially receive a string. To convert that string into a data object, we have a few options, with the most popular ones being libraries such as **Gson**, **Jackson**, and **Moshi**. In Java, we also have the built-in `org.json` package, while in Kotlin, we have **Kotlin Serialization**. We will be using the **Kotlin Serialization** library. Like Ktor, it was developed by JetBrains. It is very lightweight and well-maintained.

Now that we know what type of data we can commonly expect to use when communicating with network APIs, let's see how we can communicate with these APIs.

Fetching data from a network endpoint

In this section, we will use **The Cat API** (<https://thecatapi.com/>). This RESTful API offers us vast data about, well... cats.

Setting up Ktor and internet permissions

To get started, we will create a new project. To allow our app to make network calls, we have to grant our app the *internet access* install-time permission. We can do this by adding the following code to your `AndroidManifest.xml` file, right before the `Application` tag:

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

Next, we need to set up our app so that it includes Ktor. Ktor helps us generate **Uniform Resource Locators (URLs)**, which are the addresses of the server endpoints we want to access. It also makes decoding JSON payloads into Kotlin data structures easier by providing integration with several parsing libraries. Sending data to the server is also easier with Ktor, as it helps with encoding the requests from Kotlin data structures into JSON.



You can read more about Ktor here: <https://ktor.io/>.

To add Ktor to our project, we need to add the following dependencies to the `build.gradle` file of our app:

```
"io.ktor:ktor-client-core:(latest version)"  
"io.ktor:ktor-client-cio:(latest version)"
```

The first line adds the main client functionality to our project, while the second line adds the CIO engine. Engines are Ktor's mechanism for processing network requests. Different engines allow Ktor to run on different platforms. The CIO engine is a good choice for us because it supports Android, as well as Java virtual machines and native Kotlin.



You can find the latest version of Ktor here: <https://ktor.io>.

With Ktor included in our project, we can set the project up.

Making API requests with Ktor and displaying data

To access an HTTP(S) endpoint, we must start by creating a Ktor `HttpClient` instance. To create a client, we must provide an engine. The code to create a client with the CIO engine looks like this:

```
val client = HttpClient(CIO)
```

The preceding code is quite self-explanatory and produces a client that we can use to make network requests.

In Android, network requests cannot be made from the main thread. This makes sense because waiting for a network request to resolve is an expensive operation, and executing it from the main thread would make an app non-responsive.

Ktor solves this problem by using **coroutines**. This means that network calls using a Ktor client must either be in a suspend function or within a coroutine. For example, to make a call to the `https://api.thecatapi.com/v1/images/search` endpoint, we can use the following code:

```
suspend fun searchImages(limit: Int, size: String):  
String =  
    client.get(  
        "https://api.thecatapi.com/v1/images/search"  
    ) {  
        url {  
            parameter("limit", limit)  
        }  
    }.body()
```

There are a few things to note here. First, you will notice that the `searchImages` function is a suspend function. As we mentioned previously, this is required to make sure we don't block the main thread when making a network call. Next, you will notice we call the `get` function of the client. This performs a GET operation. The `get` function has several overloads. The one we used here takes two arguments: a URL string and a lambda.

The lambda we pass to the `get` function lets us configure the request. In our case, we use a `url` block to add a parameter to the request. There are different ways in which we can transfer data to the API. Adding parameters by calling `parameter`, as we did here, is one such way. It allows us to define values that have been added to the query of our request URL (the optional part of a URL that comes after the question mark). The `parameter` function takes a key and a value. The key is a string, and the value can be any value. Under the hood, a `null` value will be ignored, and `toString()` will be called on any other value before it is added to the query string. Any non-null value passed to `parameter` will be URL-encoded for us. In our case, we added a limit to the query string.

Ktor also provides functions for all other HTTP operations, including `post`, `put`, `patch`, and `delete`. These are all convenient extension functions that can be used on top of `request`, which allows you to set the request method explicitly. By providing a lambda to these functions, we can configure the request URL, method, headers, and body.

Going back to our example, our final call is to `body`. This function returns the server response and returns a value of a generic type. To keep things simple at this stage, we accepted the response as a string. We did this by declaring that the return type of `searchImage` be a string and relying on Kotlin's ability to infer the type for `body`.

So, where should we implement the Ktor code? In both clean architecture and Google's architecture (note that the two are not the same), data is provided by repositories. Repositories, in turn, contain data sources. One such data source could be a network data source. This is where we will implement our network calls. Our `ViewModel` objects will then request data from repositories via use case classes.



In the case of **Model-View-ViewModel (MVVM)**, `ViewModel` is an abstraction of the view that exposes properties and commands.

For our implementation, we will simplify the process by instantiating the Ktor client and making the `get` call in the `Activity` class. This is not a good practice. *Do not do this in a production app.* It does not scale well and is very difficult to test. Instead, adopt an architecture that decouples your views from your business logic and data. See *Chapter 14, Architecture Patterns*, for some ideas.

In the following exercise, you will practice what you’ve learned in this section by reading data from an API and presenting it on screen.

Exercise 5.1 – reading data from an API

In the following chapters, we will develop an app for an imaginary secret agency with a worldwide network of agents saving the world from countless dangers. The secret agency in question is quite unique: it utilizes secret cat agents.

In this exercise, we will create an app that presents us with one random secret cat agent from The Cat API. Before you can present data from an API to your user, you must fetch that data. So, let’s start with that:

1. Create a new **Empty Activity** project (**File** | **New** | **New Project** | **Empty Activity**). Then, click **Next**.
2. Name your application `Cat Agent Profile`.
3. Make sure your package name is `com.example.catagentprofile`.
4. Set **Save location** to where you want to save your project.
5. Leave everything else at its default values and click **Finish**.
6. Make sure you are on the **Android** view in the **Project** pane:

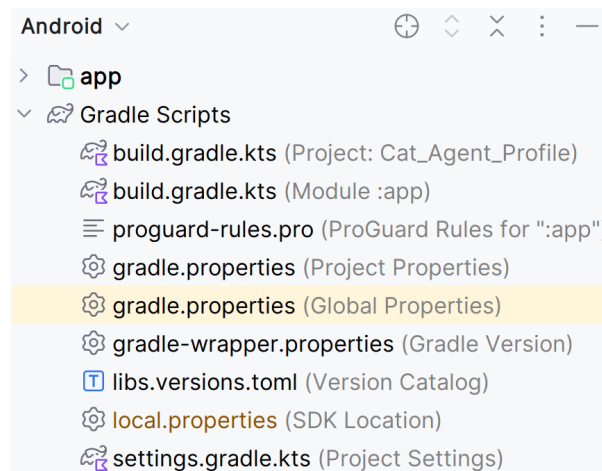


Figure 5.1 – The Android view in the Project pane

7. Open your `AndroidManifest.xml` file. Add internet permissions to your app, like so:

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



```
<application ...> ... </application>
</manifest>
```

8. To add Ktor and the CIO engine to your app, start by adding both to the `libs.versions.toml` file:

```
[versions]
...
ktor = "(latest version)"

[libraries]
...
ktor = { group = "io.ktor", name = "ktor-client-core",
        version.ref = "ktor" }
ktor-cio = { group = "io.ktor", name = "ktor-client-cio",
            version.ref = "ktor" }
```

9. Click the **Sync Project with Gradle Files** button in Android Studio.
10. Open the app module's `build.gradle.kts` file (**Gradle Scripts** | **build.gradle.kts (Module: app)**). Add the following lines anywhere inside the dependencies block:

```
dependencies {
    ...
    implementation(libs.ktor)
    implementation(libs.ktor.cio)
    ...
}
```

11. Click the **Sync Project with Gradle Files** button again.
12. Open your `MainActivity.kt` file.
13. Rename the Greeting composable to `ServerResponse`.
14. Update `ServerResponse` by renaming the `name` parameter to `contents` and setting the text value of the `Text` composable to `contents`. Your composable should look like this:

```
@Composable
fun ServerResponse(contents: String, ...) {
    Text(
        text = contents,
        modifier = modifier
    )
}
```

```
    )  
}
```

15. At the top of the MainActivity class block, add the following:

```
class MainActivity : AppCompatActivity() {  
    private val client = HttpClient(CIO)  
    ...  
}
```

This will instantiate a Ktor HttpClient object with the CIO engine.

16. In the onCreate function, update the top of the setContent block by adding the following code:

```
setContent {  
    var serverResponse by remember {  
        mutableStateOf("Loading...")  
    }  
    ...  
}
```

This will declare a mutable state variable. In the next step, we will load the server response's contents into it. For now, its value is "Loading...".

17. To fetch data from the API, add the following function to the end of the MainActivity class:

```
private suspend fun searchImages(  
    limit: Int  
): String = client.get(  
    "https://api.thecatapi.com/v1/images/search"  
) {  
    url {  
        parameter("limit", limit)  
    }  
}.body()
```

This function takes an integer for the maximal number of results to return. It calls the get convenience extension function on the client, passing in the URL as well as a parameter for the number of results to return.

18. Now, add a `LaunchedEffect` block right after the `serverResponse` declaration to execute `searchImages`:

```
LaunchedEffect(true) {
    withContext(Dispatchers.IO) {
        serverResponse = try {
            searchImages(1)
        } catch (exception: Exception) {
            "Error: $exception"
        }
    }
}
```

This code will fire off the search request every time the activity is composed and handle the possible outcomes—a successful response or any thrown exception.

19. Add all missing imports.
20. Run your app by clicking the **Run ‘app’** button or pressing `Ctrl + R`. On the emulator, it should look like this:

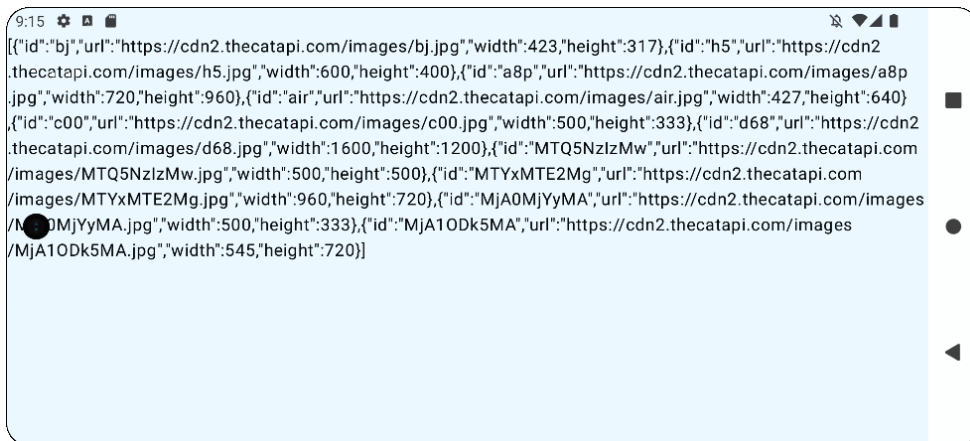


Figure 5.2 – The app presenting the server response JSON

Since every time you run your app, a new call is made and a random response is returned, your result will likely differ. However, whatever your result, if successful, it should be a JSON payload. Next, we will learn how to parse that JSON payload and extract the data we want from it.

Parsing a JSON response

Now that we have successfully retrieved a JSON response from an API, it is time to learn how to use the data we have obtained. To do so, we need to parse the JSON payload. This is because the payload is a plain string representing the data object, and we are interested in the specific properties of that object. If you look closely at *Figure 5.2*, you may notice that the JSON contains a unique identifier for each result, an image URL, and image dimensions. For our code to use this information, we must extract it.

As mentioned in the introduction, multiple libraries exist that will parse a JSON payload for us. Some of the most popular ones are Google's **Gson** (<https://packt.link/uFRVJ>) and Square's **Moshi** (<https://packt.link/LpZ9N>). Recently, Kotlin Serialization (<https://packt.link/kbYmZ>) has been gaining traction. The main reasons for this trend are its support of Kotlin Multiplatform and its lightweight nature. Due to this, we will use Kotlin Serialization in this book.

What do JSON libraries do? Basically, they help us convert data classes into JSON strings (serialization) and vice versa (deserialization). This helps us communicate with servers that understand JSON strings while allowing us to use meaningful data structures in our code.

Configuring Kotlin Serialization with Ktor

To use Kotlin Serialization in our project, we need to include it. We can do this by adding the following line to the plugins block of our app's `build.gradle.kts` file:

```
plugins {  
    ...  
    kotlin("plugin.serialization") version "(kotlin version)"  
}
```

With Kotlin Serialization set up, Ktor can be configured to automatically deserialize JSON strings. To do that, we must include the following dependencies in our catalog file and add them to the dependencies block of the app's `build.gradle.kts` file:

- `io.ktor:ktor-client-content-negotiation:(ktor_version)`: This dependency negotiates the type of content between the client and the server. It is also responsible for serializing and deserializing that content.
- `io.ktor:ktor-serialization-kotlinx-json:(ktor version)`: This dependency serializes and deserializes content to and from JSON strings.

To leverage the newly added dependencies, we need to update our `HttpClient` instantiation by installing the `JSON ContentNegotiation Ktor` plugin:

```
private val client = HttpClient(CIO) {  
    install(ContentNegotiation) {  
        json()  
    }  
}
```

The `ContentNegotiation` plugin is responsible for two tasks:

- Negotiating media types with the server. We will not expand on this.
- Serializing and deserializing requests and responses. This is the responsibility that we care about right now.

To tell the `ContentNegotiation` plugin which serialization library to use, we called `json()`, which installs the Kotlin Serialization library.

Defining data models for JSON mapping

Next, we need to create a data class into which we will map the server JSON response. One convention is to suffix the names of API response data classes with `Data`, so we'll call our data class `UserData`. Other common suffixes are `Entity` and `ApiModel`. Data classes used in serialization or deserialization using Kotlin Serialization must be annotated with `@Serializable`.

When we design our server response data classes, we need to take two factors into account:

- **The structure of the JSON response:** This will affect our data types and field names
- **Our data requirements:** This will allow us to omit fields we do not currently need

Some JSON libraries ignore data in fields that we have not defined in our data classes. For Kotlin Serialization to behave this way when using Ktor, we need to configure it:

```
install(ContentNegotiation) {  
    json(Json { ignoreUnknownKeys = true })  
}
```

One more thing JSON libraries do for us is automatically map JSON data to fields if they happen to have the same name. While this is a nice feature, it carries risk. If we rely solely on it, our data classes (and the code accessing them) will be tightly coupled to the API naming convention.

Because not all APIs are designed well, you might end up with meaningless field names, such as `fn` or `last`, or inconsistent naming. Luckily, there is a solution to this problem: Kotlin Serialization provides us with the `@SerializedName` annotation. It can be used to map a JSON field name to a meaningful field name:

```
@Serializable
data class UserData(
    @SerializedName("fn") val firstName: String,
    @SerializedName("last") val lastName: String = ""
)
```

It is generally better practice to include the annotation, even when the API name is the same as the field name. When obfuscating our code, using the `@SerializedName` annotation protects against serialization errors. It also decouples our code from naming choices made by the API. Note that for `lastName`, we provided a default value of an empty string. This value will be used if the `last` field is missing from the JSON data.

While we are not always lucky enough to have properly documented APIs, when we do, it is best to consult the documentation when designing our model. Our model would be a data class into which the JSON data from all calls we make will be decoded. The documentation for the image search endpoint of The Cat API can be found at developers.thecatapi.com.

You will often find that the documentation is inaccurate. If this happens to be the case, the best thing you can do is contact the owners of the API and request that they update the documentation, though unfortunately, you may have to resort to experimenting with an endpoint. This is risky because undocumented fields or structures are not guaranteed to remain the same, so when possible, try and get the documentation updated.

Based on the response schema obtained from the preceding link, we can define our model as follows:

```
@Serializable
data class ImageResultData(
    @SerializedName("url") val imageUrl: String,
    @SerializedName("breeds") val breeds: List<CatBreedData> = emptyList()
)

@Serializable
data class CatBreedData(
    @SerializedName("name") val name: String,
```

```

    @SerialName("temperament") val temperament: String
)

```

Note that the response structure is that of a list of results. This means we need our responses to be mapped to `List<ImageResultData>`, not simply `ImageResultData`. Now, we need to update the `searchImages` function so that it returns `List<ImageResultData>` instead of `String`:

```
private suspend fun searchImages(limit: Int): List<ImageResultData> =
```

Now that you know how to extract data from a JSON string, let's apply this knowledge to your project. This will be the subject of the following exercise.

Exercise 5.2 – extracting the image URL from the API response

Now that we have a server response as a string, we want to extract the image URL from that string and present only that URL on the screen:

1. Open the `libs.versions.toml` file and add the content negotiation and Kotlin Serialization libraries:

```

[libraries]
...
ktor-client-content-negotiation = { group = "io.ktor",
    name = "ktor-client-content-negotiation", version.ref = "ktor" }
ktor-serialization-kotlinx-json = { group = "io.ktor",
    name = "ktor-serialization-kotlinx-json", version.ref = "ktor" }

```

2. Add both libraries to the dependencies block of the app's `build.gradle.kts` file:

```

implementation(libs.ktor.client.content.negotiation)
implementation(libs.ktor.serialization.kotlinx.json)

```

3. At the top of the same file, add the serialization plugin:

```

plugins {
    ...
    kotlin("plugin.serialization") version "(Kotlin version)"
}

```

4. Click the **Sync Project with Gradle Files** button.
5. Under your app package (`com.example.catagentprofile`), create a model package.

6. Within the `com.example.catagentprofile.model` package, create a new Kotlin data class file named `CatBreedData`.
7. Populate the newly created file with the following code:

```
package com.example.catagentprofile.model

import ...

@Serializable
data class CatBreedData(
    @SerializedName("name") val name: String,
    @SerializedName("temperament") val temperament: String
)
```

8. Next, create `ImageResultData` as another data class file under the same package.
9. Set its contents to the following:

```
package com.example.catagentprofile.model

import ...

@Serializable
data class ImageResultData(
    @SerializedName("url") val imageUrl: String,
    @SerializedName("breeds") val breeds:
        List<CatBreedData> = emptyList()
)
```

10. Lastly, open `MainActivity`.
11. Update the `HttpClient` initialization block to install the `ContentNegotiation` plugin to deserialize JSON:

```
private val client = HttpClient(CIO) {
    install(ContentNegotiation) {
        json()
    }
}
```


12. Update the `searchImages` function so that it returns the `List<ImageResultData>` responses:

```
private suspend fun searchImages(limit: Int):  
List<ImageResultData> =  
    client.get(  
        "https://api.thecatapi.com/v1/images/search"  
    ) {  
        url {  
            parameter("limit", limit)  
        }  
    }.body()
```

13. Now, you need to check for a successful response and that there is at least one `ImageResultData` instance. Once you've done that, you can read the `imageUrl` property of that instance and present it to the user.
14. Run your app. It should look something like this:



Figure 5.3 – The app presenting the parsed image URL

Again, due to the random nature of the API responses, your URL will likely be different.

With that, you have successfully extracted a specific property from an API response. Next, we will learn how to load the image from the URL provided to us by the API.

Loading images from a remote URL

We just learned how to extract data from an API response. That data often includes URLs to images we want to present to the user. There is quite a bit of work involved in achieving that. First, you must fetch the image as a binary stream from the URL. Then, you need to transform that binary stream into an image (it could be a GIF, JPEG, or one of a few other image formats).

Then, you need to convert it into a bitmap instance, potentially resizing it to use less memory. You may also want to apply other transformations to it at that point. Then, you need to set it to a composable.

Sounds like a lot of work, doesn't it? Well, luckily for us, there are a few libraries that do all of that (and more) for us. The most commonly used library is **Glide** by Bump Technologies (<https://packt.link/sPJNB>). Facebook's **Fresco** (<https://frescolib.org/>) is somewhat less popular. A library that's gained traction recently is **Coil** (<https://coil-kt.github.io/coil/>).

We will proceed with Coil because it is Kotlin-first and integrates well with Compose. However, it's worth noting that the other libraries are great as well and worth knowing.

To include Coil in your project, add its Compose dependency: `io.coil-kt.coil3:coil-compose:(latest version)`. Coil supports different networking libraries, but the easiest one, which requires the least amount of additional setup, is `okHttp`. To use it, add the `io.coil-kt.coil3:coil-network-okhttp:(latest version)` dependency.



To find the latest version of Coil, visit <https://coil-kt.github.io/coil/>.

Because we might change our minds at a later point, this is a great opportunity to abstract away the concrete library so that we have a simple interface of our own. So, let's start by defining our `LoadedImage` composable:

```
@Composable
fun LoadedImage(imageUrl: String, modifier: Modifier =
Modifier) {
    AsyncImage(
        model = imageUrl,
        contentDescription = null,
        modifier = modifier
```

```
)  
}
```

This is a naive implementation. In a production implementation, you might want to add arguments (or multiple composables) to support options such as different cropping strategies or loading states.

Let's add a `LoadedImage` composable to our screen, replacing the `ServerResponse` one:

```
CatAgentProfileTheme {  
    Scaffold(modifier = Modifier.fillMaxSize()) {  
        innerPadding ->  
            LoadedImage(  
                imageUrl = serverResponse,  
                modifier = Modifier.padding(innerPadding)  
            )  
        }  
    }  
}
```

If we run our app now, we should see something like the following:

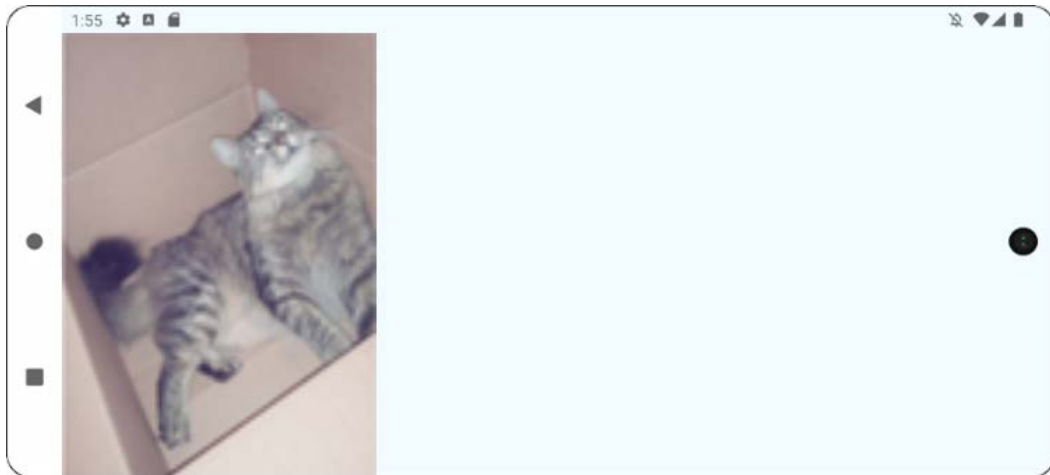


Figure 5.4 – Image loaded from the server response image URL

Remember that the API returns random results, so the actual image is likely to be different. Note that Coil does not support GIF out of the box and that GIF or WebP animations require a bit more work to support. I'll leave that as an exercise for you.



WebP is an efficient image format developed by Google. You can read more about WebP here: <https://packt.link/7wI56>.

Exercise 5.3 – loading the image from the obtained URL

In the previous exercise, we extracted the image URL from the API response. Now, we will use that URL to fetch an image from the web and display it in our app:

1. Add Coil and its Compose extension to your `libs.versions.toml` file:

```
[versions]
...
coil = "(latest version)"

[libraries]
...
coil = { group = "io.coil-kt.coil3", name = "coil-compose",
        version.ref = "coil" }
coil-network = { group = "io.coil-kt.coil3",
                name = "coil-network-okhttp", version.ref = "coil" }
```

2. Open the app's `build.gradle.kts` file and add the Coil dependencies:

```
dependencies {
    ...
    implementation(libs.coil)
    implementation(libs.coil.network)
    ...
}
```

3. Synchronize your project with the Gradle files.
4. On the left **Project** panel, right-click on your project package name (`com.example.catagentprofile`) and select **New | Kotlin File/Class**.
5. Type `LoadedImage` in the **Name** field. Then, select **File**.

6. Open the newly created `LoadedImage.kt` file and update it, like so:

```
package com.example.catagentprofile

import ...

@Composable
fun LoadedImage(imageUrl: String, modifier: Modifier = Modifier) {
    AsyncImage(
        model = imageUrl,
        contentDescription = null,
        modifier = modifier
    )
}
```

This will be your composable for loading images in the app.

7. Open the `MainActivity.kt` file.
8. Update your Compose content, replacing the `ServerResponse` composable with a `LoadedImage` one:

```
CatAgentProfileTheme {
    Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding ->
        LoadedImage(
            imageUrl = serverResponse,
            modifier =
                Modifier.padding(innerPadding)
        )
    }
}
```

Now, once you have a non-blank URL, it will be loaded into `LoadedImage`.

9. Run the app:

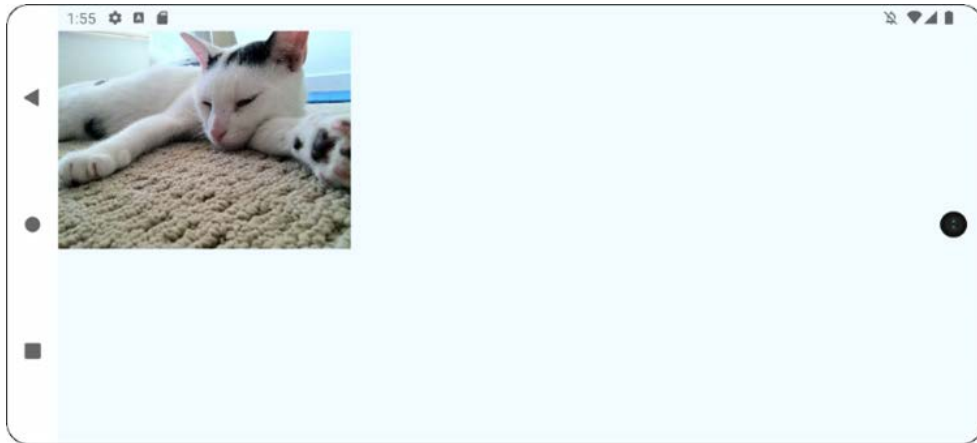


Figure 5.5 – Exercise outcome showing a random image

In this section, we learned how to fetch data from a remote API. Then, we learned how to process that data and extract the information we needed from it. Lastly, we learned how to present an image on the screen when given an image URL.

In the following activity, we will apply our knowledge to develop an app that tells the user the current weather in a specific location, presenting the user with a relevant weather icon.

Activity 5.1 – displaying the current weather

Let's say we want to build an app that shows the current weather in a certain place in the world. Furthermore, we also want to display an icon representing the current weather there.

This activity aims to create an app that polls an API endpoint for the current weather in JSON format, transforms that data into a local model, and uses that model to present the current weather. It also extracts the URL to an icon representing the current weather and fetches that icon so that it can be displayed on the screen.

We will use the free OpenWeatherMap API for this activity. The documentation can be found at <https://packt.link/2kwd7>. To sign up for an API token, please go to <https://packt.link/Yqz3V>. You can find your keys and generate new ones as needed at <https://packt.link/wZJnb>.

The steps for this activity are as follows:

1. Create a new app.
2. Grant internet permissions to the app in order to be able to make API and image requests.
3. Add the Kotlin Serialization plugin.
4. Add Ktor, the Ktor CIO engine, the ContentNegotiation plugin, the Kotlin Serialization Ktor plugin, Coil, and the Coil Kotlin extension dependencies to the app.
5. Declare a `LoadedImage` composable that wraps around a Coil composable.
6. Update the app composable so that it can present the weather in a textual form (a short and long description), as well as a weather icon image.
7. Define the model. Create classes that will contain the server response.
8. Add a Ktor `HttpClient` object to your main activity.
9. Install the JSON ContentNegotiation plugin. Remember to set it to ignore unknown keys.
10. Add a function to read the weather from the OpenWeatherMap API at <https://packt.link/27TS5>. Remember to add the `appid` parameter, as well as the desired `lat` and `lon` geocoordinates. Also, remember to make the function a suspend function.



The documentation for the current weather API can be found here: <https://packt.link/09pt6>.

To find the coordinates for your desired location, go to <https://packt.link/S1KLj>.

The icons can be fetched using the `icon` value from the API response: <https://packt.link/ExpZa>.

11. Call the function you created from your main activity every time it is composed. Store the result in a variable using `remember`.
12. Handle the successful server response.
13. Handle exceptions that are thrown when trying to fetch the weather.

The expected output is shown here:



Figure 5.6 – The final weather app



The solution for this activity can be found at <https://packt.link/aImqB>.

Summary

In this chapter, we learned how to fetch data from an API using Ktor. Then, we learned how to handle JSON responses, as well as plain text responses, using Kotlin Serialization. We also saw how different error scenarios could be handled.

Later, we learned how to load images from URLs using Coil and how to present them to the user via composables.

There are quite a few popular libraries for fetching data from APIs and loading images. We only covered some of the most popular ones. You might want to try out some of the other libraries to find out which ones fit your purposes best.

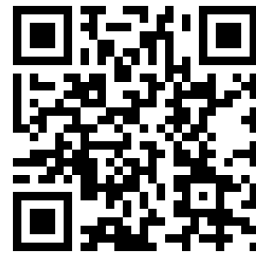
In the next chapter, we will introduce Compose lazy lists, powerful UI components that we can use to present our users with lists of items.

Unlock this book's exclusive benefits now

Scan this QR code or go to packtpub.com/unlock, then search this book by name.

Note: Keep your purchase invoice ready before you start.

UNLOCK NOW



6

Building Lists with Jetpack Compose

In the previous chapter, we learned how to fetch data (including lists of items and image URLs) from APIs and how to load images from URLs. Combining that knowledge with the ability to display lists of items allows us to implement rich user interfaces and is the goal of this chapter.

Quite often, you will want to present your users with a list of items. For example, you might want to show them a list of pictures on their device or let them select their country from a list of all countries. To do that, you would need to populate multiple views, all sharing the same layout but presenting different content.

In this chapter, you will learn how to add lists and grids of items to your apps and effectively leverage the recycling power of **lazy lists**. You'll also learn how to handle user interaction with the item views on the screen and support different item view types, such as for titles. Later in the chapter, you'll add and remove items dynamically.

In previous editions of this book, we showed how this was achievable using the `RecyclerView` component. In this edition, because of the growing popularity of Jetpack Compose, we will use lazy lists. Lazy lists, such as `LazyColumn` and `LazyRow`, offer greater functionality than regular lists such as `Row` and `Column`. Primarily, they support large datasets well.

So, what does a lazy list do? Lazy lists optimize the presentation of large numbers of items. Imagine having hundreds of items to show the user. Using regular lists, we would have to store composables for all items in memory. This will have a considerable negative impact on performance.

Lazy lists rely on `LazyListScope` blocks to manage their contents. This is unlike regular lists, which expect composable content. The `LazyListScope`-extending blocks expose a **domain-specific language (DSL)** for describing the contents of individual items.

To learn about lazy lists, in this chapter, we will develop an app that lists secret agents and whether they are currently active or sleeping (and, thus, unavailable). The app will then allow us to add new agents or delete existing ones by swiping them away. There is a twist, though – as you saw in *Chapter 5, Essential Libraries – Ktor, Kotlin Serialization, and Coil*, all our agents will be cats.

We will cover the following topics:

- Adding a lazy list to our layout
- Populating a `LazyColumn` composable
- Responding to clicks in `LazyColumn` composables
- Supporting different item types
- Swiping to remove items
- Adding items interactively

By the end of the chapter, you will have the skills required to present your users with interactive lists of rich items.

Technical requirements

The complete code for all the exercises and the activity in this chapter is available on GitHub at <https://packt.link/a011P>.

Adding a lazy list to our layout

In *Chapter 3, Developing the UI with Jetpack Compose*, we saw how we can add composables to our `Activity` class to be rendered on the screen. Lazy lists are just such composables. In this chapter, we will focus on `LazyColumn`, but the same principles apply when using `LazyRow`, `LazyVerticalGrid`, and `LazyHorizontalGrid` composables. To add a `LazyColumn` composable to our layout, we need to add the following tag to our containing composable:

```
LazyColumn (tag -> composable) {  
    }
```

The appearance and behavior of a `LazyColumn` composable can be altered by passing different arguments to the `LazyColumn` composable function. You can explore the source code of `LazyColumn` to discover the different options.



You can read more about `LazyColumn` and `LazyRow` (its horizontal equivalent) here: <https://packt.link/Qd2Zi>.

It is hard to think of a scenario where we would not want our lazy list to be scrollable, since its main value is in its ability to optimize elements that are off-screen. To support scrolling, we must remember its state:

```
val columnState = rememberLazyListState()

LazyColumn(state = columnState) {
}
```

Adding `LazyColumn` to our layout means we now have an empty container to hold the child composables representing our items. Once populated, it will handle the efficient presentation and scrolling of child composables for us.

Exercise 6.01 – Adding an empty `LazyColumn` to your main activity

To use a `LazyColumn` composable in your app, you first need to add it to one of your composables. Let's add it to the `Greeting` composable in your main activity:

1. Start by creating a new empty activity project (**File | New | New Project | Empty Activity**). Name your application `Cat Deployer`. Make sure your package name is `com.example.catdeployer`.
2. Set the **Save** location to where you want to save your project. Leave everything else at their default values and click **Finish**. Make sure you are on the **Android** view in your **Project** pane, as shown in *Figure 6.1*:

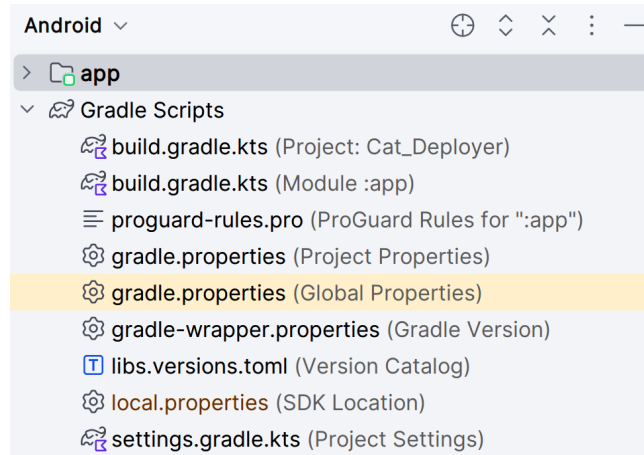


Figure 6.1 – The Android view in the Project pane

3. Open the MainActivity.kt file.
4. Inside the Greeting composable, replace the Text composable with an empty LazyColumn composable:

```
@Composable
fun Greeting(
    name: String, modifier: Modifier = Modifier
) {
    LazyColumn(modifier = modifier) {
    }
}
```

5. Add a variable named columnState to remember the state of the LazyColumn composable you just added, enabling future scrolling:

```
fun Greeting(
    name: String, modifier: Modifier = Modifier
) {
    val columnState = rememberLazyListState()

    LazyColumn(
        state = columnState, modifier = modifier
    ) {
    }
}
```

6. Run your app by clicking the **Run app** button or pressing *Ctrl + R* (*Shift + F10* in Windows). On the emulator, it should look like *Figure 6.2*:

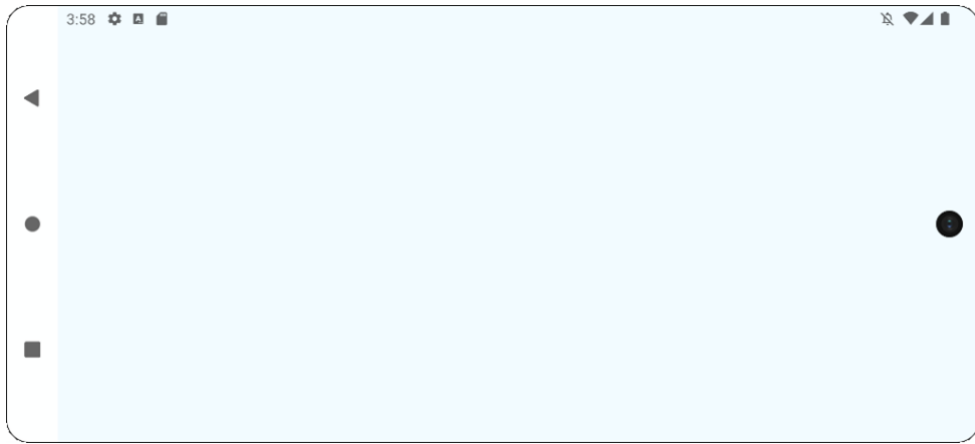


Figure 6.2 – The app with an empty RecyclerView

As you can see, our app runs and our composable is presented on the screen. However, we do not see our `LazyColumn` composable. Why is that? At this stage, our `LazyColumn` composable has no content. Lazy lists with no content are not rendered by default, so while our `LazyColumn` composable is indeed on the screen, it is not visible. This brings us to the next step – populating a `LazyColumn` composable with content that we can actually see.

Populating a `LazyColumn` composable

So, we added a `LazyColumn` composable to our screen. For us to benefit from the `LazyColumn` composable, we need to add content to it. Let's see how we go about doing that.

As we mentioned before, to add content to our `LazyColumn` composable, we would need to populate its `LazyListScope`-extending block. In this block, we tell the `LazyColumn` composable how to plug data into composables designed to present that data.

For example, let's say we want to present a list of employees. Here are the steps that we'll take:

1. We need to design our UI model. This will be a data class holding all the information our app needs for it to present a single employee. Because this is a UI model, one convention is to suffix its name with `UiModel`:

```
data class EmployeeUiModel(  
    val name: String,  
    val role: EmployeeRole, val gender: Gender,
```

```
        val imageUrl: String
    )
}
```

We will define `EmployeeRole` and `Gender` as follows:

```
enum class EmployeeRole(val label: String) {
    HUMAN_RESOURCES("Human resources"),
    MANAGEMENT("Management"),
    TECHNOLOGY("Technology")
}

enum class Gender {
    FEMALE,
    MALE,
    UNKNOWN
}
```

The values are provided as an example, of course. Feel free to add more of your own!

You can see the expected model hierarchy in *Figure 6.3*:

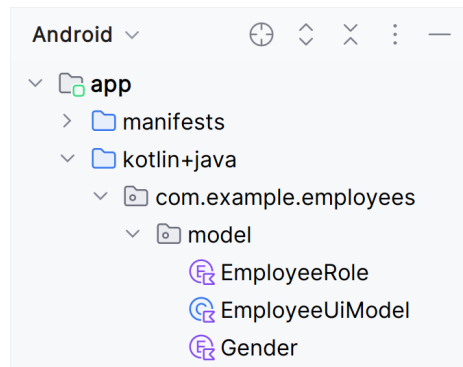


Figure 6.3 – The model hierarchy

- Now we know what data to expect when writing our composable, we can continue to design our composable to present this data, which we'll save as `Employee.kt`. We'll start with a `LoadedImage` composable (check the previous chapter to find out how to implement the `LoadedImage` composable):

```
@Composable
fun Employee(employee: EmployeeUiModel) {
    LoadedImage(
```

```
        imageUrl = employee.imageUrl,
        modifier = Modifier.size(64.dp)
    )
}
```

3. Then, we will add a Text composable for each field. We will use a row and a column to improve appearances, and make the LoadedImage conditional on there an image URL:

```
@Composable
fun Employee(employee: EmployeeUiModel) {
    Row {
        if (employee.imageUrl.isEmpty()) {
            Spacer(modifier = Modifier.size(64.dp))
        } else {
            LoadedImage(
                imageUrl = employee.imageUrl,
                modifier = Modifier.size(64.dp)
            )
        }
        Column {
            Text(text = employee.name)
            Text(text = employee.role.label)
        }
    }
}
```

So far, there is nothing new. You should be able to recognize all of the different composables from the previous chapters. *Figure 6.4* shows us a preview of the Employee composable:

Preview

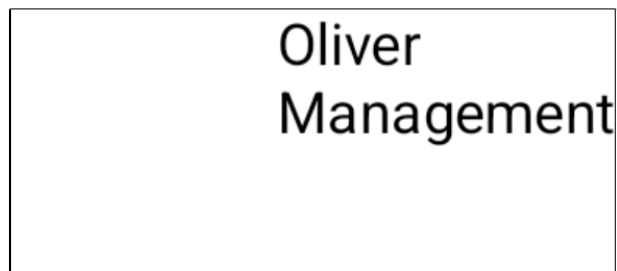


Figure 6.4 – A preview of the Employee composable

4. With a data model and a composable, we now have everything we need to bind our data to the LazyColumn composable. To do that, we will populate the LazyListScope-extending block passed to our LazyColumn composable:

```
@Composable
fun Employees(
    employees: List<EmployeeUiModel>,
    modifier: Modifier = Modifier
) {
    val columnState = rememberLazyListState()

    LazyColumn(state = columnState, modifier = modifier) {
        items(employees.size) { index ->
            Employee(employee = employees[index])
        }
    }
}
```

There are a few things worth noting in the preceding code. First, we replaced the name field with a list of employees. Second, we added an items block inside the LazyColumn block that extends LazyListScope. The items function takes two arguments: the number of items to present and a composable block. Within the block, we have access to the index of the currently presented item. Having the index allows us to access the corresponding data – in our case, it is the *n*th employee from the list of employees. We then pass the employee to an Employee composable.

The complete working code will look like this:

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {
            EmployeesTheme {
                Scaffold(
                    modifier = Modifier.fillMaxSize()
                ) { innerPadding ->
                    Greeting(
                        employees = listOf(
```

```

        EmployeeUiModel(
            "Robert",
            EmployeeRole.MANAGEMENT,
            Gender.MALE,
            "https://images.pexels.com/
            photos/220453/
            pexels-photo-220453.jpeg
            ?h=650&w=940"
        ),
        EmployeeUiModel(
            "Wilma",
            EmployeeRole.TECHNOLOGY,
            Gender.FEMALE,
            "https://images.pexels.com/
            photos/3189024/
            pexels-photo-3189024.jpeg
            ?h=650&w=940"
        ),
        EmployeeUiModel(
            "Curious George",
            EmployeeRole.HUMAN_RESOURCES,
            Gender.UNKNOWN,
            "https://images.pexels.com/
            photos/771742/
            pexels-photo-771742.jpeg
            ?h=750&w=1260"
        )
    ),
    modifier =
        Modifier.padding(innerPadding)
    )
}
}
}
}
}
}
}

```

Running our app now, we will see a list of our employees (*Figure 6.5*):

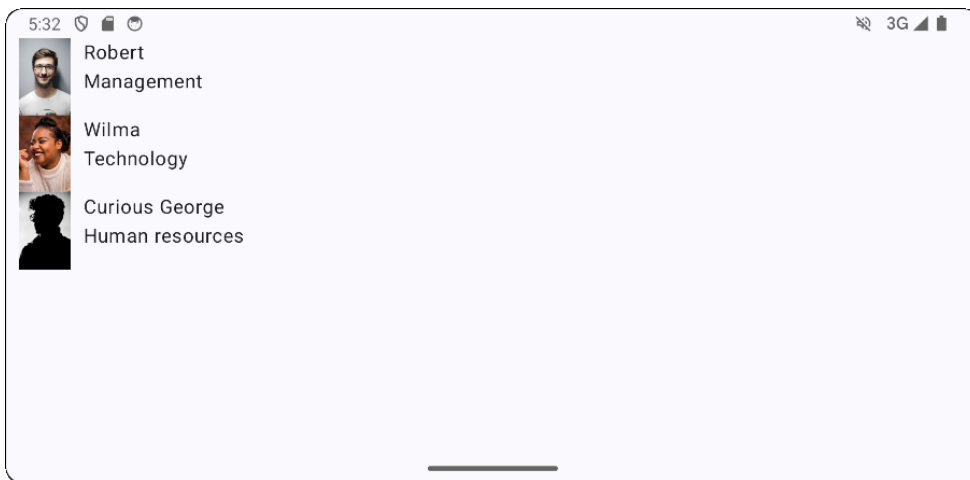


Figure 6.5 – The Employees app

Note that we hardcoded the list of employees. In a production app, following a **Model-View-View-Model (MVVM)** pattern (we will cover this pattern in *Chapter 14, Architecture Patterns*), `ViewModel` would provide data to present.

Exercise 6.02 – Populating your `LazyColumn` composable

A `LazyColumn` composable is not very interesting without any content. It is time to populate your `LazyColumn` composable by adding your secret cat agents to it.

A quick recap before you dive in – in the previous exercise, we introduced an empty list designed to hold a list of secret cat agents that users have at their disposal. In this exercise, you will be populating that list to present the users with the available secret cat agents in the agency:

1. To keep our file structure tidy, we will start by creating a model package. Right-click on the package name of our app, and then select **New | Package** (see *Figure 6.6*).

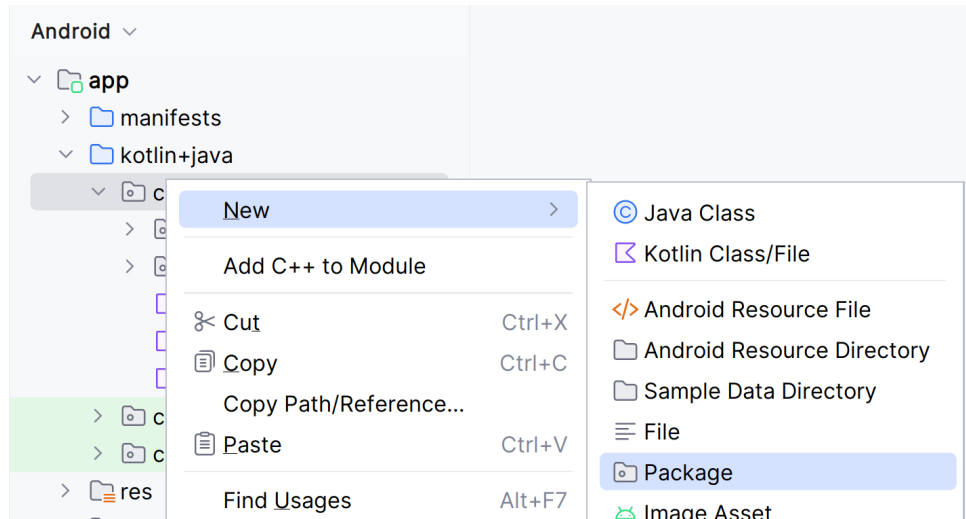


Figure 6.6 – Creating a new package

2. Name the new package `model`. Click **OK** to create the package.
3. To create our first model data class, right-click on the newly created model package and then select **New | Kotlin File/Class**.
4. Under **Name**, type in `CatUiModel`. Double-click the **Data class** option. This will be the class holding the data we have about every individual cat agent.
5. Add the following to the newly created `CatUiModel.kt` file to define the data class with all the relevant properties of a cat agent:

```
data class CatUiModel(
    val gender: Gender,
    val name: String,
    val biography: String,
    val imageUrl: String
)
```

For each cat agent, other than their name and photo, we want to know their gender and biography. This will help us choose the right agent for a mission.

6. Again, right-click on the model package and then navigate to **New | Kotlin File/Class**.
7. This time, name the new file `Gender` and double-click on the **Enum class** option. This class will hold our different cat genders.

8. Update the Gender enum, like so:

```
enum class Gender(val symbol: String) {
    FEMALE("\u2640"),
    MALE("\u2642"),
    UNKNOWN("?")
}
```

9. Introduce Coil to the project by adding the following two libraries:

```
coil = { group = "io.coil-kt", name = "coil", version.ref = "coil" }
coil-compose = { group = "io.coil-kt", name = "coil-compose",
    version.ref = "coil" }
```

10. Synchronize the project with the Gradle files.
11. You will need a copy of `LoadedImage.kt`, introduced in *Chapter 5, Essential Libraries – Ktor, Kotlin Serialization, and Coil*, so right-click on the package name of your app, navigate to **New | Kotlin File/Class**, then set the name to `ImageLoader` and double-click **File**.
12. Similar to the implementation in that chapter, add this code to the file:

```
@Composable
fun LoadedImage(
    imageUrl: String, modifier: Modifier = Modifier
) {
    AsyncImage(
        model = imageUrl,
        contentDescription = null,
        modifier = modifier
    )
}
```

Make sure to import the dependencies that are required.

13. Now, to define the composable holding the data about each cat agent, create a new Kotlin file by right-clicking on the app package and then selecting **New | Kotlin File/Class**.
14. Name your file `Cat`. Double-click on the **File** option.
15. Update the contents of the newly created `Cat.kt` file:

```
@Composable
fun Cat(cat: CatUiModel) {
    Row {
```

```
        if (cat.imageUrl.isEmpty()) {
            Spacer(modifier = Modifier.size(64.dp))
        } else {
            LoadedImage(
                imageUrl = cat.imageUrl,
                modifier = Modifier.size(64.dp)
            )
        }
        Column {
            Text(text = cat.name)
            Text(text = cat.biography)
        }
    }
}
```

This will create a composable with an image along with text fields for a name and biography to be used in our list.

16. Open your MainActivity file. Replace the Greeting composable with CatAgents:

```
@Composable
fun CatAgents(
    cats: List<CatUiModel>,
    modifier: Modifier = Modifier
) {
    val columnState = rememberLazyListState()

    LazyColumn(
        state = columnState, modifier = modifier
    ) {
        items(cats.size) { index ->
            Cat(cat = cats[index])
        }
    }
}
```

The CatAgents composable now populates the LazyColumn composable with a Cat composable for every CatUiModel instance that is passed into CatAgents.

17. Still in MainActivity, replace the call to Greeting with the following:

```
Scaffold(...) { innerPadding ->
    CatAgents(
        cats = listOf(
            CatUiModel(
                Gender.MALE,
                "Fred",
                "Silent and deadly",
                "https://24.media.tumblr.com/
                tumblr_lsln7s1Z8f1qasbyxo1_250.jpg"
            ),
            CatUiModel(
                Gender.FEMALE,
                "Wilma",
                "Cuddly assassin",
                "https://cdn2.thecatapi.com/images/KJF8fB_20.jpg"
            ),
            CatUiModel(
                Gender.UNKNOWN,
                "Curious George",
                "Award winning investigator",
                "https://cdn2.thecatapi.com/images/vJB8rwfdX.jpg"
            )
        ),
        modifier = Modifier.padding(innerPadding)
    )
}
```

18. In your AndroidManifest.xml file, add the following in the manifest tag before the application tag:

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

Having this tag will allow your app to download images from the internet.

19. Run your app. It should look like *Figure 6.7*:

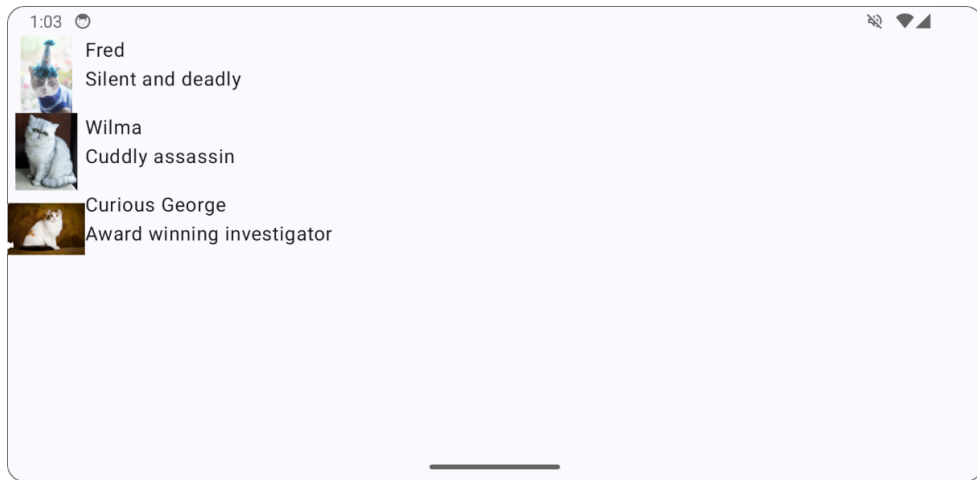


Figure 6.7 – A `LazyColumn` composable with hardcoded secret cat agents

As you can see, the `LazyColumn` composable now has content, and your app is starting to take shape. Note how the same `Cat` composable, which is only mentioned once in the `LazyColumn` composable, is used to present different items based on the data bound to each instance. As you would expect, if you add enough items for them to go off-screen, scrolling works. Next, we'll look into allowing a user to interact with the items inside `LazyColumn`.

Responding to clicks in `LazyColumn` composables

What if we want to let our users select an item from a presented list? To achieve that, we need to communicate clicks back to our app.

The first step in implementing click interaction is to ensure that our item composable takes a lambda. Internally, the composable then uses the `clickable` modifier to identify clicks. When a click is performed, it executes the lambda. Let's see how this looks for the `Employee` composable we discussed earlier, in the *Populating a `LazyColumn` composable* section:

```
@Composable
fun Employee(
    employee: EmployeeUiModel, onClick: () -> Unit
) {
    Row(modifier = Modifier.clickable {
        onClick()
    }) {
```



```

        ...
    }
}

```

Next, we can revisit our container of `LazyColumn` where `Employee` is used:

```

@Composable
fun Employees(
    employees: List<EmployeeUiModel>,
    onEmployeeClick: (Int) -> Unit,
    modifier: Modifier = Modifier
) {
    val columnState = rememberLazyListState()

    LazyColumn(state = columnState, modifier = modifier) {
        items(employees.size) { index ->
            Employee(
                employee = employees[index],
                onClick = { onEmployeeClick(index) }
            )
        }
    }
}

```

To keep our code clean, we renamed `Greeting` to `Employees`. We continued to add an `onEmployeeClick` lambda. Its input parameter is the index of the employee that was clicked. We then passed a lambda to the `onClick` parameter of the `Employee` composable, which calls `onEmployeeClick` with the index associated with that employee.

Finally, to handle the click event and show a toast, we pass a lambda to `Employees` in our activity for `onEmployeeClick`. We set that lambda to show a toast when clicked. In an MVVM implementation, you would be notifying the `ViewModel` instance of the click at this point instead. The `ViewModel` instance would then update its state, telling the view (our activity) that it should display the toast.

Exercise 6.03 – Responding to clicks

Your app already shows the user a list of secret cat agents. It is time to allow your user to choose a secret cat agent by clicking on its view. Click events are delegated from the item composable to the lazy list container to the activity, as shown in *Figure 6.8*:

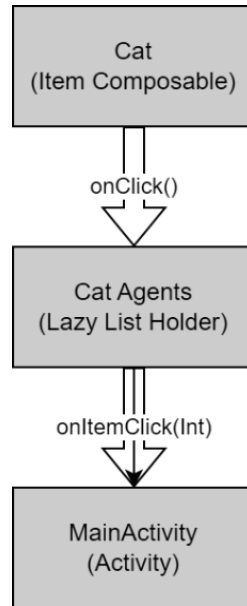


Figure 6.8 – The flow of click events

The following are the steps that you need to follow to complete this exercise:

1. Open your `Cat.kt` file holding your `Cat` composable. Add an `onClick` parameter to it:

```
fun Cat(  
    cat: CatUiModel, onClick: () -> Unit = {}  
) {  
    ...  
}
```

2. This will be the callback you will execute when the user clicks the composable. The default empty lambda makes `onClick` optional, which is useful in previews, for example.
3. Still in the `Cat` composable, add the `clickable` modifier to the `Row` composable:

```
fun Cat(cat: CatUiModel, onClick: () -> Unit = {}) {  
    Row(modifier = Modifier.clickable {  
        onClick()  
    }) {  
        ...  
    }  
}
```

The `clickable` modifier delegates click events to the `onClick` function.

- Next, open the `MainActivity.kt` file. Update your `CatsAgents` composable to accept an `onCatClick` argument, as follows:

```
fun CatAgents(  
    cats: List<CatUiModel>,  
    modifier: Modifier = Modifier,  
    onCatClick: (Int) -> Unit = {}  
) {  
    ...  
}
```

Note that the `onCatClick` callback is added after the modifier. This is because the modifier should be the first optional parameter. Also, note that the `onCatClick` lambda has an `Int` input parameter. This will provide the index of the clicked element.

- For you to be able to reference a cat when given an index, extract the list of cats from the call to `CatAgents`:

```
val cats = listOf(CatUiModel(...), ...)  
  
CatAgents(  
    cats = cats,  
    modifier = Modifier.padding(innerPadding)  
)
```

- Finally, pass a lambda to `CatAgents` that presents a toast when a cat is clicked:

```
CatAgents(  
    cats = cats,  
    modifier = Modifier.padding(innerPadding),  
    onCatClick = { catIndex ->  
        Toast.makeText(  
            context,  
            "${cats[catIndex].name} clicked!",  
            Toast.LENGTH_SHORT  
        ).show()  
    }  
)
```

7. Run your app. Clicking on one of the cats should now open a dialog, as shown in *Figure 6.9*:

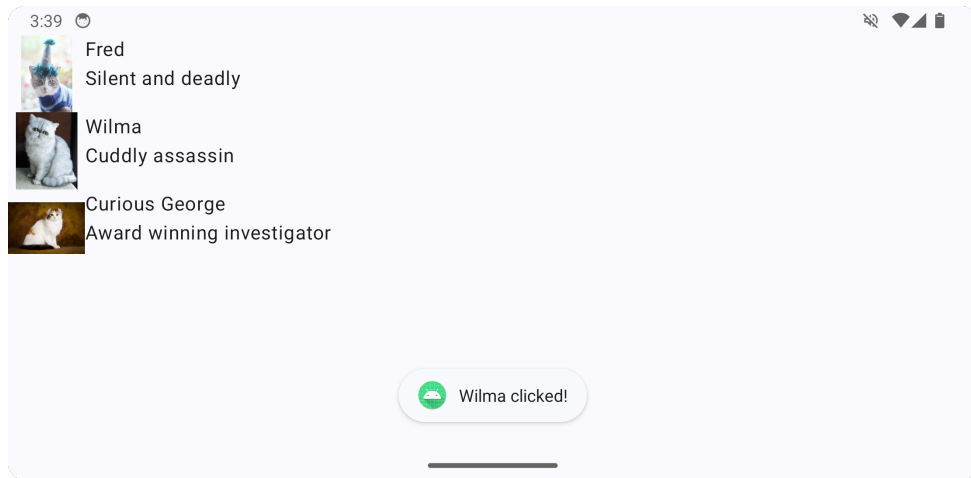


Figure 6.9 – A dialog showing that an agent was selected

Try clicking the different items and note the different messages presented. You now know how to respond to users clicking on items inside your lazy list. Next, we will look at how we can support different item types in our lists.

Supporting different item types

In the previous sections, we learned how to handle a list of items of a single type (in our case, all our items were `EmployeeUiModel`). What happens if you want to support more than one type of item? A good example of this would be having group titles on our list.

Let's say that instead of getting a list of employees, we get a list containing current and past employees. Each of the two groups of employees is preceded by the title of the corresponding group. Instead of a list of `EmployeeUiModel` instances, our list would now contain `ListItemUiModel` instances. `ListItemUiModel` might look like this:

```
sealed interface ListItemUiModel {
    data class GroupTitle(val name: String) : ListItemUiModel
    data class Employee(
        val employee: EmployeeUiModel
    ) : ListItemUiModel
```

We wrap the two object types in a sealed interface for compile-time type safety. A list containing only objects of the `GroupTitle` and `Employee` types can be of the `ListItemUiModel` type, guaranteeing that all its elements are either titles or employees.

Our list of items may look like this:

```
listOf(
    ListItemUiModel.GroupTitle("Current Employees"),
    ListItemUiModel.Employee(
        EmployeeUiModel(...)
    ),
    ListItemUiModel.Employee(
        EmployeeUiModel(...)
    ),
    ListItemUiModel.GroupTitle("Past Employees"),
    ListItemUiModel.Employee(
        EmployeeUiModel(...)
    ),
    ListItemUiModel.Employee(
        EmployeeUiModel(...)
    )
)
```

In this case, having just one layout type will not do. Luckily, we can use code inside the `items` block, which allows us to identify each item type and provide the appropriate composable for it. In our `items` block, instead of providing one composable, we can implement a conditional statement to identify which composable should be used for each item, as in the following example:

```
items(listItems.size) { index ->
    when (val item = listItems[index]) {
        is ListItemUiModel.GroupTitle -> {
            Text(item.name)
        }
        is ListItemUiModel.Employee -> {
            Employee(item.employee)
        }
    }
}
```

This implementation maps the item type at a given position to a composable. In our case, we know about titles and employees, thus we have two checks in our `when` statement. For our group title, a simple `Text` composable may be sufficient. For an employee, the `Employee` composable can be used. Kotlin's smart casting makes it easy for us to access the relevant fields of each type.

Having made those changes, we can now expect to see the Current Employees and Past Employees group titles, each followed by the relevant employees.

Exercise 6.04 – Adding titles to lazy lists

We now want to be able to present our secret cat agents in two groups – **active agents** that are available for us to deploy to the field, and **sleeper agents** that cannot currently be deployed. We will do that by adding a title above the active agents and another above the sleeper agents:

1. Under `com.example.catdeployer.model`, create a new Kotlin file called `ListItemUiModel`.
2. Add the following to the `ListItemUiModel.kt` file, defining our two data types – titles and cats:

```
sealed interface ListItemUiModel {  
    data class Title(val title: String) :  
        ListItemUiModel  
    data class Cat(val cat: CatUiModel) :  
        ListItemUiModel  
}
```

3. In `MainActivity.kt`, replace the `CatAgents` list parameter (`cats`) with `listItems: List<ListItemUiModel>` and update the contents of the items block within `CatAgents`:

```
items(listItems.size) { index ->  
    when (val item = listItems[index]) {  
        is ListItemUiModel.Title -> {  
            Text(item.title)  
        }  
  
        is ListItemUiModel.Cat -> {  
            Cat(  
                cat = item.cat,  
                onClick = { onItemClick(index) }  
            )  
        }  
    }  
}
```

4. Instead of adding a `Cat` composable for every item provided, `CatAgents` will now check the type of each item. If it is a `Title` type, it will add a `Text` composable instead of a `Cat` one:

```
items(listItems.size) { index ->
    when (val item = listItems[index]) {
        is ListItemUiModel.Title -> {
            Text(item.title)
        }

        is ListItemUiModel.Cat -> {
            Cat(
                cat = item.cat,
                onClick = { onItemClick(index) }
            )
        }
    }
}
```

5. Update `MainActivity` to provide `CatAgents` with appropriate data, replacing the previous list of `CatUiModel` with the following:

```
val listItems = listOf(
    ListItemUiModel.Title("Sleeper Agents"),
    ListItemUiModel.Cat(
        CatUiModel(
            Gender.MALE,
            "Fred",
            "Silent and deadly",
            "https://24.media.tumblr.com/
            tumblr_lsl7s1Z8f1qasbyxo1_250.jpg"
        )
    ),
    ListItemUiModel.Cat(
        CatUiModel(
            Gender.FEMALE,
            "Wilma",
            "Cuddly assassin",
            "https://cdn2.thecatapi.com/images/KJF8fB_20.jpg"
        )
    )
)
```

```

    ),
    ListItemUiModel.Title("Active Agents"),
    ListItemUiModel.Cat(
        CatUiModel(
            Gender.UNKNOWN,
            "Curious George",
            "Award winning investigator",
            "https://cdn2.thecatapi.com/images/vJB8rwfdX.jpg"
        )
    )
)
)

```

6. The changes you should make in the code are as follows: rename `cats` to `listItems`, add `ListItemUiModel.Title` instances, and wrap every `CatUiModel` constructor call with a `ListItemUiModel.Cat` one.
7. Run the app. You should see something similar to *Figure 6.10*:

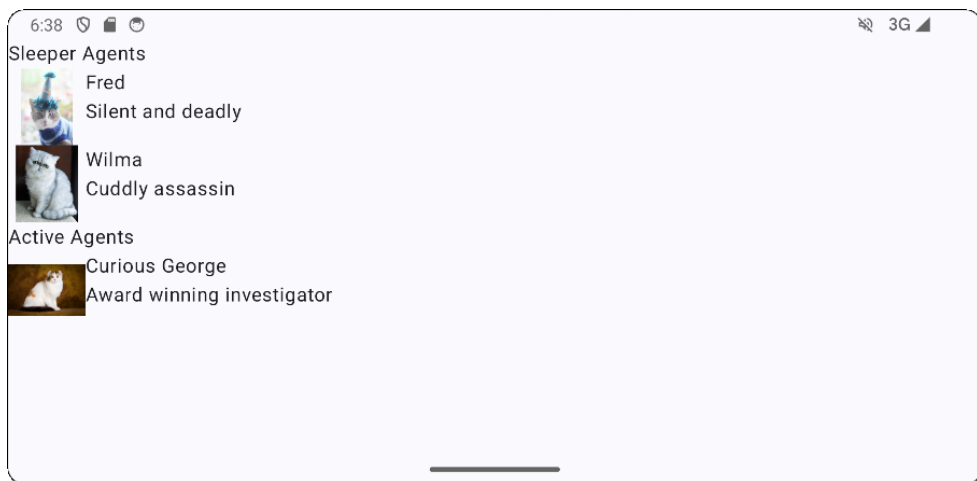


Figure 6.10 – Lazy list with the *Sleeper Agents* and *Active Agents* header views

As you can see, we now have titles above our two agent groups. These titles will scroll with our content. Next, we will learn how to swipe an item to remove it from a lazy list.

Swiping to remove items

In the previous sections, we learned how to present different composables for different data types. However, up until now, we have worked with a fixed list of items. What if you want to be able to

remove items from the list? There are a few common mechanisms to achieve that – fixed **Delete** buttons on each item, swiping to delete, and long-clicking to select and then tapping a **Delete** button, to name a few. In this section, we will focus on the *swiping to delete* approach.

Let's start by making our list mutable. If we replace `listOf` with `mutableStateListOf` and wrap it in a `remember` block, changes to our list would reflect on the UI. Our agent list should look like this:

```
val listItems = remember {
    mutableStateListOf(
        ListItemUiModel.GroupTitle("Current Employees"),
        ListItemUiModel.Employee(...),
        ListItemUiModel.Employee(...),
        ListItemUiModel.GroupTitle("Past Employees"),
        ListItemUiModel.Employee(...),
        ListItemUiModel.Employee(...)
    )
}
```

Now, when we remove an element from the list, the UI will automatically refresh to show the element gone.

Next, we need to define swipe gesture detection. In Jetpack Compose, this is a relatively complicated process. Let's break it down.

First, we must declare anchors. Anchors are the final positions in which the composable can be. For example, we can call the default position of the `START` composable and decide that the composable can be swiped right all the way to a position we will call `END`. We can define these positions using `enum class`:

```
private enum class DragAnchors {
    START,
    END
}
```

We can keep this declaration in the same file as our item composable. Now, we can declare the drag state. This is where we will store the current state of the dragging gesture for the composable:

```
val dragState = remember {
    AnchoredDraggableState(initialValue = DragAnchors.START)
}
```

The drag state for the drag gesture is saved as `AnchoredDraggableState`, which is a class that holds the state of the `anchoredDraggable` modifier. We will discuss `anchoredDraggable` shortly. For now, know that it is a modifier that allows us to drag a composable.



You can read more about `anchoredDraggable` in the official Google documentation here: <https://packt.link/VcIMd>.

The argument that we provide to the `AnchoredDraggableState` constructor is `initialValue`. This argument can be data of any type, but `sealed interface`, `sealed class`, and `enum class` are ideal candidates because they encapsulate a closed set of values.

The last argument is `animationSpec`. This is the animation that will be used to settle into a new state after a drag. While `tween` is a good choice, we can use any animation.

When the user changes the state of the draggable, or, more specifically, swipes it right, we want to inform the container of our composable of the event. This is similar to how we informed the container of a click – we add a lambda parameter to our composable:

```
fun Employee(..., onClick: () -> Unit, onSwipe: () -> Unit) {
```

The container can now pass in a lambda to be executed when a swipe occurs. All that's left for us to do is to call the `onSwipe` lambda after a drag. Of course, we have to detect a drag first:

```
    LaunchedEffect(dragState.settledValue) {  
        if (dragState.settledValue == DragAnchors.END) {  
            onSwipe()  
        }  
    }  
}
```

`LaunchedEffect` executes the block we provide when the value passed to it (`dragState.settledValue`, in our case) changes. And so, when the drag gesture completes, we can make sure that the new settled value is `DragAnchors.END`, and if so, trigger the `onSwipe` callback.

But we are not done yet. To make our composable draggable, we have to apply several modifiers to it.

The first modifier we provide detects when our composable resizes. This allows it to update the anchor positions relative to the new size of the composable:

```
.onSizeChanged { layoutSize ->
    dragState.updateAnchors(
        DraggableAnchors {
            DragAnchors.START at 0f
            DragAnchors.END at layoutSize.width.toFloat()
        }
    )
}
```

As you can see, we update the `dragState` anchors, positioning the start at `0` and the end on the rightmost edge of the composable (determined by the layout width).

The next modifier offsets the composable based on the position held in `dragState`:

```
.offset {
    IntOffset(
        x = dragState.requireOffset().roundToInt(),
        y = 0
    )
}
```

Finally, the `anchoredDraggable` modifier ties `dragState` to drag gestures, and sets the gesture orientation:

```
.anchoredDraggable(
    state = dragState,
    orientation = Orientation.Horizontal
)
```

If we want our composable to be draggable through the entire width of its container, we should also use the `fillMaxWidth` modifier. The entire modifier chain of our outermost composable would look like this:

```
modifier = Modifier
    .fillMaxWidth()
    .onSizeChanged { layoutSize ->
        dragState.updateAnchors(
            DraggableAnchors {
```

```

        DragAnchors.START at 0f
        DragAnchors.END at
            layoutSize.width.toFloat()
    }
)
}
.offset {
    IntOffset(
        x = dragState.requireOffset().roundToInt(),
        y = 0
    )
}
.anchoredDraggable(
    state = dragState,
    orientation = Orientation.Horizontal
)
.clickable {
    onClick()
}

```

Now that our item composable communicates swipe events, we would want to communicate those to the app. In the *Employees* app, the composable that will receive these events is our *Employees* composable. And so, we can add an `onItemSwipe` parameter to it:

```

@Composable
fun Employees(
    listItems: List<ListItemUiModel>,
    modifier: Modifier = Modifier,
    onItemClick: (Int) -> Unit = {},
    onItemSwipe: (Int) -> Unit = {}
) { ... }

```

To call `onItemSwipe`, we add a lambda argument to the *Employee* composable inside *Employees*:

```

Employee(
    employee = item.employee,
    onClick = { ... },
    onSwipe = {
        onItemSwipe(index)
    }
)

```

```
    }
  )
}
```

Now, when a user swipes an employee, `onSwipe` will be called, and we will delegate the call to the container of `Employees`, providing the index of the swiped employee. To complete the functionality, all that's left is to update `MainActivity`, passing a lambda to `Employees` that removes an employee when it is swiped:

```
Employees(
    listItems = listItems,
    modifier = Modifier.padding(innerPadding).fillMaxWidth(),
    onItemClick = { ... },
    onItemSwipe = { itemIndex ->
        listItems.removeAt(itemIndex)
    }
)
```

We updated the `Employees` container to take the full width of its container. This makes it take the entire width of the screen. More importantly, though, we added a swipe listener that removes the item from the swiped index. Swiping now works as we intended.

Exercise 6.05 – Adding swipe-to-delete functionality

You previously added a lazy list to your app and then added items of different types to it. You will now allow users to delete some items (you want to let the users remove secret cat agents but not titles) by swiping them right:

1. To start, open `Cat.kt`. At the bottom of the file, declare `DragAnchors`, an enum class type with two values, `START` and `END`:

```
private enum class DragAnchors {
    START,
    END,
}
```

2. Next, add an `onSwipe` callback to the `Cat` composable parameters:

```
fun Cat(
    cat: CatUiModel,
    onClick: () -> Unit = {},
    onSwipe: () -> Unit = {}
)
```

3. At the top of the Cat composable, declare the dragState variable, passing in the initialValue argument of DragAnchors.START:

```
fun Cat(...) {
    val dragState = remember {
        AnchoredDraggableState(
            initialValue = DragAnchors.START)
    }
}
```

4. Add a LaunchedEffect composable that triggers when dragState.settledValue changes. Place it after dragState. Make it report a swipe if the current value of the drag state is DragAnchors.END:

```
LaunchedEffect(dragState.settledValue) {
    if (dragState.settledValue == DragAnchors.END) {
        onSwipe()
    }
}
```

5. Update the Row composable modifier by adding the fillMaxWidth and onSizeChanged modifiers to make the composable fill the width of its container and update the position of the anchors on resize:

```
modifier = Modifier
    .fillMaxWidth()
    .onSizeChanged { layoutSize ->
        dragState.updateAnchors(
            DraggableAnchors {
                DragAnchors.START at 0f
                DragAnchors.END at
                    layoutSize.width.toFloat()
            }
        )
    }
```

6. Right after both modifiers, add the offset modifier to set the position of the composable based on the dragState offset value:

```
.offset {
    IntOffset(
        x = dragState
```

```

        .requireOffset()
        .roundToInt(),
        y = 0
    )
}

```

7. Lastly, add the `anchoredDraggable` modifier to enable dragging:

```

        .anchoredDraggable(
            state = dragState,
            orientation = Orientation.Horizontal
        )

```

8. Open `MainActivity.kt`. Find the `CatAgents` composable and add an `onItemSwipe` callback to it:

```

fun CatAgents(
    listItems: List<ListItemUiModel>,
    modifier: Modifier = Modifier,
    onItemClick: (Int) -> Unit = {},
    onItemSwipe: (Int) -> Unit = {}
) { ... }

```

9. Now, find the `Cat` composable inside the `items` block, and provide an `onSwipe` argument to it:

```

Cat(
    cat = item.cat,
    onClick = { ... },
    onSwipe = {
        onItemSwipe(index)
    }
)

```

10. To enable the removal of items from the list, make `listItems` mutable inside the `Scaffold` block by replacing `listOf` with `remember` and `mutableStateListOf`. Leave the rest of the list unchanged:

```

val listItems = remember {
    mutableStateListOf(
        ListItemUiModel.Title("Sleeper Agents"),

```

```

        ...
    )
}

```

11. Now that you can modify the `listItems` collection, add the `onItemSwipe` argument to the `CatAgents` call:

```

CatAgents(
    listItems = listItems,
    modifier = Modifier
        .padding(innerPadding).fillMaxWidth(),
    onItemClick = { ... },
    onItemSwipe = { itemIndex ->
        listItems.removeAt(itemIndex)
    }
)

```

If you run your app now, you may find that, sometimes, when a cat is swiped away, another item in the list disappears. This happens because the lazy list confuses states between the items—it doesn't know which item is which. To fix that, you need to tell it how to distinguish between items. You do that by assigning each item a unique ID. The following two steps do just that.

12. First, update `ListItemUiModel` by replacing `sealed` interface with `sealed` class and adding an ID field that is set to a random UUID:

```

sealed class ListItemUiModel {
    val id: String = UUID.randomUUID().toString()

    data class Title(val title: String) :
        ListItemUiModel()

    data class Cat(val cat: CatUiModel) :
        ListItemUiModel()
}

```

13. Next, update the `items` block of `CatAgents` in `MainActivity.kt` to read the unique value:

```

items(
    listItems.size,
    key = { index -> listItems[index].id }
) { index ->

```


It is important to mention that this isn't a production-ready solution. Every time you create a new instance of `ListItemUiModel`, even if you intended for it to replace an existing instance, the lazy list will consider it to be a new element. It won't be able to associate the two. In production, items usually have persisted unique identifiers.

14. Run your app. You should now be able to swipe secret cat agents right to remove them from the list. Note that the lazy list handles the collapsing animation for us (*Figure 6.11*):

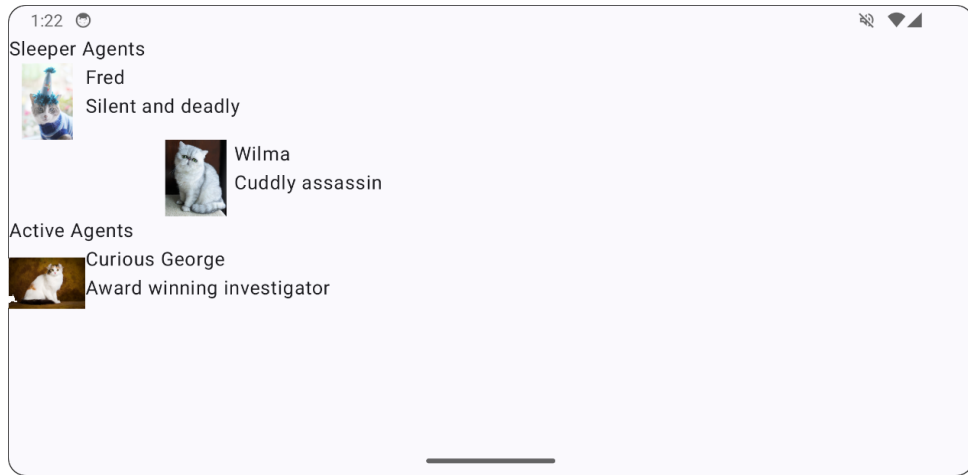


Figure 6.11 – A cat (Wilma) being swiped to the right

Note how titles cannot be swiped. You have implemented swiping only for the Cat composables. Now, you know how to remove items interactively. Next, you will learn how to add new items as well.

Adding items interactively

We have just learned how to remove items interactively. What about adding new items? Let's look into it. First, let's add a button to the top of our list:

```
Column(
    modifier =
        Modifier.padding(innerPadding).fillMaxWidth()
) {
    Button(onClick = {}) {
        Text(text = "Add Employee")
    }
    Employees( ... ) { ... }
}
```

Note that we moved the padding modifier from `Employees` to the `Column` composable because it should be applied to the outermost composable. We now have a button above our list for adding employees.

In a production app, you could add a rationale about what a new item would be. For example, you could have a form for a user to fill in different details. For the sake of simplicity, in our example, we will always add the same dummy item – an anonymous employee.

To add the functionality of actually adding the employee to the list, let's update the `onClick` block of the button that we introduced previously:

```
onClick = {
    listItems.add(
        1,
        ListItemUiModel.Employee(
            EmployeeUiModel(
                gender = Gender.MALE,
                role = EmployeeRole.HUMAN_RESOURCES,
                name = "Anonymous",
                imageUrl = "https://..."
            )
        )
    )
}
```

And that is it. We add the item at position 1 so that it is added right below our first title, which is the item at position 0. In a production app, you could have logic to determine the correct place to insert an item. It could be below the relevant title; items may always be added at the top, bottom, or in the correct place to preserve some existing order.

We can now run the app. We will now have a new **Add Employee** button. Every time we click the button, an anonymous employee will be added to the lazy list. The newly added employees can be swiped away to be removed, just like the hardcoded employees before them.

Exercise 6.06 – Implementing an Add Cat button

Having implemented a mechanism to remove items, it is time we implemented a mechanism to add items:

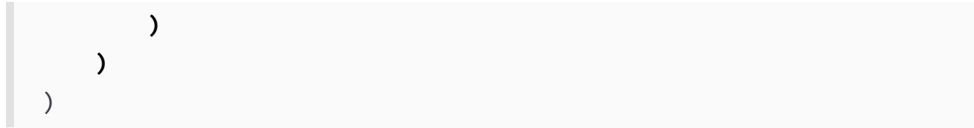
1. Add a button to the content of `MainActivity`, updating the layout, which currently consists of a `CatAgents` composable:

```
Column(  
    modifier = Modifier  
        .padding(innerPadding)  
        .fillMaxWidth()  
) {  
    Button(onClick = {}) {  
        Text(text = "Add Cat")  
    }  
    CatAgents(  
        listItems = listItems,  
        onItemClick = { ... },  
        onItemSwipe = { ... }  
    )  
}
```

Note that we moved the modifiers from `CatAgents` to the new enclosing `Column` composable.

2. Update the contents of the newly introduced `onClick` lambda function of the `Button` composable:

```
onClick = {  
    listItems.add(  
        1,  
        ListItemUiModel.Cat(  
            CatUiModel(  
                gender = Gender.FEMALE,  
                name = "Anonymous",  
                biography = "Unknown",  
                imageUrl =  
                    "https://cdn2.thecatapi.com/images/  
                    /zJkeHza2K.jpg"  
            )  
        )  
    )  
}
```



3. Run the app. You should see a new button at the top of your app. *Figure 6.12* shows the app after the button is clicked once:

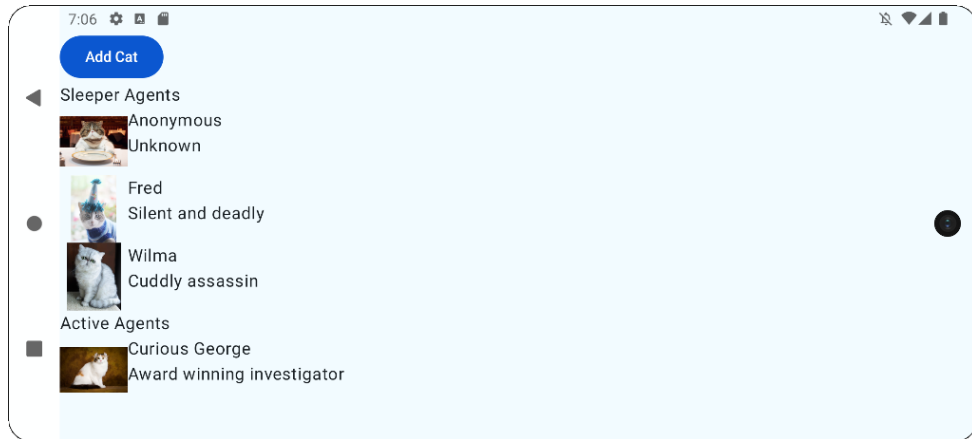


Figure 6.12 – An anonymous cat is added with the click of a button

4. Try clicking it a few times. Every time you click it, a new, anonymous secret cat agent is added to your lazy list. You can swipe away the newly added cats, just like you could with the hardcoded ones.

In this exercise, you added new items to a lazy list in response to user interaction. You now know how to change the contents of a lazy list at runtime. It is useful to know how to update lists at runtime because, quite often, the data you are presenting to your users changes while the app is running, and you want to present your users with a fresh, up-to-date state.

Activity 6.01 – Managing a list of Items

Imagine you want to develop a recipe management app. Your app would support sweet and savory recipes. Users of your app could add new sweet or savory recipes, scroll through the list of added recipes – grouped by flavor (sweet or savory) – click a recipe to get information about it, and finally, delete recipes by swiping them aside.

The aim of this activity is to create an app with `RecyclerView` that lists the titles of recipes, grouped by flavor. `RecyclerView` will support user interaction. Each recipe will have a title, a description, and a flavor. Interactions will include clicks and swipes.

A click will present a user with a dialog showing the description of the recipe. A swipe will remove the swiped recipe from the app. Finally, with two `EditText` fields (see *Chapter 3, Developing the UI with Jetpack Compose*) and two buttons, a user can add a new sweet or savory recipe, respectively, with the title and description set to the values in the `EditText` fields.

The steps to complete this are as follows:

1. Create a new empty activity app.
2. Add a `LazyColumn` composable, two `TextField` composables (one for entering recipe titles and another for adding recipe descriptions), and two buttons (one to add a savory recipe and one to add a sweet one) to the main layout. Only allow one line of text for the title. If you add a preview, it should look similar to *Figure 6.13*:

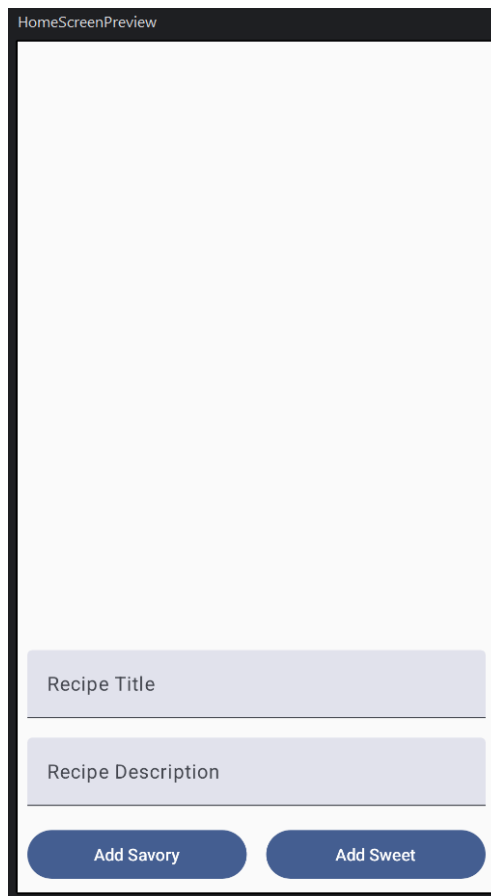


Figure 6.13 – The layout with a `LazyColumn` composable, two `TextField` composables, and two buttons

3. Add an enum for flavor with two values: savory and sweet. Add a model to hold a recipe. Add a sealed class model for list items with two data types: one for titles and one for recipes.
4. Create a composable for a recipe, with a title and the first line of its description.
5. Add a mutable list of recipes. Populate it with a title for savory recipes and another title for sweet recipes. Pass the list to your main composable.
6. Add the list to the LazyColumn composable. Handle the different types of items.
7. Delegate clicks on both buttons to MainActivity. Make sure both buttons trigger the relevant callback(s). Pass the title, description, and the type of recipe in the callback(s). Make sure the form is cleared after a recipe is added.
8. Update MainActivity to add a new recipe when a click is reported. Add the recipe under the correct title.
9. When a recipe is clicked, show a toast with the recipe description.
10. When a recipe is swiped right, delete it.
11. Bonus step: try implementing the same app without using a list item. Use two recipe lists: one for savory recipes and another for sweet ones. Instead of having one items block, have one for each of the two flavor types. For the titles, explore the item block. It acts much like the items block but is designed to present a single composable. The final output should resemble *Figure 6.14*:

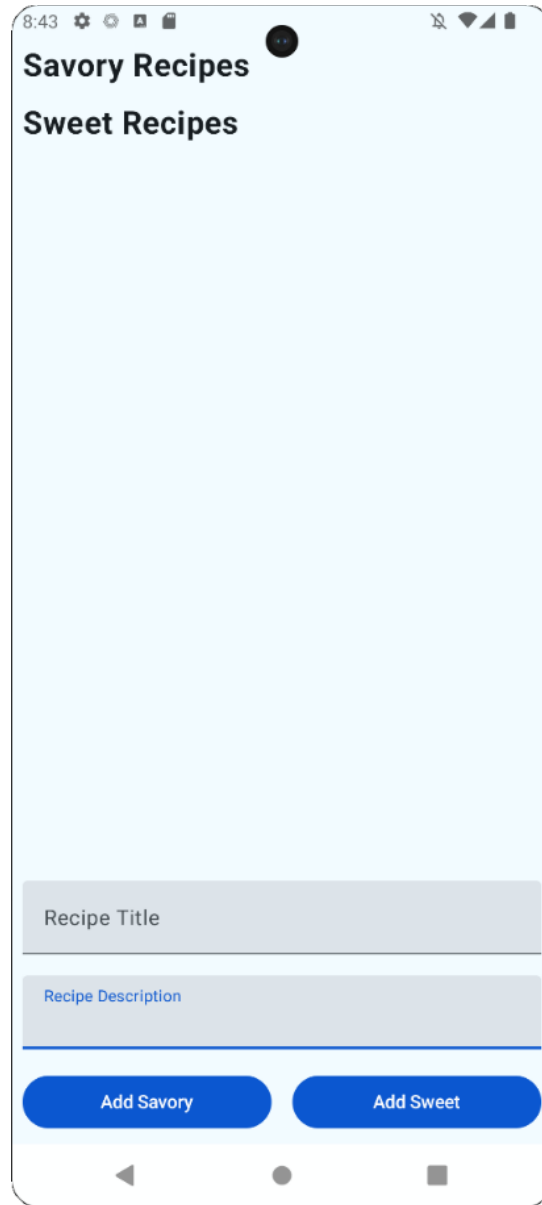


Figure 6.14 – The Recipe Book app



The solution to this activity can be found at <https://packt.link/2IToe>.

Summary

In this chapter, we learned how to add a `LazyList` composable to our app. We also learned how to populate it with items. We went through adding different item types, which is particularly useful for titles. We covered interaction with `LazyList` composable items responding to clicks and swipe gestures.

Lastly, we learned how to dynamically add and remove items to and from a lazy list. The world of lazy lists is very rich, and we have only scratched the surface. Going further would be beyond the scope of this book. However, it is strongly recommended that you investigate it on your own so that you can have carousels, designed dividers, and fancier swipe effects in your apps.

In the next chapter, we will look into requesting special permissions on behalf of our app to enable it to perform certain tasks, such as accessing a user's contacts list or microphone. We will also explore using Google's Maps API and accessing a user's physical location.

Further reading

You can start your exploration of the rich world of lazy lists here: <https://packt.link/U5GZo>.

Unlock this book's exclusive benefits now

Scan this QR code or go to packtpub.com/unlock, then search this book by name.

Note: Keep your purchase invoice ready before you start.

UNLOCK NOW



7

Android Permissions and Google Maps

In the previous chapter, we learned how to present data in lists using lazy lists. Then, we used that knowledge to present the user with a list of secret cat agents. This chapter will teach you how to request and obtain app permissions in Android to access device features that provide richer functionality. You will also gain a solid understanding of how to include local and global interactive maps in your app by using the Google Maps API.

First, we will explore the Android permissions system. Many Android features are not immediately available to us. These features are gated behind a permission system to protect the user. For us to access those features, we must ask the user to allow us to do so. These features include, but are not limited to, obtaining the user's location, accessing the user's contacts, accessing their camera, and establishing a Bluetooth connection. Different Android versions enforce different permission rules.



When Android 6 (Marshmallow) was introduced, back in 2015, for example, several permissions you could silently obtain on installation were deemed insecure and became runtime permissions, requiring explicit user consent.

We will then look at the Google Maps API. This API allows us to present the user with a map of any desired location in the world. We will add data to that map and let the user interact with the map. The API also lets you show points of interest and render a street view of supported locations, though we will not explore these features in this book.

We will cover the following topics in this chapter:

- Requesting permission from the user
- Showing a map of the user's location
- Map clicks and custom markers

By the end of the chapter, you will be able to create permission requests for your app and handle missing permissions. You will also learn how to find the user's current location on the map. You will practice these learnings through the deployment of cat agents to the field by selecting locations on the map.

Technical requirements

The complete code for all the exercises and the activity in this chapter is available on GitHub at <https://packt.link/mPVuE>.

Requesting permission from the user

We might want to implement certain features on the app that Google deems dangerous. This usually means access to those features could risk the user's privacy. For example, some permissions may allow us to determine the user's current location or read their messages.

Depending on the required permission and the target Android API level we are developing, we may need to request that permission from the user. If the device is running on Android 6 (Marshmallow, API level 23), and the target API of our app is 23 or higher (it almost certainly will be, as most devices by now will run newer versions of Android), there will be no alert for the user about any permissions requested by the app at install time. Instead, our app must ask the user to grant those permissions at runtime.

When we request permission, the user sees a dialog like the one shown in *Figure 7.1*:

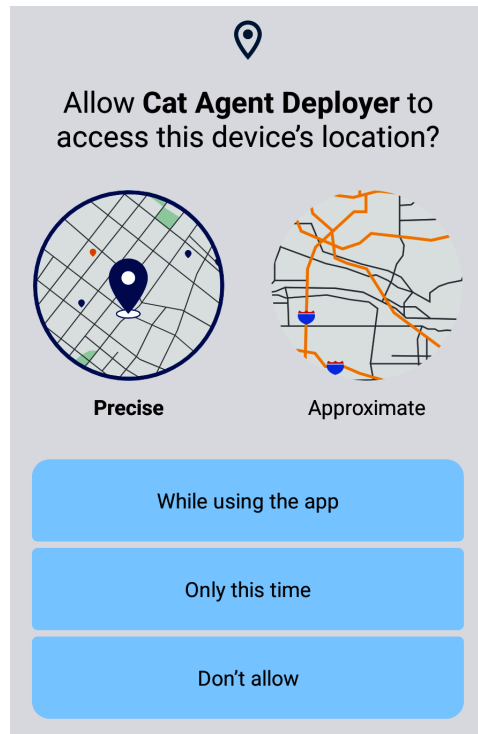


Figure 7.1 – Permission dialog for device location access



For a full list of permissions and their protection level, see here: <http://packt.link/57BdN>.

For an overview of permissions in Android, see here: <https://packt.link/SEICd>.

We must include permissions in our manifest file when we intend to use them. A manifest with the `SEND_SMS` permission would look something like the following snippet:

```
<manifest>
  xmlns:android= "http://schemas.android.com/apk/res/android"
  package="com.example.snazzyapp">
    <uses-permission
      android:name="android.permission.SEND_SMS" />
    <application ...> ... </application>
  </manifest>
```

💡 **Quick tip:** Enhance your coding experience with the **AI Code Explainer** and **Quick Copy** features. Open this book in the next-gen Packt Reader. Click the **Copy** button (1) to quickly copy code into your coding environment, or click the **Explain** button (2) to get the AI assistant to explain a block of code to you.

```
function calculate(a, b) {  
  return {sum: a + b};  
};
```

Copy

1

Explain

2



🔒 The next-gen Packt Reader is included for free with the purchase of this book. Scan the QR code OR go to packtpub.com/unlock, then use the search bar to find this book by name. Double-check the edition shown to make sure you get the right one.



Safe permissions (or **normal** permissions, as Google calls them) would be automatically granted to the user. However, **dangerous** ones would only be granted if explicitly approved by the user. If we fail to request permission from the user and try to execute an action that requires that permission, the best we can hope for is that the action will be performed. At worst, our app could crash.

Before asking the user for permission, we should first check whether the user has already granted us that permission. If the user has not yet granted us permission, we may need to check whether a rationale dialog should be shown prior to the permission request. This depends on how obvious the justification for the request would be to the user.

For example, if a camera app requests permission to access the camera, we can safely assume that the reason would be clear to the user. However, some cases may not be as clear to the user, especially if the user is not tech-savvy. In those cases, we may have to justify the request to the user before asking for it.

Google provides us with an activity function called `shouldShowRequestPermissionRationale(String)` for this purpose. Under the hood, this function has an SDK version and past user behavior checks.

The idea of this mechanism is to allow us to justify our request to the user for permission before requesting it, thus increasing the likelihood of them approving the request. Once we determine whether the app should present the user with our rationale or whether no rationale is required, we can proceed to request permission.

Note that if the user denies permission, we can check `shouldShowRequestPermissionRationale(String)` again. If it returns `true`, we can present our rationale and then ask the user for permission once more.

Let's see how we can request permission. To request permissions in Jetpack Compose, we must first declare a launcher for the activity result. Here is how it looks:

```
val requestLocationPermissionLauncher =
    rememberLauncherForActivityResult(
        contract = ActivityResultContracts
            .RequestMultiplePermissions(),
        onResult = { permissions ->
            locationPermissionsGranted =
                permissions.values.all { it }
            if (!locationPermissionsGranted) {
                shouldShowLocationPermissionRationale =
                    shouldShowLocationPermissionRationale()
            }
        }
    )
```

We obtain a launcher by calling `rememberLauncherForActivityResult`. For the contract parameter, we pass in the `RequestMultiplePermissions` contract. As the name suggests, this is the `ActivityResultContracts` contract for requesting permissions.

In the `onResult` callback, we check whether all requested permissions were granted. We assign the result of this check to `locationPermissionsGranted`, which is a mutable Boolean state. If not all permissions are granted, we update the `shouldShowLocationPermissionRationale` variable, which is another mutable Boolean state.

When placed inside the `setContent` block, `requestLocationPermissionLauncher` allows us to request the location permission. This is how it is used:

```
fun requestLocationPermission() {
    requestLocationPermissionLauncher.launch(
        arrayOf(
```

```

        Manifest.permission.ACCESS_FINE_LOCATION,
        Manifest.permission.ACCESS_COARSE_LOCATION
    )
)
}

```

There are several reasonable triggers for requesting the location permission: the activity starting and a button being pressed, which starts a certain flow, for example. For simplicity, let us assume that we have a button that triggers the request. This will be how its `onClick` argument might look:

```

if (!locationPermissionsGranted) {
    shouldShowLocationPermissionRationale =
        shouldShowLocationPermissionRationale()
}
if (!locationPermissionsGranted &&
    !shouldShowLocationPermissionRationale
) {
    requestLocationPermission()
}

```

First, we check that we don't already have the location permission. If we don't, we update `shouldShowLocationPermissionRationale`. We continue to check whether the rationale is needed. If not, we can go ahead and request the location permission. Of course, as part of a complete feature, we will likely perform some action if we have the location permission. But we will get to that later.

The previous code snippets don't offer a complete picture. We still have a few missing pieces. First, we need `locationPermissionsGranted` to be initialized with the correct value:

```

var locationPermissionsGranted by remember {
    mutableStateOf(areLocationPermissionsGranted())
}

```

We also need a function that checks the permissions:

```

private fun areLocationPermissionsGranted(): Boolean =
    ContextCompat.checkSelfPermission(
        this,
        Manifest.permission.ACCESS_FINE_LOCATION
    ) == PackageManager.PERMISSION_GRANTED

```

Note that we only check for the `ACCESS_FINE_LOCATION` constant. This is because what we really want is permission to know the user's exact location; whether they allowed us to access their coarse location or haven't allowed us to access their location at all doesn't matter – in neither case do we have the access we want.

Next, let's see how we check whether rationale should be shown to the user:

```
private fun shouldShowLocationPermissionRationale() =
    shouldShowRequestPermissionRationale(
        Manifest.permission.ACCESS_COARSE_LOCATION
    ) || shouldShowRequestPermissionRationale(
        Manifest.permission.ACCESS_FINE_LOCATION
    )
```

We check whether the Android SDK wants us to present the rationale for either of the permissions – fine or coarse. If it does, we return true. Otherwise, we return false. The final missing piece of the puzzle is the presentation of the rationale itself. In our example, we will use a snackbar. To support a snackbar, we must first introduce a `snackbarHostState` variable:

```
val snackbarHostState = remember { SnackbarHostState() }
```

With the state in place, we can add it to the Scaffold composable function:

```
Scaffold(
    modifier = Modifier.fillMaxSize(),
    snackbarHost = {
        SnackbarHost(hostState = snackbarHostState)
    }
)
```

Finally, we can observe `shouldShowLocationPermissionRationale` and show a snackbar if it becomes true:

```
val scope = rememberCoroutineScope()

LaunchedEffect(
    key1 = shouldShowLocationPermissionRationale,
    block = {
        if (shouldShowLocationPermissionRationale) {
            scope.launch {
                val userAction = snackbarHostState
```



```
        .showSnackbar(  
            message = "[Rationale here]",  
            actionLabel = "Approve",  
            duration = SnackbarDuration.Indefinite,  
            withDismissAction = true  
        )  
        when (userAction) {  
            SnackbarResult.ActionPerformed -> {  
                requestLocationPermission()  
            }  
  
            SnackbarResult.Dismissed -> {}  
        }  
    }  
}  
}
```

This code triggers an effect when the value of `shouldShowLocationPermissionRationale` changes. If it changes to `true`, we show the user a snackbar with a brief explanation of why we need their permission (Figure 7.2). The **Approve** button will request permission again, and dismissal will have no side effects.

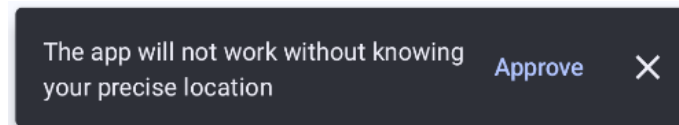


Figure 7.2 – Rationale snackbar

This wraps up the implementation details of requesting permissions. Let's start with our first exercise.

Exercise 7.01 – requesting the location permission

In this exercise, we will request that the user allow us to access their location. We will first create an empty activity project. Then, we will declare the permissions required in the manifest file. Finally, we will ask for the necessary permissions and present our rationale if needed.

Follow the next steps:

1. Start by creating a new empty activity project (**File | New | New Project | Empty Activity**). Click **Next**.
2. Name your application `Cat Agent Deployer`. Make sure your package name is `com.example.catagentdeployer`.
3. Set the save location to where you want to save your project. Leave everything else at the default values and click **Finish**.
4. Make sure you are on the **Android** view in your **Project** pane.
5. Open your `AndroidManifest.xml` file. Report that the app requires location access permission by adding the following tags:

```
<manifest ...>
    <uses-permission android:name=
        "android.permission.ACCESS_COARSE_LOCATION" />
    <uses-permission android:name=
        "android.permission.ACCESS_FINE_LOCATION" />
    <application ...> ... </application>
</manifest>
```



`ACCESS_FINE_LOCATION` is the permission you will need to obtain the user's location based on GPS. To obtain less accurate Wi-Fi and mobile data-based location information, you could request the `ACCESS_COARSE_LOCATION` permission. On Android 12 or higher, requesting the coarse location permission is mandatory when asking for the fine location one.

6. Open your `MainActivity.kt` file. At the bottom of the `MainActivity` class block, add an empty `areLocationPermissionsGranted()` function:

```
private fun areLocationPermissionsGranted(): Boolean =
    ContextCompat.checkSelfPermission(
        this,
        Manifest.permission.ACCESS_FINE_LOCATION
    ) == PackageManager.PERMISSION_GRANTED
```

This will check whether the user has granted you the fine location permission. Remember – if they granted you the coarse location permission or no location permission at all, you want to request the location permission, because your goal is to obtain the accurate location of the user.

7. Right above this function, introduce another function for determining whether rationale should be presented to the user:

```
private fun shouldShowLocationPermissionRationale() =
    shouldShowRequestPermissionRationale(
        Manifest.permission.ACCESS_COARSE_LOCATION
    ) || shouldShowRequestPermissionRationale(
        Manifest.permission.ACCESS_FINE_LOCATION
    )
```

8. Next, introduce two variables at the top of the setContent block – one to remember whether the location permission was granted and another to remember whether we need to show the permission rationale:

```
var locationPermissionsGranted by remember {
    mutableStateOf(areLocationPermissionsGranted())
}

var shouldShowLocationPermissionRationale by remember {
    mutableStateOf(false)
}
```

The default value for the permission granted flag is fetched from the current permission state because you will want to always rely on its value. The rationale flag is set to false because you will update it when the flow warrants it.

9. Now, add a request permission launcher immediately after the two new variables:

```
val requestLocationPermissionLauncher =
    rememberLauncherForActivityResult(
        contract = ActivityResultContracts
            .RequestMultiplePermissions(),
        onResult = { permissions ->
            locationPermissionsGranted =
                permissions.values.all { it }
            if (!locationPermissionsGranted) {
                shouldShowLocationPermissionRationale =
                    shouldShowLocationPermissionRationale()
            }
        })
```

This is the variable through which we will launch the permission request and track user responses.

10. Add a function under `requestLocationPermissionLauncher` to request the location permissions:

```
fun requestLocationPermission() {
    requestLocationPermissionLauncher.launch(
        arrayOf(
            Manifest.permission.ACCESS_FINE_LOCATION,
            Manifest.permission.ACCESS_COARSE_LOCATION
        )
    )
}
```

11. To present users with the rationale for the permission request, first declare the `snackbarHostState` variable before your `Scaffold` composable:

```
val snackbarHostState = remember { SnackbarHostState() }
```

12. Continue to add a `SnackbarHost` function to the `Scaffold` composable function:

```
Scaffold(
    modifier = Modifier.fillMaxSize(),
    snackbarHost = {
        SnackbarHost(hostState = snackbarHostState)
    }
) { ... }
```

13. After the `Greeting` block, add a `LaunchedEffect` composable function to track the state of the rationale flag:

```
val scope = rememberCoroutineScope()

LaunchedEffect(
    key1 = shouldShowLocationPermissionRationale,
    block = {
        if (shouldShowLocationPermissionRationale) {
        }
    }
)
```

14. Finally, populate the if statement:

```
if (shouldShowLocationPermissionRationale) {
    scope.launch {
        val userAction =
            snackbarHostState.showSnackbar(
                message = "The app will not work
                    without knowing your precise
                    location",
                actionLabel = "Approve",
                duration = SnackbarDuration.Indefinite,
                withDismissAction = true
            )
        when (userAction) {
            SnackbarResult.ActionPerformed -> {
                requestLocationPermission()
            }

            SnackbarResult.Dismissed -> {}
        }
    }
}
```

This code will present the user with a snackbar explaining that the app will not work without knowing their current location, as shown in *Figure 7.2*. Tapping **Approve** will request the location permission again. Dismissing the snackbar will have no side effects.

15. Finally, replace the call to `Welcome()` with a button that launches the permission flow, if necessary:

```
Button(
    onClick = {
        if (!locationPermissionsGranted) {
            shouldShowLocationPermissionRationale =
                shouldShowLocationPermissionRationale()
        }
        if (!locationPermissionsGranted &&
            !shouldShowLocationPermissionRationale
        ) {
```

```
        requestLocationPermission()
    },
    modifier = Modifier.padding(innerPadding)
) {
    Text(text = "Request permission")
}
```

16. Run your app. You should see a **Request permission** button. Click that button. You should now see a system permission dialog requesting you to allow the app to access the location of the device, as shown in *Figure 7.3*.

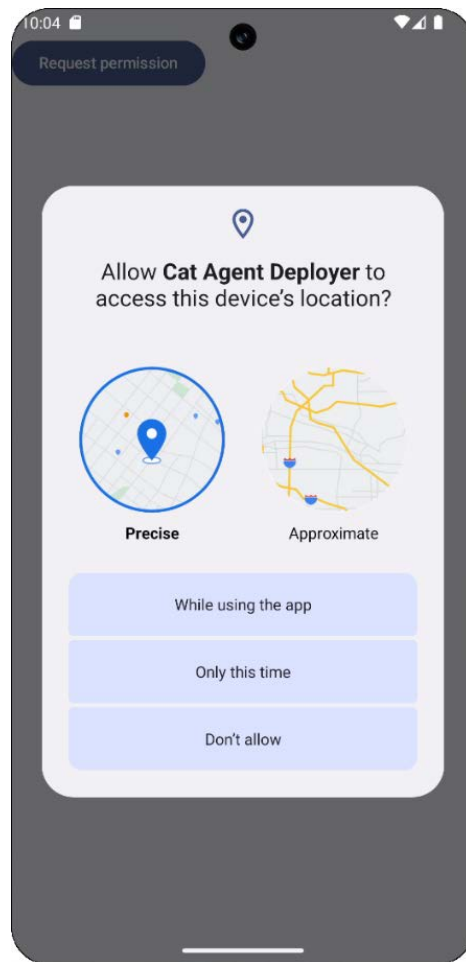


Figure 7.3 – App requesting the location permission

If the user denies the permission, the rationale dialog will appear. If the rationale is accepted, the system permission dialog will show again. Up until SDK 31, the user had the option to choose not to let the app ask for permission again (*Figure 7.4*). From SDK 31 onward, not asking a third time is the default. Allowing it afterward requires using the device settings.

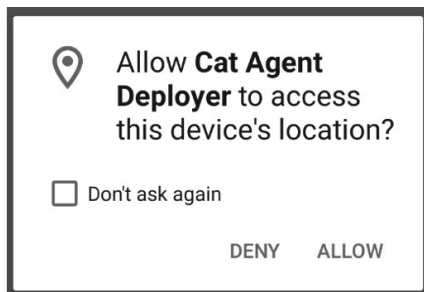


Figure 7.4 – The Don't ask again message

Once the user has allowed or permanently denied the permission, the dialog will never show again. To reset the state of your app permissions, you would have to manually grant it permission via the **App Info** interface or clear the app data.

Now that we can get the location permission, we will look into obtaining the user's current location.

Showing a map of the user's location

Having successfully obtained permission from the user to access their location, we can now access the last known location of the device. This location is also usually the user's current location. We can use this information to present the user with a map of their current location.

To obtain the user's last known location, Google provides the Google Play location service and, more specifically, the `FusedLocationProviderClient` class. The `FusedLocationProviderClient` class helps us interact with Google's Fused Location Provider API, which is a location API that intelligently combines different signals from multiple device sensors to provide us with device location information.

To access the `FusedLocationProviderClient` class, we must first include the Google Play location service library in our project. This simply means adding the following dependency to our app, using the latest version available:

```
play-services-location = {  
    group = "com.google.android.gms",  
    name = "play-services-location",  
    version.ref = "playServicesLocation"  
}
```

With the location service imported, we can now obtain an instance of the `FusedLocationProviderClient` class by calling `LocationServices.getFusedLocationProviderClient(this@MainActivity)`.

Once we have a fused location client, we can obtain the user's current location by calling `fusedLocationClient.getCurrentLocation()`. The input parameters of this function are the **priority** of the request and a **cancellation token**. The name of the first parameter is somewhat misleading because the valid values all indicate a strategy rather than a priority. The permissible values for priority are `PRIORITY_HIGH_ACCURACY`, `PRIORITY_BALANCED_POWER_ACCURACY`, `PRIORITY_LOW_POWER`, and `PRIORITY_PASSIVE`. A cancellation token is obtained by instantiating a `CancellationTokenSource` object and reading its token value. This mechanism allows us to cancel the request if we no longer need a response. It may make sense to cancel the request if the user leaves the current screen, for example. To cancel the request, we can call `cancel()` on the `CancellationTokenSource` instance we created earlier.

The `getCurrentLocation` function is an asynchronous call and returns a `Task<Location>` instance. `Task` is a Google API abstract class whose implementations perform async operations. To read the value from the task, we should also provide a success listener. If we wanted to, we could also add listeners for cancellation, failure, and the completion of requests.



We can only obtain the current location if we have already received the location permission from the user.

In this case, the task in question returns a location. Adding listeners is simple:

```
fusedLocationClient.getCurrentLocation(...)
    .addOnSuccessListener { location -> }
```

It's important to note that the `location` parameter could be `null` if the client fails to obtain the user's current location. This is not very common but could happen if, for example, the user disabled their location services during the call.

Once the code inside our success listener block is executed, and `location` is not `null`, we have the user's current location in the form of a `Location` instance.

A `Location` instance holds a single coordinate on Earth, expressed using **longitude** and **latitude** `Double` values.



For our purpose, it is sufficient to know that every single point on the surface of the Earth can be expressed as a single pair of **longitude** (Lng) and **latitude** (Lat) values.

This is where it gets exciting. Google lets us present any location on an interactive map by using a `GoogleMap` composable. All it takes is signing up for a free API key. We can obtain one here: <https://packt.link/FK58V>.

On the website, follow the directions and create an API key. Once we have a key, we can add it to the project. Google recommends using `secrets-gradle-plugin` to protect the key. The plugin helps us avoid saving our key in our version control system by reading the value from `secrets.properties`, which is omitted from such systems. To use the plugin, first update your `libs.versions.toml` file to include the secrets plugin under `[libraries]`:

```
[versions]
...
secretsGradlePlugin = "[latest version]"

[libraries]
...
secrets-gradle-plugin = { module = "com.google.android.libraries
.mapsplatform.secrets-gradle-plugin:secrets-gradle-plugin", version.ref =
"secretsGradlePlugin" }
```

Next, add a `buildscript` block to the top of the project `build.gradle` file. If it already exists, we should update it to add the plugin inside the `dependencies` block:

```
buildscript {
    dependencies {
        classpath( libs.secrets.gradle.plugin )
    }
}
```

We continue to add the plugin to the `build.gradle` app module:

```
plugins {
    ...
```

```
        id("com.google.android.libraries.mapsplatform  
            .secrets-gradle-plugin")  
    }
```

We also need to configure the plugin, so while we have the app module open, we can add the following at the bottom of the same file:

```
secrets {  
    propertiesFileName = "secrets.properties"  
    defaultPropertiesFileName = "local.defaults.properties"  
}
```

This tells the plugin in which file to find our secrets, and which file to fall back on in the absence of that file. With the plugin installed, we can create or edit the `secrets.properties` files at the root directory of our project to add our API key:

```
MAPS_API_KEY=(our key)
```

To prevent the project from failing to build in the absence of the `secrets.properties` file, which wouldn't be present if we ever cloned our project from our version control system, we create a `local.defaults.properties` file at the root directory of our project and update it to contain the following:

```
MAPS_API_KEY=DEFAULT_API_KEY
```

Finally, we need to update our `AndroidManifest.xml` file to read `MAPS_API_KEY`, by adding the following tag within the application tag:

```
<meta-data  
    android:name="com.google.android.geo.API_KEY"  
    android:value="${MAPS_API_KEY}" />
```

Next, we can add the *Google Maps Compose* dependency to our project:

```
google-maps-compose = {  
    group = "com.google.maps.android",  
    name = "maps-compose",  
    version.ref = "googleMapsCompose"  
}
```

With all dependencies sorted out, we can add a `GoogleMap` composable to our `Scaffold` composable function. At this point, if you run your app, you will see an interactive map on your screen, as seen in *Figure 7.5*.



Figure 7.5 – Interactive map

To position the map based on our current location, we create a `LatLng` instance with the coordinates from our `Location` instance and assign it to the `cameraPositionState` parameter of the `GoogleMap` composable. The call would look something like this:

```
val cameraPositionState = rememberSaveable(
    currentUserLocation,
    saver = CameraPositionState.Saver
) {
    CameraPositionState(
        position = CameraPosition
            .fromLatLngZoom(currentUserLocation, 10f)
    )
}
GoogleMap(cameraPositionState = cameraPositionState)
```

We start by creating a saveable `CameraPositionState` object. This class is a bridge between a `CameraPosition` object and our Jetpack Compose code. `CameraPosition` stores the properties of the camera through which we see the map – such as the target, zoom, and tilt. As an input to the `rememberSaveable` function, we pass in the `currentUserLocation` state. This ensures that the camera state is updated whenever we update the current user’s location. The second argument we pass is `CameraPositionState.Saver`, which implements the saving behavior of the state and is provided by the Google Maps SDK.

The `CameraPositionState` class itself takes a position, which we generate by calling `CameraPosition.fromLatLngZoom`, which, as its name suggests, produces a position given a `LatLng` coordinate and a zoom level. We pass in the `currentUserLocation` coordinates and a fixed zoom of `10f`.



Valid zoom values range between 2.0 (farthest) and 21.0 (closest). Values outside of that range are capped. Some areas may not have tiles to render the closest zoom values.

Next, let’s add a marker to the map at the user’s coordinate. Markers are composables. They use a `MarkerState` instance to position themselves on the map. Marker composables accept other values, such as a title to present when the pin is tapped. Looking at the `Marker` function declaration, we can see all of the configuration options available. I will leave exploring the different possibilities to you. For now, let’s start with a saveable `MarkerState` object:

```
val markerState = rememberSaveable(
    currentUserLocation,
```

```
saver = MarkerState.Saver
) {
    MarkerState(position = currentUserLocation)
}
```

Declaring the saveable marker state is very similar to declaring the map state. We pass in the `currentUserLocation` variable to ensure we track changes to it, pass a saver for persisting the `MarkerState` value, and finally, produce a `MarkerState` object from the `currentUserLocation` value.

When using the `markerState` variable, we first ensure that it holds valid coordinates. Note that while `(0, 0)` is indeed a valid coordinate, it is extremely unlikely that we would want to use it for any real case, so it is reasonably safe to use as an unset value. If the coordinate is anything but `(0, 0)`, we add a marker at the position provided by the state. We also set a title for the marker:

```
if (markerState.position.latitude != 0.0 &&
    markerState.position.longitude != 0.0
) {
    Marker(state = markerState, title = "Pin Label")
}
```

Let's practice obtaining the user's current location and placing it on a map in the following exercise.

Exercise 7.02 – obtaining the user's current location

Now that your app can be granted location permission, you can use the location permission to get the user's current location. You will then display the map and update it to zoom in on the user's current location and show a pin at that location. To do this, perform the following steps:

1. Obtain an API key from <http://packt.link/FK58V>. Follow the instructions on the website until you have generated a new API key.
2. Next, add the secrets Gradle plugin to your version catalog:

```
[versions]
...
secretsGradlePlugin = "[latest version]"

[libraries]
...
```

```
secrets-gradle-plugin = { module =  
    "com.google.android.libraries.mapsplatform.secrets-gradle-  
        plugin:secrets-gradle-plugin",  
    version.ref = "secretsGradlePlugin" }
```

3. Add the plugin to the top of your `build.gradle` project file, too:

```
buildscript {  
    dependencies {  
        classpath( libs.secrets.gradle.plugin )  
    }  
}
```

4. Add the plugin to the `plugins` block at the top of your `build.gradle` app file.
5. Still in the `build.gradle` app file, add a `secrets` configuration block at the end of the file:

```
secrets {  
    propertiesFileName = "secrets.properties"  
    defaultPropertiesFileName =  
        "local.defaults.properties"  
}  
  
plugins {  
    ...  
    id("com.google.android.libraries.mapsplatform  
        .secrets-gradle-plugin")  
}
```

6. At the root of the project, create a `secrets.properties` file. Populate it with the Google Maps API key you obtained in the first step:

```
MAP_API_KEY=[Your API key]
```

7. Still at the root of the project, create a `local.defaults.properties` file and add the following into it precisely as shown:

```
MAPS_API_KEY=DEFAULT_API_KEY
```

Your project file structure should look like *Figure 7.6*.

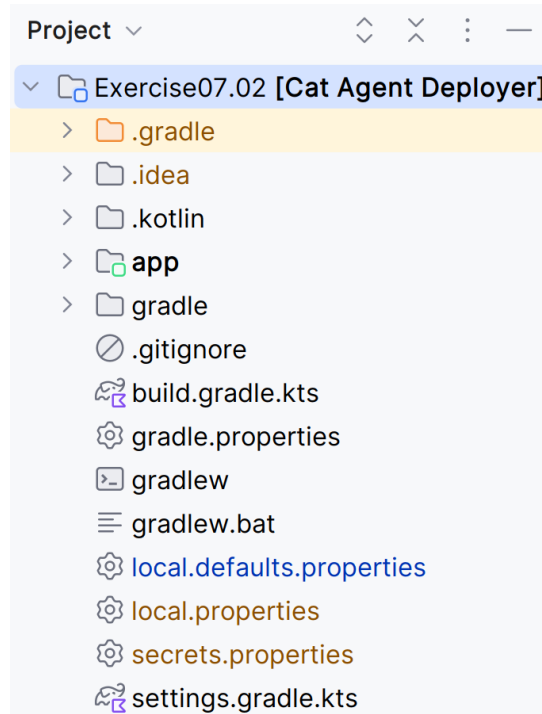


Figure 7.6 – Interactive map with a marker at the current location

8. Add the two following dependencies to your project, using the latest versions available. Include both in your `build.gradle app`:

```
play-services-location = { group = "com.google.android.gms", name =
    "play-services-location", version.ref = "playServicesLocation" }
google-maps-compose = { group = "com.google.maps.android", name =
    "maps-compose", version.ref = "googleMapsCompose" }
```

9. Click the **Sync Project with Gradle Files** button in Android Studio for Gradle to fetch the newly added dependency.
10. Open your `MainActivity.kt` file. At the top of your `MainActivity` class, define a lazily initialized fused location provider client:

```
class MainActivity : ... {
    private val fusedLocationProviderClient by lazy {
        LocationServices.getFusedLocationProviderClient(this)
    }
    override fun onCreate(savedInstanceState: Bundle?) { ... }
```

```
...
}
```

By making `fusedLocationProviderClient` initialize lazily, you are ensuring it is only initialized when needed, which essentially guarantees the `Activity` class will have been created before initialization.

11. Introduce a variable for holding the current user's location inside the `setContent` block:

```
var currentUserLocation by remember {
    mutableStateOf(LatLng(0.0, 0.0))
}
```

12. Now, introduce a `getUserLocation()` function immediately after `currentUserLocation`:

```
fun getUserLocation() {
    val cancellationTokenSource = CancellationTokenSource()
    lifecycleScope.launch
    @SuppressLint("MissingPermission") {
        suspendCancellableCoroutine { continuation ->
            fusedLocationClient.getCurrentLocation(
                PRIORITY_HIGH_ACCURACY,
                cancellationTokenSource.token
            ).addOnSuccessListener {
                location: Location? ->
                if (location != null) {
                    currentUserLocation =
                        LatLng(
                            location.latitude,
                            location.longitude
                        )
                }
            }
            continuation.invokeOnCancellation {
                cancellationTokenSource.cancel()
            }
        }
    }
}
```


Your code instantiates a `CancellationTokenSource` object to support request cancellations. It then requests the current location by calling `getCurrentLocation` from within a coroutine. The lambda function passed as an `OnSuccessListener` interface implementation persists valid locations. Finally, on cancellation of the coroutine, it cancels the request.

13. Update the `onResult` block of `requestLocationPermissionLauncher` to call the new `getUserLocation` function if permission is granted:

```
onResult = { permissions ->
    locationPermissionsGranted = ...
    if (locationPermissionsGranted) {
        getUserLocation()
    } else {
        shouldShowLocationPermissionRationale =
            shouldShowLocationPermissionRationale()
    }
}
```

14. Update the button label to **Get location** and include the value of the current user location:

```
Button(...) {
    Text(text = "Get location (${currentUserLocation})")
}
```

15. Update the `onClick` block of the button to call `getUserLocation` if permission is granted, too:

```
onClick = {
    if (locationPermissionsGranted) {
        getUserLocation()
    } else {
        shouldShowLocationPermissionRationale =
            shouldShowLocationPermissionRationale()
    }
    if (!locationPermissionsGranted &&
        !shouldShowLocationPermissionRationale
    ) {
        requestLocationPermission()
    }
}
```

16. Now, add a `cameraPositionState` variable and a `GoogleMap` composable to your layout before the `Button` composable:

```
val cameraPositionState = rememberSaveable(
    currentUserLocation,
    saver = CameraPositionState.Saver
) {
    CameraPositionState(
        position = CameraPosition
            .fromLatLngZoom(currentUserLocation, 10f)
    )
}
GoogleMap(
    modifier = Modifier.padding(innerPadding),
    cameraPositionState = cameraPositionState
)
Button(...)
```

17. Finally, add a `markerState` variable followed by a conditional `Marker` composable to the `GoogleMap` composable:

```
GoogleMap(...) {
    val markerState = rememberSaveable(
        currentUserLocation,
        saver = MarkerState.Saver
    ) {
        MarkerState(position = currentUserLocation)
    }
    if (markerState.position.latitude != 0.0 &&
        markerState.position.longitude != 0.0
    ) {
        Marker(state = markerState, title = "You")
    }
}
```

18. Run your app and tap the **Get location** button. Assuming you've already granted the location permission to the app, it should look like *Figure 7.7*.

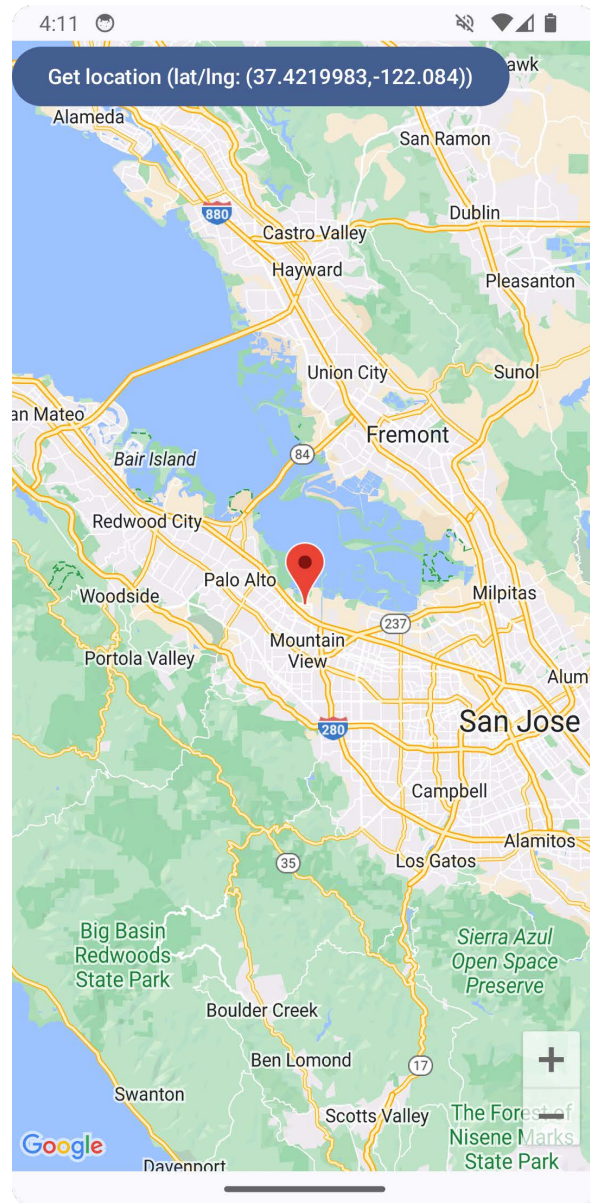


Figure 7.7 – Interactive map with a marker at the current location

19. Once the app has been granted permission, it can request the user's current location from the Google Play location service via the fused location provider client. This gives you an easy way to fetch the user's current location. Remember to turn on the location on your device for the app to work.

20. With the user's location, your app can tell the map where to zoom and where to place a marker. If the user clicks on the marker, they will see the title you assigned to it (*you* in this exercise).

In the next section, we will learn how to respond to clicks on the map and how to move markers.

Map clicks and custom markers

With a map showing the user's current location by zooming in at the right location and placing a marker there, we have rudimentary knowledge of how to render the desired map and how to obtain the required permissions and the user's current location.

In this section, we will learn how to respond to a user interacting with the map and how to use markers more extensively. We will learn how to move markers on the map and replace the default pin marker with custom icons. When we know how to let the user place a marker anywhere on the map, we can let them choose where to deploy the secret cat agent.

To handle clicks on the map, we pass a listener to the `GoogleMap` composable. A simple implementation might look like this:

```
GoogleMap(  
    modifier = Modifier.padding(innerPadding),  
    cameraPositionState = cameraPositionState,  
    onMapClick = { latLng ->  
        Toast.makeText(  
            this, "Clicked at $latLng", LENGTH_SHORT  
        ).show()  
    }  
)
```

To control the position of the marker on the map, we can replace the toast by updating the position value of the `markerState` variable. This also means that the `markerState` variable should now be declared outside of the `GoogleMap` composable. With this change, the marker will automatically move to the location of each tap on the map.

To replace the default pin icon with a custom icon, we must provide a `BitmapDescriptor` object to the marker or the `MarkerOptions()` instance. The `BitmapDescriptor` classes are wrappers that work around bitmaps used by `GoogleMap` to render markers and ground overlays, but we won't cover that in this book.

We obtain `BitmapDescriptor` by using `BitmapDescriptorFactory`. The factory will require an asset, which can be provided in a few ways. You can provide it with the name of a bitmap in the assets directory, a `Bitmap` object, the filename of a file in the internal storage, or a resource ID.

The factory can also create default markers of different colors. We are interested in the `Bitmap` option because we intend to use a vector drawable, and the factory does not directly support those. In addition, when converting the drawable to a `Bitmap` object, we can manipulate it to suit our needs (for example, we could change its color).

Android Studio offers us quite a wide range of free vector drawables out of the box. For this example, we want the paw drawable. To do this, right-click on the `res` directory in the left Android pane and select **New | Vector Asset**.

Now, click the Android icon next to the **Clip art** label for a list of icons (see *Figure 7.8*):

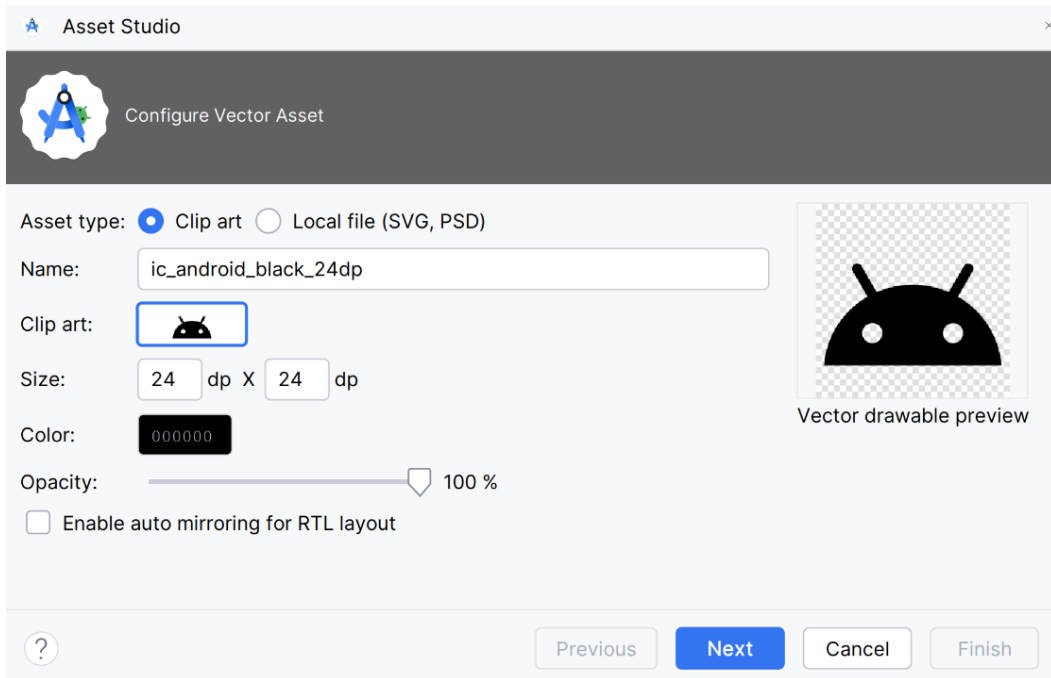


Figure 7.8 – Asset Studio

We'll now access a window to choose from the offered pool of clip art (*Figure 7.9*):

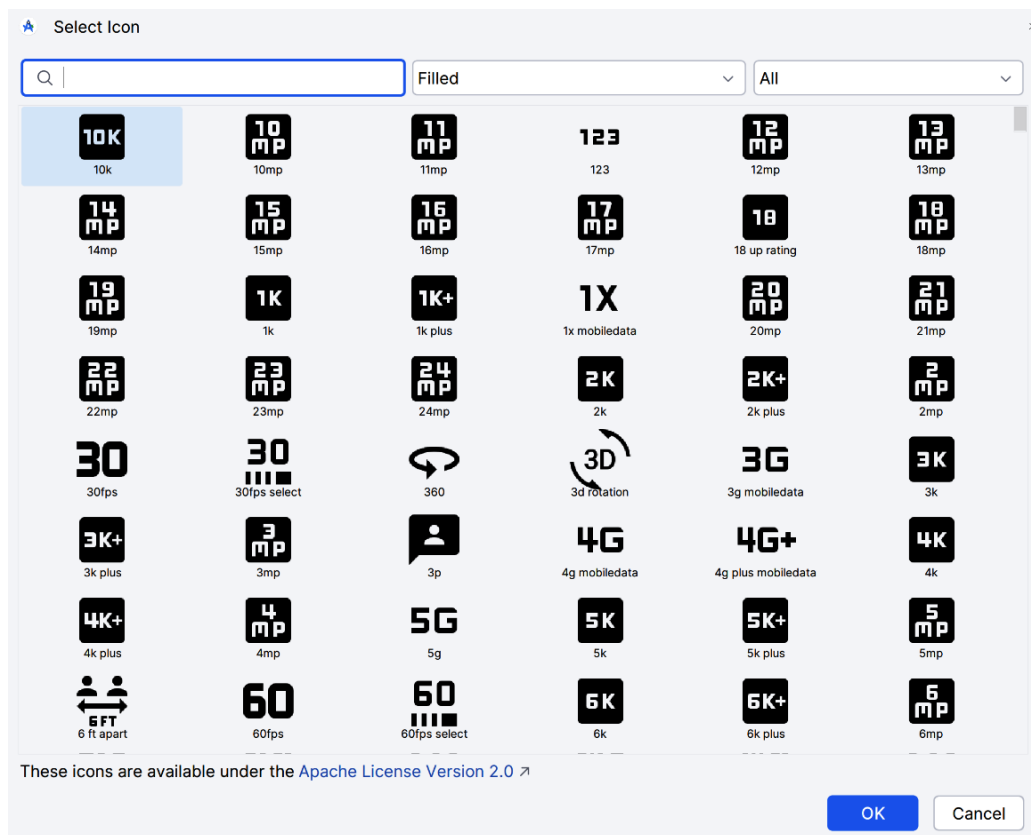


Figure 7.9 – Selecting an icon

 **Quick tip:** Need to see a high-resolution version of this image? Open this book in the next-gen Packt Reader or view it in the PDF/ePub copy.

 **The next-gen Packt Reader** and a **free PDF/ePub copy** of this book are included with your purchase. Scan the QR code OR visit packtpub.com/unlock, then use the search bar to find this book by name. Double-check the edition shown to make sure you get the right one.



Once we choose an icon, we can name it, and it will be created for us as a vector drawable XML file. We will name it `target_icon`.

Note that, despite the support for resource IDs, `BitmapDescriptorFactory` does not support vector drawables. And so, to use the created asset, we must first get it as a `Drawable` instance. This is done by calling `ContextCompat.getDrawable(Context, Int)`, passing in the activity and `R.drawable.target_icon` as a reference to our asset. Next, we need to define bounds for the `Drawable` instance to draw in.

Calling `Drawable.setBounds(Int, Int, Int, Int)` with `(0, 0, drawable.intrinsicWidth, drawable.intrinsicHeight)` will tell the drawable to draw within its intrinsic size.

To change the color of our icon, we can tint it. To tint a `Drawable` instance in a way that is supported by devices running APIs older than 21, we must first wrap our `Drawable` instance with `DrawableCompat` by calling `DrawableCompat.wrap(Drawable)`. The returned `Drawable` instance can then be tinted using `DrawableCompat.setTint(Drawable, Int)`.

Next, we need to create a `Bitmap` object to hold our icon. Its dimensions should match those of the `Drawable` bounds, and we want its config argument to be `Bitmap.Config.ARGB_8888`. This is done for us now when we use the `createBitmap` Kotlin extension function.



ARGB stands for **alpha**, **red**, **green**, and **blue**. **8** indicates the number of bits per channel. `ARGB_8888` means we want 8-bit red, green, blue, and alpha (transparency) channels.

We then create a canvas for the `Bitmap` object, allowing us to draw our `Drawable` instance by calling `Drawable.draw(Canvas)`. With the `Bitmap` object containing our icon, we are now ready to obtain a `BitmapDescriptor` instance from `BitmapDescriptorFactory`. Don't forget to recycle your `Bitmap` object afterward. This will avoid a memory leak.

Let's wrap it all together into a private function on `MainActivity`:

```
private fun getBitmapDescriptorFromVector(
    @DrawableRes vectorDrawableResourceId: Int
): BitmapDescriptor? = ContextCompat.getDrawable(
    this,
    vectorDrawableResourceId
)?.let { vectorDrawable ->
    vectorDrawable.setBounds(
        0,
```

```
        0,
        vectorDrawable.intrinsicWidth,
        vectorDrawable.intrinsicHeight
    )
    val drawableWithTint = DrawableCompat.
wrap(vectorDrawable)
DrawableCompat.setTint(drawableWithTint, Color.RED)
    val bitmap = createBitmap(
        vectorDrawable.intrinsicWidth,
        vectorDrawable.intrinsicHeight
    )
    val canvas = Canvas(bitmap)
    drawableWithTint.draw(canvas)
    BitmapDescriptorFactory.fromBitmap(bitmap)
        .also { bitmap.recycle() }
}
```

You have learned how to present the user with a meaningful map by centering it on their current location, responding to taps, and showing their current location using custom markers.

Exercise 7.03 – adding a custom marker where the map was clicked

In this exercise, you will respond to a user's map click by placing a red paw-shaped marker at the location on the map the user clicked:

1. In `MainActivity.kt` (found under `app/src/main/java/com/example/catagentdeployer`), move `markerState` outside of the `GoogleMap` composable (place it right before the `GoogleMap` call).
2. Update the `GoogleMap` call to handle map clicks:

```
GoogleMap(
    modifier = Modifier.padding(innerPadding),
    cameraPositionState = cameraPositionState,
    onMapClick = { latLng ->
        markerState.position = latLng
    }
)
```


3. Create a `getBitmapDescriptorFromVector(Int): BitmapDescriptor?` function right before the closing curly bracket of the `MainActivity` class to provide `BitmapDescriptor` with a `Drawable` resource ID:

```
private fun getBitmapDescriptorFromVector(
    @DrawableRes vectorDrawableResourceId: Int
): BitmapDescriptor? = ContextCompat.getDrawable(
    this,
    vectorDrawableResourceId
)?.let { vectorDrawable ->
    vectorDrawable.setBounds(
        0,
        0,
        vectorDrawable.intrinsicWidth,
        vectorDrawable.intrinsicHeight
    )
    val drawableWithTint =
        DrawableCompat.wrap(vectorDrawable)
        DrawableCompat.setTint(drawableWithTint, Color.DKGRAY
    )
    val bitmap = Bitmap.createBitmap(
        vectorDrawable.intrinsicWidth,
        vectorDrawable.intrinsicHeight,
        Bitmap.Config.ARGB_8888
    )
    val canvas = Canvas(bitmap)
    drawableWithTint.draw(canvas)
    BitmapDescriptorFactory.fromBitmap(bitmap)
        .also { bitmap.recycle() }
}
```

4. Before you can use the `Drawable` instance, you must first create it. Right-click on the `Android` pane, and then select **New | Vector Asset**.
5. In the window that opens, click on the `Android` icon next to the **Clip art** label to select a different icon (see *Figure 7.8*).

6. From the list of icons, select the **pets** icon. You can type **pets** into the search field if you can't find the icon. Once you select the **pets** icon (see *Figure 7.10*), click **OK**.

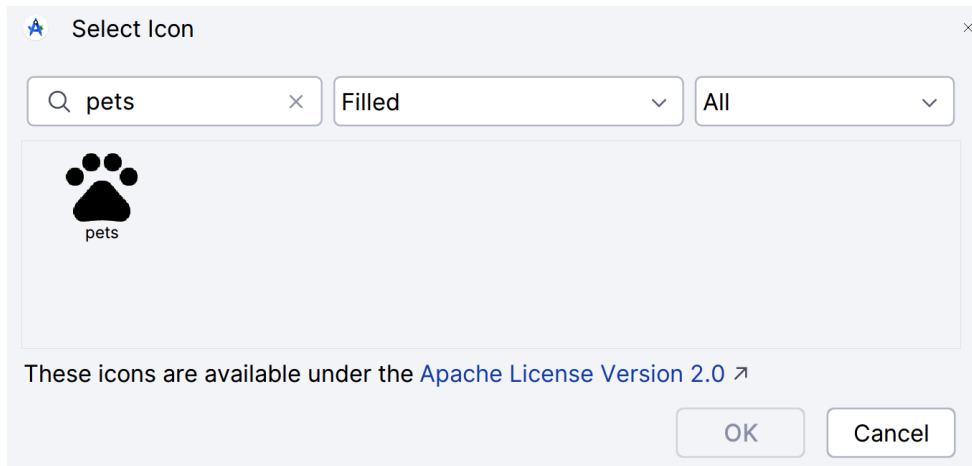


Figure 7.10 – Selecting an icon

7. Name your icon `target_icon`. Click **Next** and **Finish**.
8. Before the call to the `Marker` composable, declare an icon variable to load the icon you just created. Wrap it in `remember` to avoid recreating it on every composition:

```
val icon by remember {
    mutableStateOf(
        getBitmapDescriptorFromVector(
            R.drawable.target_icon
        )
    )
}
```

9. Finally, update the `Marker` call to include an icon. Also, update the title of the `Marker` object to "Deploy Here":

```
Marker(
    state = markerState,
    title = "Deploy Here",
    icon = icon
)
```

10. Run your app and click on the **Get location** button. It should look like *Figure 7.11*.

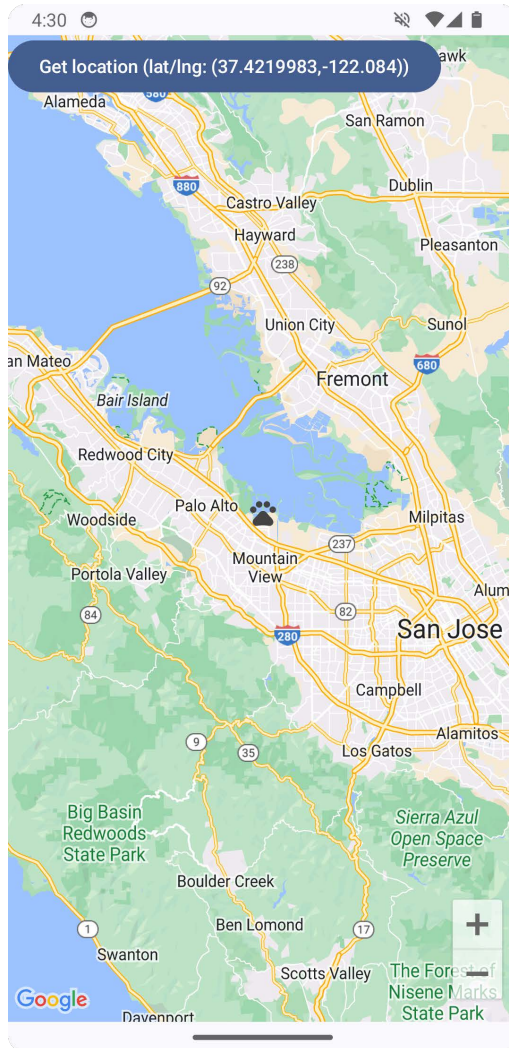


Figure 7.11 – The complete app

11. Tapping anywhere on the map will move the paw icon to that location. Tapping the paw icon will show the **Deploy here** label and center the map on the icon.



The location of the paw is a geographical one, not a screen one. That means if you drag your map or zoom in, the paw will move with the map and remain in the same geographical location.

You now know how to respond to user taps on the map and how to add and move markers around. You also know how to customize the appearance of markers.

In the following activity, you will use the knowledge you acquired to develop an app that helps you remember where you parked your car.

Activity 7.01 – creating an app to find the location of a parked car

Some people often forget where it was that they parked their car. Let's say you want to help these people by developing an app that lets the user store the last place they parked. The app will show a pin at the car's location when the user launches the app. The user can click an **I'm parked here** button to update the pin location to the current location the next time they park.

Your goal in this activity is to develop an app that shows the user a map of the current location. The app must first ask the user for permission to access their location. Make sure to also provide a rationale dialog, if needed, according to the SDK.

The app will show a car icon where the user last told it the car was. The user can click a button labeled **I'm parked here** to move the car icon to the current location. When the user relaunched the app, it will show the user's current location and the car icon where the car was last parked.

As a bonus feature of your app, you can choose to add functionality that stores the car's location so that it can be restored after the user has killed and then reopened the app. This bonus functionality relies on using `SharedPreferences`, a concept that will be covered in *Chapter 12, Persisting Data*. As such, *steps 9 and 10* here will give you the required implementation.

The following steps will help you complete the activity:

1. Create an empty activity app.
2. Obtain a Google Maps API key for the app.
3. Add the `secret.s` plugin and set it up with the key you obtained in the last step.
4. Add the **Play Services Location** and **Google Maps Compose** dependencies.
5. Include a `GoogleMap` composable in your layout.
6. Add a marker to the map. Label it `My Car`.
7. Introduce a button at the bottom of the screen with an **I'm parked here** label.
8. Include the fused location client in your app.
9. Implement code to read the current user's location.

10. Add logic to request the user's permission to access their location. Present the rationale if needed.
11. When the button is clicked, check the location permission, request it if needed, and move the marker to the user's current location.
12. Add a car icon to your project.
13. Apply the car icon to the marker.
14. Add functionality to move the car icon by tapping on the map.
15. Bonus step: Store the selected location in SharedPreferences. This function, placed in your activity, will help you do this:

```
private fun saveLocation(latLng: LatLng) =
    getPreferences(MODE_PRIVATE)?.edit()?.apply {
        putString(
            "latitude", latLng.latitude.toString()
        )
        putString(
            "longitude", latLng.longitude.toString()
        )
        apply()
    }
```

16. Bonus step: Restore any saved location from SharedPreferences. You can use the following function:

```
val latitude = sharedPreferences
    .getString("latitude", null)
    ?.toDoubleOrNull()
    ?: return null
val longitude = sharedPreferences
    .getString("longitude", null)
    ?.toDoubleOrNull()
    ?: return null
```

With this activity completed, you have demonstrated your understanding of requesting permissions in an Android app. You have also shown that you can present the user with a map and control pins on that map. Finally, you have also demonstrated your knowledge of obtaining the user's current location. Well done!



The solution to this activity can be found at <https://packt.link/TQf3V>.

Summary

In this chapter, we learned about Android permissions. We touched on the reasons for having them and saw how we can request the user's permission to perform certain tasks. We also learned how to use Google's Maps API and how to present the user with an interactive map.

Lastly, we leveraged our knowledge of presenting a map and requesting permissions to find out the user's current location and present it on the map. Of course, there is a lot more that can be done with the Google Maps API, and you could explore a lot more possibilities with certain permissions.

You should now have enough understanding of the foundations to explore further. To read more about permissions, visit <https://packt.link/57BdN>. To read more about the Maps API, visit <https://packt.link/8akrP>.

In the next chapter, we will learn how to perform background tasks using `Services` and `WorkManager`. We will also learn how to present the user with notifications, even when the app is not running. These are powerful tools to have in your arsenal as a mobile developer.

Unlock this book's exclusive benefits now

Scan this QR code or go to packtpub.com/unlock, then search this book by name.

Note: Keep your purchase invoice ready before you start.

UNLOCK NOW



8

Services, WorkManager, and Notifications

In the previous chapter, we learned how to request permissions from the user and use Google's Maps API. With that knowledge, we obtained the user's location and allowed them to deploy an agent on a local map. In this chapter, we will learn how to track a long-running process and report its progress to the user.

Ongoing background tasks are quite common in the mobile world. Background tasks run even when an application is not active. Examples of long-running background tasks include the downloading of files, resource clean-up jobs, playing music, and tracking the user's location.

This chapter will introduce you to the concepts of managing long-running tasks in the background of an app. We will cover the two primary mechanisms that Android provides for managing such tasks: services and workers. **Services** are application components designed to run in the background, even when an app is not running. They have no user interface, though foreground services are an exception. **Workers** are classes that can be scheduled to execute tasks in the background based on different rules that are managed by WorkManager.

By the end of this chapter, you will be able to trigger a background task, create a notification for the user when a background task is complete, and launch an application from a notification. This chapter will give you a solid understanding of how to manage background tasks and keep the user informed about the progress of these tasks.

Let's get started and look at the implementation of the multiple approaches available in Android for managing a background task.

We will cover the following topics in this chapter:

- Starting a background task using WorkManager
- Background operations noticeable to the user – using foreground workers
- Canceling or updating ongoing work

Technical requirements

The complete code for all the exercises and the activity in this chapter is available on GitHub at <https://packt.link/9MVrd>.

In this chapter, we will build an example app where we will assume that **Secret Cat Agents (SCAs)** get deployed in a record time of 15 seconds. When a cat successfully deploys, we will notify the user and let them launch the app, presenting them with a successful deployment message.



We will go with 15 seconds because that way, we will avoid having to wait for very long before our background task completes.

Starting a background task using WorkManager

Historically, Google offered Android developers multiple ways of executing such tasks: services, JobScheduler, and Firebase's JobDispatcher and AlarmManager. With the fragmentation that existed in the Android world, it was quite a mess to cope with. Luckily for us, since March 2019, we have had a better (more stable) option.

With the introduction of `WorkManager`, Google has abstracted the logic of choosing a background execution mechanism based on the API version away from us. `WorkManager` works on Android, going all the way back to API 14 (Android 4.0, or Ice Cream Sandwich). We still use a foreground service, which is a special kind of service, for certain tasks that should be known to the user while running, such as playing music or tracking the user's location in a running app.

The first question we will address in this section is as follows: Should we opt for `WorkManager` or a foreground service? To answer that, a good rule of thumb is to ask whether you need the action to be tracked by the user in real time.

If the answer is yes (for example, if you have a task such as responding to the user's location or playing music in the background), then you should use a foreground service with its attached notification to give the user a real-time indication of state. When the background task can be delayed or does not require user interaction (for example, downloading a large file), use `WorkManager`.



Starting with version 2.3.0-alpha02 of `WorkManager`, you can launch a foreground service via the `WorkManager` singleton by calling `setForegroundAsync(ForegroundInfo)`.

Let's say we wanted to create an app that scans the user's device for duplicate files to help them save space. Before we can generate a report, we must scan the filesystem for such files. This task may take a while. `WorkManager` is perfect for such a scenario.

To use `WorkManager`, we need to familiarize ourselves with its four main classes (see *Figure 8.1*):

- **WorkManager:** This receives work and enqueues it based on provided arguments and constraints (such as internet connectivity and whether the device is charging).
- **Worker:** This is a wrapper around the work that needs doing. It has one public function we need to implement: `doWork()`. In it, we perform any background work. The `doWork()` function will be executed in a background thread for us. A special kind of `Worker` class is `CoroutineWorker`, which has a `doWork()` suspend function and allows us to leverage coroutines in our work.
- **WorkRequest:** This class binds a `Worker` class to arguments and constraints. There are two `WorkRequest` types: `OneTimeWorkRequest`, which runs the work once, and `PeriodicWorkRequest`, which can be used to schedule work to run at a fixed interval.

- `ListenableWorker.Result`: You have probably guessed it, but this is the class holding the result of the executed work. The result can be one of `Success`, `Failure`, or `Retry`.

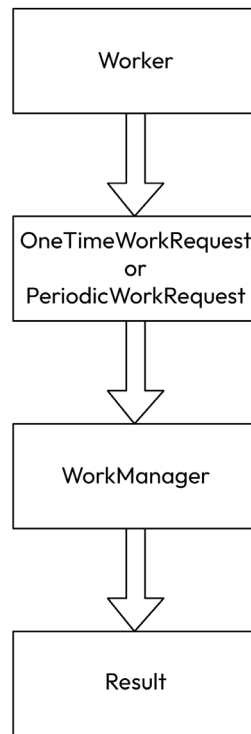


Figure 8.1 – The classes involved in executing work

Other than these four classes, we also have the `Data` class, which holds data passed to and from the worker.



Work can be immediate, long-running, or deferrable. You can read more about the different types of work and `WorkManager` at <https://packt.link/Gc9jc>.

Let's get back to our example. We want to define two tasks that need to occur in sequential order: first, we want to list all the files in the filesystem. Then, we want to scan them for duplicates.

Before we can start using `WorkManager`, we have to first include its dependency in our app:

```
androidx-work-runtime = {  
    group = "androidx.work",  
    name = "work-runtime",
```

```
        version.ref = "androidxWorkRuntime"  
    }
```

With `WorkManager` included in our project, we'll go ahead and create our workers. The first worker will look something like this:

```
class FileIndexingWorker(  
    context: Context,  
    workerParameters: WorkerParameters  
) : Worker(context, workerParameters) {  
    override fun doWork(): Result {  
        val fileType = inputData.getString(INPUT_DATA_TYPE)  
        [Save the list of files somewhere and update filesCount]  
        val outputData = workDataOf(  
            OUTPUT_DATA_FILES to filesCount  
        )  
        return Result.success(outputData)  
    }  
    companion object {  
        const val INPUT_DATA_TYPE = "type"  
        const val OUTPUT_DATA_FILES = "files"  
    }  
}
```

We start by extending `Worker` and overriding its `doWork()` function. We then read the file type we're interested in from the input data. Then (this section is omitted from the code for brevity), we scan the filesystem for files matching the provided type and store their paths in a string array. Finally, we construct an `outputData` class with the count of files we found and return it with the successful result.

Once we've created a worker for `DuplicateFindingWorker`, following the implementation of `FileIndexingWorker`, we can call `WorkManager` to chain the workers. Let's also assume we don't want to run unless the device is charging because the work may drain the user's battery. Our call would look something like this:

```
val chargingConstraints = Constraints.Builder()  
    .setRequiresCharging(true)  
    .build()  
val fileIndexingInputData = Data.Builder()  
    .putString(FileIndexingWorker.INPUT_DATA_TYPE, "jpg")
```

```

        .build()
    val fileIndexingRequest =
        OneTimeWorkRequestBuilder<FileIndexingWorker>()
    val duplicateFindingRequest =
        OneTimeWorkRequestBuilder<DuplicateFindingWorker>()
            .setConstraints(chargingConstraints)
            .setInputData(catStretchingInputData)
            .build()
    WorkManager.getInstance(this)
        .beginWith(fileIndexingRequest)
        .then(duplicateFindingRequest)
        .enqueue()

```

We first construct a `Constraints` instance, declaring that we need the device to be charging for the work to execute. We then define our input data, setting the file type to "jpg". Next, we bind the constraints and input data to our `Worker` class by constructing a `OneTimeWorkRequest` object. We do this for both `Worker` instances.

We can now chain all the requests and enqueue them in the `WorkManager` class. You can enqueue a single `WorkRequest` instance by passing it directly to the `enqueue()` function of `WorkManager`, or you can have multiple `WorkRequest` instances run in parallel by passing them all to the `enqueue()` function of `WorkManager` as a list. If called from within a coroutine, the work can be blocked until completion by calling `await()` after `enqueue`.

Our tasks will be executed by `WorkManager` when the constraints are met.

Each `Request` instance has a unique identifier. `WorkManager` exposes a `Flow` coroutine for each request, allowing us to track the progress of its work by passing its unique identifier, as shown in the following code:

```

workManager.getWorkInfoByIdFlow(
    fileIndexingRequest.id
).collect { info ->
    if (info?.state?.isFinished == true) { doSomething() }
}

```

We first check for nullability because if the work manager fails to find the work with the requested ID, it will set the `info` variable to `null`. If the work is found, the state of work can be `RUNNING` (the work in `doWork()` is executing) and `SUCCEEDED`. Work can be canceled, leading to a `CANCELLED`

state, or it can fail, leading to a `FAILED` state. If there is a chain of requests, the state can also be `BLOCKED` (this work is not next in the chain) or `ENQUEUED` (this work is next). Note that `isFinished` can be true even if the work failed or was canceled.

Finally, there's `Result.retry`. Returning this result from the worker's `doWork()` function tells the `WorkManager` class to enqueue the work again. The policy governing when to run the work again is defined by a backoff criterion set on `WorkRequest.Builder`. The default backoff policy is exponential, but we can set it to be linear instead. We can also define the initial backoff time.

The `WorkInfo` object we collect can contain another useful piece of information: `stopReason` can tell us why a worker was stopped.

Let's put the knowledge gained so far into practice in the following exercise. In this section, we will track our SCA from the moment we fire off the command to deploy it to the field to the moment it arrives at its destination.

Exercise 8.01 – Executing background work with the `WorkManager` class

In this first exercise, we will track the SCA as it prepares to head out by enqueueing the chained `WorkRequest` classes. Before an agent can head out, they need to stretch, groom their fur, visit the litter box, and suit up. Each one of these tasks takes some time. Since you can't rush a cat, the agent will finish each step in its own time. All we can do is wait (and let the user know when the task is done):

1. Start by creating a new **Empty Activity** project (**File** | **New** | **New Project** | **Empty Activity**). Click **Next**.
2. Name your application `Cat Agent Tracker`.
3. Make sure your package name is `com.example.catagenttracker`.
4. Set the save location to where you want to save your project.
5. Leave everything else at its default values and click **Finish**.
6. Make sure you are on the Android view in your **Project** pane.
7. Add the latest version of the `WorkManager` dependency to your version catalog and app module `build.gradle.kts` file:

```
androidx-work-runtime = {  
    group = "androidx.work",  
    name = "work-runtime",  
    version.ref = "androidxWorkRuntime"  
}
```

This will allow you to use WorkManager and its dependencies in your code.

8. Create a new package under your app package (right-click on **com.example.catagenttracker**, then **New | Package**). Name the new package `com.example.catagenttracker.worker`.
9. Create a new class under `com.example.catagenttracker.worker` named `CatStretchingWorker` (right-click on **worker**, then **New | New Kotlin File/Class**). Fill in the class name and double-click on **Class**.
10. To define a Worker instance that will sleep for 3 seconds, update the new class like so:

```
class CatStretchingWorker(
    context: Context,
    workerParameters: WorkerParameters
) : Worker(context, workerParameters) {
    override fun doWork(): Result {
        val catAgentId = inputData
            .getString(INPUT_DATA_CAT_AGENT_ID)
        Thread.sleep(3000L)
        val outputData = workDataOf(
            OUTPUT_DATA_CAT_AGENT_ID to catAgentId
        )
        return Result.success(outputData)
    }

    companion object {
        const val INPUT_DATA_CAT_AGENT_ID = "inId"
        const val OUTPUT_DATA_CAT_AGENT_ID = "outId"
    }
}
```

This will add the required dependencies for a Worker implementation and then extend the Worker class. To implement the actual work, you will override `doWork(): Result`. First, you will make it read the cat agent ID from the input. Then, you will have it sleep for 3 seconds (3,000 milliseconds). Finally, it will construct an output data instance with the cat agent ID and pass it via a `Result.success` value.

11. Repeat *steps 9 and 10* to create three more identical workers named `CatFurGroomingWorker`, `CatLitterBoxSittingWorker`, and `CatSuitUpWorker`.

12. At the top of the MainActivity class, declare the workManager variable:

```
private val workManager =  
    WorkManager.getInstance(this)
```

13. Add the following to the onCreate(Bundle?) function, right after the super.onCreate(Bundle?) function call:

```
val networkConstraints = Constraints.Builder()  
    .setRequiredNetworkType(NetworkType.CONNECTED)  
    .build()  
val catAgentId = "CatAgent1"  
val catStretchingRequest = OneTimeWorkRequestBuilder  
    <CatLitterBoxSittingWorker>()  
    .setConstraints(networkConstraints)  
    .setInputData(  
        workDataOf(  
            CatStretchingWorker  
                .INPUT_DATA_CAT_AGENT_ID to catAgentId  
        )  
    ).build()  
val catFurGroomingRequest =  
    OneTimeWorkRequestBuilder<CatFurGroomingWorker>()  
    .setConstraints(networkConstraints)  
    .setInputData(  
        workDataOf(  
            CatFurGroomingWorker  
                .INPUT_DATA_CAT_AGENT_ID to catAgentId  
        )  
    ).build()  
val catLitterBoxSittingRequest =  
    OneTimeWorkRequestBuilder  
    <CatLitterBoxSittingWorker>()  
    .setConstraints(networkConstraints)  
    .setInputData(  
        workDataOf(  
            CatLitterBoxSittingWorker  
                .INPUT_DATA_CAT_AGENT_ID to catAgentId  
        )  
    )
```



```

        ).build()
    val catSuitUpRequest =
        OneTimeWorkRequestBuilder<CatSuitUpWorker>()
            .setConstraints(networkConstraints)
            .setInputData(
                workDataOf(
                    CatSuitUpWorker
                        .INPUT_DATA_CAT_AGENT_ID to catAgentId
                )
            ).build()

```

The first line added defines a network constraint. It tells the `WorkManager` class to wait for an internet connection before executing work. Then, you define your cat agent ID. Finally, you define four requests, passing in the `Worker` classes, the network constraints, and the cat agent ID in the form of input data.

14. Add a chained enqueue request right below the code you just added, still within the `onCreate` function:

```

    val catSuitUpRequest = ...
    workManager.beginWith(catStretchingRequest)
        .then(catFurGroomingRequest)
        .then(catLitterBoxSittingRequest)
        .then(catSuitUpRequest)
        .enqueue()

```

Your `WorkRequest` instances are now enqueued to be executed in sequence when their constraints are met and the `WorkManager` class is ready to execute them.

15. Define a function at the end of the `MainActivity` class to show a toast with a provided message. It should look like this:

```

    private fun showResult(message: String) {
        Toast.makeText(this, message, LENGTH_SHORT).show()
    }

```

16. To track the progress of the enqueued `WorkRequest` instances, declare a private `collectWorkInfo(UUID, String)` suspend function in the `MainActivity` class:

```

    private suspend fun collectWorkInfo(
        requestId: UUID,

```

```

        message: String
    ) {
        workManager.getWorkInfoByIdFlow(requestId)
            .collect { info ->
                if (info?.state?.isFinished == true) {
                    showResult(message)
                }
            }
    }
}

```

The preceding code collects `WorkInfo` updates from the `WorkManager` class for each `WorkRequest` object. When each request is finished, a toast is shown with a relevant message.

17. Call `collectWorkInfo` for each request. To start collecting as soon as the activity is composed, use `LaunchedEffect`:

```

CatAgentTrackerTheme {
    LaunchedEffect(true) {
        collectWorkInfo(
            catStretchingRequest.id,
            "Agent done stretching"
        )
    }
    LaunchedEffect(true) {
        collectWorkInfo(
            catFurGroomingRequest.id,
            "Agent done grooming its fur"
        )
    }
    LaunchedEffect(true) {
        collectWorkInfo(
            catLitterBoxSittingRequest.id,
            "Agent done sitting in litter box"
        )
    }
    LaunchedEffect(true) {
        collectWorkInfo(
            catSuitUpRequest.id,

```

```
}  
    "Agent done suiting up. Ready to go!")
```

18. Run your app. It should go through the steps shown in *Figure 8.2*:

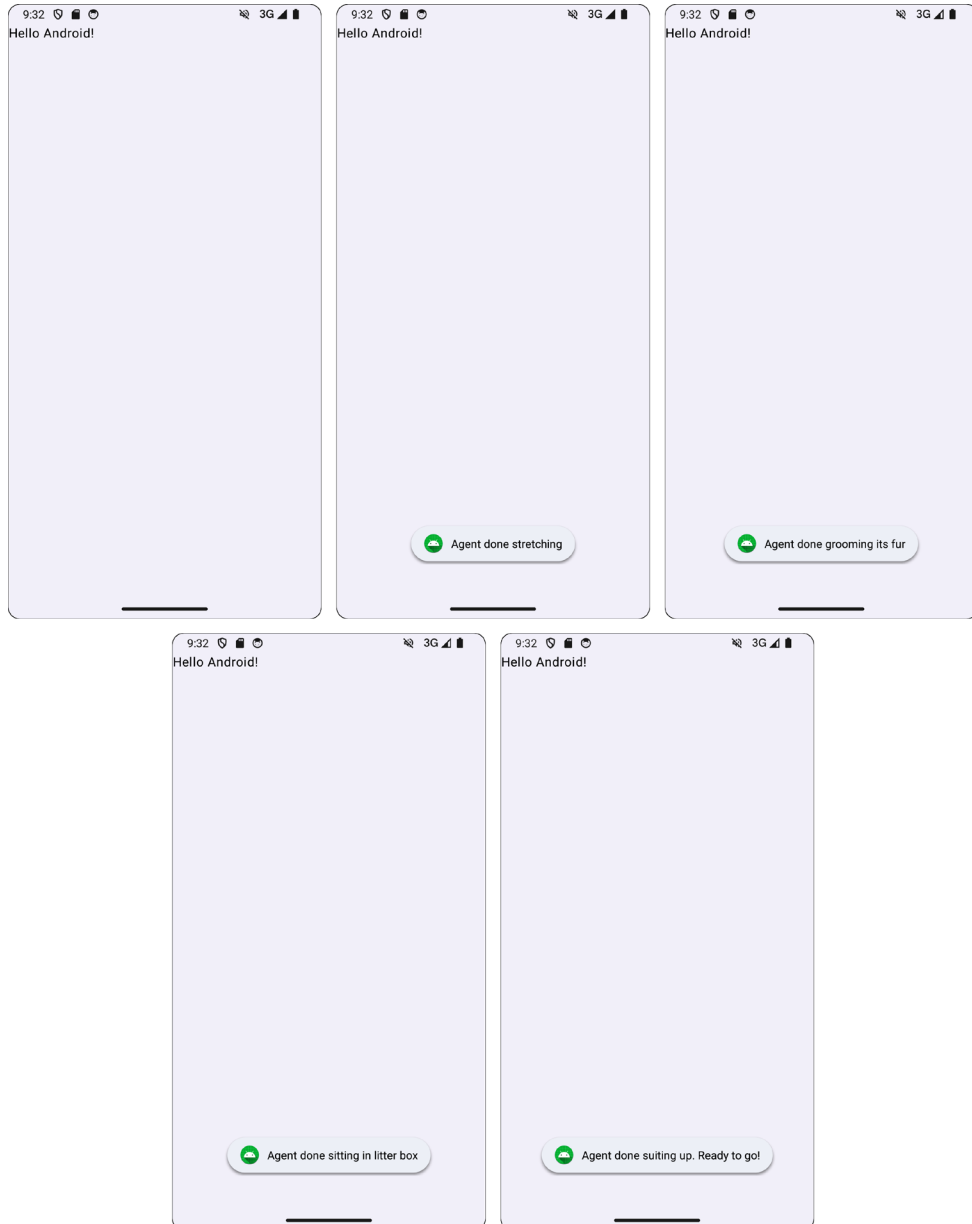


Figure 8.2 – Toasts showing in order

You should now see a simple **Hello Android!** screen. However, if you wait a few seconds, you will start seeing toasts informing you of the progress of your SCA preparing to deploy to the field. You will notice that the toasts follow the order in which you enqueued the requests and execute sequentially.

In the next section, we will explore a different aspect of long-running operations: a type of operation that is user-facing, or foreground-running.

Background operations noticeable to the user – using foreground workers

Foreground workers are another way of performing background operations. The name is meant to differentiate these workers from the base (background) workers. The former are tied to a notification, while the latter run in the background with no user-facing representation built in.

Foreground workers are designed to execute work that is expected to run for longer than 10 minutes and should not be interrupted. Under the hood, `WorkManager` uses a foreground service to execute the work.

Different versions of Android have different permission requirements for foreground work, with a general trend of reducing the number of notifications disturbing the user:

- For Android 9 (Pie, or API level 28), we have to request the `FOREGROUND_SERVICE` permission to use foreground services. Since it is a normal permission, it will be granted to our app automatically once declared in the manifest.
- From Android 10 (API level 29) onward, we must be more specific if we want to access the location of the device and obtain the `FOREGROUND_SERVICE_LOCATION` permission.
- For Android 11 (API level 30) and above, we must specify whether our long-running work requires access to the camera or the microphone by requesting the `FOREGROUND_SERVICE_CAMERA` or `FOREGROUND_SERVICE_MICROPHONE` permissions.
- Since Android 13, Tiramisu (API level 33), things have gotten even harder. For devices running this version or newer, we must now actively request the `POST_NOTIFICATIONS` permission in addition to having the foreground service permission in our manifest file.
- As of API level 34, requirements were hardened even further. Specifying the type of foreground service that we intend to run is no longer optional. We must request specific permissions for each type instead of the generic one. The options include `FOREGROUND_SERVICE_CONNECTED_DEVICE`, `FOREGROUND_SERVICE_DATA_SYNC`, `FOREGROUND_SERVICE_HEALTH`, and others.

Before we can launch a foreground worker, we must first create a worker. A foreground worker is a worker that calls `setForeground(ForegroundInfo)`. And as you already know by now, a foreground worker must be tied to a notification. This is enforced via the `ForegroundInfo` instance, which defines the properties of the notification.

In the following section, we'll explore an example of creating a `ForegroundInfo` instance. First, we must sort out some prerequisites:

```
val channelId = if (
    Build.VERSION.SDK_INT >= VERSION_CODES.O
) {
    val newChannelId = "Channel ID"
    val channelName = "Channel Name"
    val channel = NotificationChannel(
        newChannelId, channelName,
        NotificationManager.IMPORTANCE_DEFAULT
    )
    val service = requireNotNull(
        ContextCompat.getSystemService(
            applicationContext,
            NotificationManager::class.java
        )
    )
    service.createNotificationChannel(channel)
    newChannelId
} else { "" }

val flag = if (
    Build.VERSION.SDK_INT >= Build.VERSION_CODES.S
) FLAG_IMMUTABLE else 0
val pendingIntent = Intent(
    this, MainActivity::class.java
).let { notificationIntent ->
    PendingIntent.getActivity(this, 0,
        notificationIntent, flag)
}
```

Let's break this down.

We start by defining the channel ID. This is only required for Android 8 (Oreo) or above and is ignored in earlier versions of Android. In Android Oreo, Google introduced the concept of channels. **Channels** are used to group notifications and allow users to filter out unwanted notifications.

Next, we define `pendingIntent`. This will be the `Intent` instance launched if the user taps on the notification. In this example, the main activity would be launched. It is constructed by wrapping an `Intent` activity in a `PendingIntent` object. The request code is set to `0` because this example doesn't expect a result, so the code will not be used.

The flag is set to `0` for APIs older than `S (31)`. Otherwise, it is set to the recommended `PendingIntent.FLAG_IMMUTABLE` constant, which means that additional arguments passed to the `Intent` object on `send` will be ignored.

With the channel ID and with `pendingIntent`, we can construct a notification. We use `NotificationCompat`, which takes away some of the boilerplate around supporting older API levels. We pass in the application context and the channel ID. We define the title, text, small icon, an `Intent` object, and ticker message, and build the notification. Note that since Android 13 (API level 33), we have to call `setOngoing()` as we do here to make the notification sticky. Prior to that, notifications were sticky by default:

```
val notification = NotificationCompat.Builder(  
    applicationContext, channelId  
)  
.setContentTitle("Content title")  
.setContentText("Content text")  
.setSmallIcon(  
    R.drawable.ic_launcher_foreground  
)  
.setContentIntent(pendingIntent)  
.setTicker("Ticker message")  
.setOngoing().build()
```

Finally, we are ready to construct the `ForegroundInfo` instance. We pass in a notification ID (any unique `int` value to identify this service, which must not be `0`) and the notification. Note that we check the Android version we are running. If it's Android Q or later, we must specify the foreground service type; otherwise, we can omit it:

```
val foregroundInfo = if (  
    Build.VERSION.SDK_INT >= VERSION_CODES.Q  
) {  
    ForegroundInfo(  
        NOTIFICATION_ID, notification,  

```

```

        FOREGROUND_SERVICE_TYPE_PHONE_CALL
    )
} else { ForegroundInfo(NOTIFICATION_ID, notification) }

```

To start a worker in the foreground, attaching the notification to it, we call `setForeground(ForegroundInfo)`, which we mentioned earlier, from within our `Worker` class.

When executed, our worker will now show a notification. Clicking on the notification will launch our main activity.

To add some functionality to our worker, we override the `doWork()` function. It is a good idea to set the worker to be a foreground worker before performing any operations within this function.

Communicating with a foreground worker is similar to working with a background one. Calling `getWorkInfoByIdFlow(UUID)` on a `WorkManager` instance allows us to process progress and state updates. To report progress updates from the worker, we can call `setProgress(Data)` at any point during our work. The flow will then include a `WorkInfo` update with the progress field containing the data we provided.

Before we can execute our worker, we must make sure we have added the foreground service to our `AndroidManifest.xml` file within the `<application></application>` block, as shown in the following code:

```

<application ...>
    ...
    <service
        android:name=
            "androidx.work.impl.foreground
                .SystemForegroundService"
        android:foregroundServiceType="phoneCall"
        tools:node="merge" />
    </application>

```

Note that the type of work (a phone call, in this example) must match the type specified in the `ForegroundInfo` instance. For the full list of foreground service types, see <https://packt.link/6oISH>. Setting `tools:node="merge"` tells the manifest merger to combine this service with the one defined by `WorkManager`.

We can launch the worker like we did before, using a request builder:

```

val workerRequest = OneTimeWorkRequestBuilder<MyWorker>()
    .setInputData(workDataOf(...))

```

```
.setExpedited(RUN_AS_NON_EXPEDITED_WORK_REQUEST)
    .build()
workManager.enqueue(workerRequest)
```

There are two differences between this code and the one we encountered earlier in the chapter, in *Starting a background task using WorkManager*.

The first difference is that we set the worker to be expedited. This tells the operating system that the work is important and helps ensure that it starts quickly rather than being deprioritized. If the expedited quota is reached, we tell the work manager to run it as normal work (`RUN_AS_NON_EXPEDITED_WORK_REQUEST`). We can choose to tell the work manager to drop the work if the quota is reached by passing in `DROP_WORK_REQUEST` instead. Expedited requests can only be fulfilled if the worker overrides `getForegroundInfo()` to return a `ForegroundInfo` instance.

The second, less significant difference is that you pass the request directly to the `enqueue(WorkRequest)` function. As mentioned in *Starting a background task using WorkManager*, this is possible because, in this example, we only have a single request.

Having a `WorkRequest` object, we can obtain a request ID via the `id` field, which will allow us to get updates, as we covered in *Starting a background task using WorkManager* earlier.

Exercise 8.02 – Tracking your SCA with a foreground worker

In the first exercise, you tracked the SCA as it was preparing to head out using the `WorkManager` class and multiple `Worker` instances showing toasts. With our SCA all suited up, they are now ready to get to the assigned destination. In this exercise, you will track the SCA as it deploys to the field and moves toward the assigned target by showing a sticky notification (a notification that cannot be dismissed by the user). You will periodically poll the location of the SCA using the foreground worker and update the notification attached to that worker, counting down the time to arrival at the destination.



For the sake of simplicity, we will fake the location. Following what you learned in *Chapter 7, Android Permissions and Google Maps*, you could later replace this implementation with a real one that uses a map.

This notification will be driven by a foreground service, which will present and continuously update it. The service will be managed by a foreground worker. Clicking the notification at any time will launch your main activity if it's not already running and will always bring it to the foreground:

1. Create a *Cat Agent Tracker* app by following steps 1–8 of *Exercise 8.01 – Tracking your SCA with a foreground worker*.

2. Within the `com.example.catagenttracker.worker` package, create a class called `RouteTrackingWorker`, extending the abstract `CoroutineWorker` class:

```
class RouteTrackingWorker(  
    context: Context,  
    parameters: WorkerParameters  
) : CoroutineWorker(context, parameters) {}
```

3. In the newly created worker, define some constants that you will need later:

```
companion object {  
    private const val NOTIFICATION_ID = 0xCA7  
    private const val INITIAL_SECONDS_LEFT = 10  
    const val DATA_KEY_CAT_AGENT_ID = "AgentId"  
    const val DATA_KEY_SECONDS_LEFT = "SecondsLeft"  
}
```

`NOTIFICATION_ID` must be a unique identifier for the notification owned by this worker and must not be 0. `INITIAL_SECONDS_LEFT` holds the duration you want your worker to run. `DATA_KEY_CAT_AGENT_ID` is the constant you will use to pass data to and from the worker. `DATA_KEY_SECONDS_LEFT` will be used to publish progress.

4. Add a function to your `RouteTrackingWorker` class to provide `PendingIntent` to your sticky notification:

```
private fun getPendingIntent(): PendingIntent {  
    val flag = if (  
        Build.VERSION.SDK_INT >= Build.VERSION_CODES.S  
    ) { FLAG_IMMUTABLE } else { 0 }  
    return PendingIntent.getActivity(  
        applicationContext, 0, Intent(  
            applicationContext,  
            MainActivity::class.java  
        ), flag  
    )  
}
```

This function returns a `PendingIntent` object that will launch `MainActivity` whenever the user clicks on your notification. You call `PendingIntent.getActivity()`, passing the application context, no request code (0), an `Intent` object that will launch `MainActivity`, and the `FLAG_IMMUTABLE` flag if available – otherwise, no flags (0).

5. Add another function to create a notification channel on devices running Android Oreo or newer, and return a channel ID:

```
private fun createNotificationChannel(): String = if (
    Build.VERSION.SDK_INT >= Build.VERSION_CODES.O
) {
    val newChannelId = "CatDispatch"
    val channelName = "Cat Dispatch Tracking"
    val channel = NotificationChannel(
        newChannelId, channelName,
        NotificationManager.IMPORTANCE_HIGH
    )
    val service = requireNotNull(
        ContextCompat.getSystemService(
            applicationContext,
            NotificationManager::class.java
        )
    )
    service.createNotificationChannel(channel)
    newChannelId
} else { "" }
```

You start by checking the Android version. You only need to create a channel if it's Android O or later. Otherwise, you can return an empty string. For Android O, you define the channel ID. This needs to be unique for a package.

Next, you define a channel name that will be visible to the user. This can (and should) be localized. We skipped that part for the sake of simplicity. A `NotificationChannel` instance is then created with the importance set to `IMPORTANCE_HIGH`. The importance dictates how disruptive the notifications posted to this channel are.

Lastly, a channel is created using the notification service with the data provided in the `NotificationChannel` instance. The function returns the channel ID so that it can be used to construct the `Notification` object.

At the top of the `RouteTrackingWorker` class, create a field to lazily create a channel and store its ID for repeated use. Create another field to store the current tracking progress, initialized to `INITIAL_SECONDS_LEFT`:

```
private val channelId by lazy {
    createNotificationChannel()
}
private var secondsLeft = INITIAL_SECONDS_LEFT
```

6. Create a function to provide you with `Notification.Builder`:

```
private fun getNotificationBuilder():
    NotificationCompat.Builder
{
    val pendingIntent = getPendingIntent()
    return NotificationCompat.Builder(
        applicationContext, channelId
    ).setContentTitle("Agent approaching destination")
        .setSmallIcon(
            R.drawable.ic_launcher_foreground
        ).setContentIntent(pendingIntent)
        .setTicker(
            "Agent dispatched, tracking movement"
        ).setOngoing(true)
        .setForegroundServiceBehavior(
            FOREGROUND_SERVICE_IMMEDIATE
        ).setOnlyAlertOnce(true)
}
```

This function constructs a `NotificationCompat.Builder` class. The builder lets you define a title, a small icon (the size of the icon differs based on the device) to use, the `Intent` object to be triggered when the user clicks on the notification, and a ticker (this is used for accessibility; before Android Lollipop, this showed before the notification was presented).

Setting the notification to ongoing prevents users from dismissing it. This also prevents Android from muting the notification due to frequent updates. Setting the foreground service behavior to `FOREGROUND_SERVICE_IMMEDIATE` ensures that the notification shows up straight away. Without it, the notification may take a few seconds to appear. Finally, setting the alert to only show once prevents the system from disrupting the user (with sound or vibrations, for example) with every update to the notification past the initial call.

You can set other properties, too. Explore the `NotificationCompat.Builder` class. In a real project, remember to use string resources from `strings.xml` rather than hardcoded strings.

7. Still in `RouteTrackingWorker`, implement the following code to introduce a function to create a `ForegroundInfo` instance:

```
private fun createForegroundInfo(
    secondsLeft: Int
): ForegroundInfo {
    val notification = getNotificationBuilder()
        .setContentText(
            "$secondsLeft seconds to destination"
        ).build()

    return if (
        Build.VERSION.SDK_INT >= VERSION_CODES.Q
    ) {
        ForegroundInfo(
            NOTIFICATION_ID, notification,
            FOREGROUND_SERVICE_TYPE_DATA_SYNC
        )
    } else {
        ForegroundInfo(NOTIFICATION_ID, notification)
    }
}
```

You start by obtaining a `NotificationCompat.Builder` object. You then construct a `ForegroundInfo` instance, providing it with `NOTIFICATION_ID` and a notification built using the builder. If the Android version is Android Q or newer, you also provide the foreground service type. The function returns the newly created `ForegroundInfo` instance.

8. Add a function to update an existing notification with the current progress:

```
@SuppressWarnings("MissingPermission")
private fun updateNotification() {
    val notification = getNotificationBuilder()
        .setContentText(
            "$secondsLeft seconds to destination"
        ).build()
    NotificationManagerCompat
```

```

        .from(applicationContext)
        .notify(NOTIFICATION_ID, notification)
    }

```

9. Define a call that tracks your deployed SCA as it approaches its designated destination:

```

private suspend fun trackToDestination(
    catAgentId: String
) {
    secondsLeft = INITIAL_SECONDS_LEFT
    for (i in INITIAL_SECONDS_LEFT downTo 0) {
        secondsLeft = i
        setProgress(
            workDataOf(
                DATA_KEY_CAT_AGENT_ID to catAgentId,
                DATA_KEY_SECONDS_LEFT to secondsLeft
            )
        )
        updateNotification()
        delay(1000L)
    }
}

```

This will first set the time left to the initial time you declared in *step 3* earlier. Then, it will count down from 10 (the value of `INITIAL_SECONDS_LEFT`) to 0, updating the time left, worker progress, and notification with the remaining time. It will then delay the next step for one second.

10. Override `doWork()` like so:

```

override suspend fun doWork(): Result {
    val catAgentId = inputData
        .getString(DATA_KEY_CAT_AGENT_ID) ?:
        error("Agent ID must be provided")

    setForeground(getForegroundInfo())
    trackToDestination(catAgentId)
    return Result.success()
}

```

You first obtain the SCA ID from the input data. This worker will not work without an agent ID, so you throw an exception if one is not provided.

Next, you also set the worker to a foreground worker, providing it with a `ForegroundInfo` instance. You continue to track the SCA. When tracking is done, you notify observers that the task is complete by returning `success()`.

11. Also, override `getForegroundInfo()`:

```
override suspend fun getForegroundInfo() =  
    createForegroundInfo(secondsLeft)
```

This function provides the system with the information for the notification to be shown as soon as the worker becomes a foreground worker.

12. Update your `AndroidManifest.xml` file to request the necessary permissions and introduce the service:

```
<manifest ...>  
    <uses-permission  
        android:name=  
            "android.permission  
                .FOREGROUND_SERVICE_DATA_SYNC" />  
    <uses-permission  
        android:name=  
            "android.permission.POST_NOTIFICATIONS" />  
    <application ...>  
        <service  
            android:name=  
                "androidx.work.impl  
                    .foreground.SystemForegroundService"  
            android:foregroundServiceType="dataSync"  
            tools:node="merge" />  
    <activity ...>
```

First, we declare that your app requires the `FOREGROUND_SERVICE_DATA_SYNC` permission. Unless we do so, the system will block your app from using foreground services. We also request the `POST_NOTIFICATIONS` permission, without which we cannot present notifications on SDK 33+. Next, we declare the service. Setting `android:name` tells the system to use the `SystemForegroundService` class for this service. Defining the service with `android:foregroundServiceType="dataSync"` tells the system what type of work the service does.

13. Open your MainActivity.kt file. At the top of the MainActivity class, declare a field for holding a WorkManager instance:

```
class MainActivity : ComponentActivity() {  
    private val workManager =  
        WorkManager.getInstance(this)  
    ...  
}
```

14. At the bottom of the MainActivity class, add a function to check whether you have permission to show the user notifications:

```
private fun isNotificationPermitted(): Boolean = if (  
    Build.VERSION.SDK_INT >= VERSION_CODES.TIRAMISU  
) {  
    ContextCompat.checkSelfPermission(  
        this, POST_NOTIFICATIONS  
    ) == PackageManager.PERMISSION_GRANTED  
} else {  
    true  
}
```

Remember, if the Android version is older than Tiramisu, there is no need for special permission to show notifications.

15. Right above isNotificationPermitted, add a function to start tracking the SCA:

```
private fun startTracking(catAgentId: String) {  
    val catTrackingRequest =  
        OneTimeWorkRequestBuilder<RouteTrackingWorker>()  
            .setInputData(  
                workDataOf(  
                    DATA_KEY_CAT_AGENT_ID to  
                    catAgentId  
                )  
            )  
            .setExpedited(  
                RUN_AS_NON_EXPEDITED_WORK_REQUEST  
            )  
            .build()
```

```
        workManager.enqueue(catTrackingRequest)
    }
```

This is similar to how you enqueued work in *Exercise 8.01*.

16. Immediately after `startTracking(String)`, implement a function to collect work information:

```
private suspend fun collectWorkInfo(
    requestId: UUID
) {
    workManager.getWorkInfoByIdFlow(requestId)
        .collect { info ->
            when {
                Info == null -> return@collect
                info.state.isFinished -> {
                    Toast.makeText(
                        this, "Agent arrived!",
                        LENGTH_SHORT
                    ).show()
                }
            }
            info.progress != Data.EMPTY -> {
                val catAgentId = info.progress
                    .getString(DATA_KEY_CAT_AGENT_ID)
                val timeSeconds = info.progress
                    .getInt(DATA_KEY_SECONDS_LEFT, 0)
                Toast.makeText(
                    this,
                    "$catAgentId arrives in $timeSeconds
                    seconds",
                    LENGTH_SHORT
                ).show()
            }
        }
}
```

Much like in the previous exercise, this function monitors the completion of a worker given its request ID. Unlike the previous exercise, the function also monitors progress.

17. At the top of the `setContent` block, declare a `trackingWorkId` field. This will hold the ID of the work request:

```
setContent {
    var trackingWorkId: UUID? by remember {
        mutableStateOf(null)
    }
}
```

18. Right after `trackingWorkId`, declare a `requestPermissionLauncher` field. You will be implementing a simplified version of permission requesting in this exercise. A more thorough implementation was covered in *Chapter 7, Android Permissions and Google Maps*:

```
var trackingWorkId: UUID? by remember { ... }

val requestPermissionLauncher =
    rememberLauncherForActivityResult(
        contract = RequestPermission(),
        onResult = { hasPermission ->
            trackingWorkId = startTracking("Agent007")
        }
    )
...

```

19. Immediately after `requestPermissionLauncher`, add a `LaunchedEffect` block to track the work progress:

```
LaunchedEffect(key1 = trackingWorkId) {
    trackingWorkId?.let { collectWorkInfo(it) }
}
```

20. Replace the contents of the `Scaffold` composable with the following button:

```
Button(
    onClick = {
        if (isNotificationPermitted()) {
            trackingWorkId = startTracking("Agent007")
        } else {
            requestPermissionLauncher
                .launch(POST_NOTIFICATIONS)
        }
    },
)
```

```
        modifier = Modifier.padding(innerPadding)
    ) {
        Text(text = "Start")
    }
}
```

When clicked, this button will check whether permission is required to show a notification. If it is, it will request the user to allow it. If not, it will start tracking the agent.

21. Launch the app. After tapping the button, a notification should pop up, as shown in *Figure 8.3*.

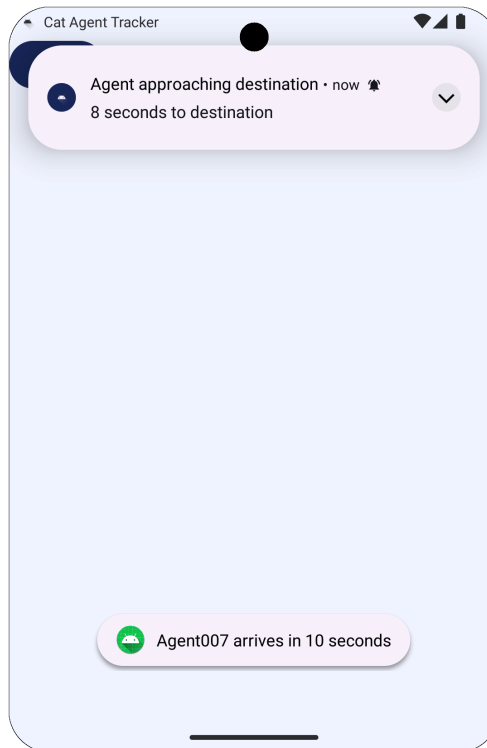


Figure 8.3 – The notification counting down

After tapping the **Start** button, you should see a notification in your status bar and at the top of the screen. That notification should then count down from 10 to 0 and disappear. Toasts should also inform you of the progress and arrival of the agent at its destination. Seeing that last toast tells you that you managed to communicate the SCA ID to the service, as well as get it back throughout the progression of the background task.

Up until now, we have let the work run to completion without interference. In the next section, we will see how we can cancel or update work that is already ongoing.

Canceling or updating ongoing work

Up until now, we have fired off a worker and waited for work to complete. However, what happens if you want to cancel the work? What happens if you want to update the requirements mid-work? For example, what if you want to stop music playback or a file download? What do you do if the backup task hasn't run in days because the constraints were too restrictive, and you want to loosen them? In this section, we will discuss the mechanisms for aborting and updating ongoing work.

To cancel ongoing work, we need the `WorkRequest` ID. This is the same UUID that we used to collect updates for the work. With that ID and a `WorkManager` instance, we can call `WorkManager.cancelWorkById(UUID)`. The work manager will then attempt to cancel the work. Cancellation is not guaranteed – ongoing work may still be executed.

To update work, Google recommends avoiding canceling the ongoing work and recreating it. This is because cancellation is not guaranteed, which means the user might end up with duplicate work taking place. There is also the risk of disrupting the natural flow of the original work, such as in the case of scheduled work. Writing the logic to handle these scenarios is expensive, error-prone, and best avoided. There are exceptions to this recommendation, and we will touch on those shortly.

For most cases, it is best to use the `WorkManager.updateWork(WorkRequest)` function. Like the `enqueue` function, `updateWork(WorkRequest)` takes a `WorkRequest` instance as input. The main difference between the two functions is that `updateWork` expects the `WorkRequest` object to have an ID explicitly set to an earlier enqueued `WorkRequest` ID. This can be done by calling `WorkRequest.Builder.setId(UUID)`. The other difference is that we can only set new constraints, replacing the existing ones. There is no way to send data to a worker that is already enqueued or running.

I mentioned earlier that there are cases where canceling and recreating work is the way to go. In some scenarios, we may need to fundamentally change the nature of the work. For example, we may want to replace a `OneTimeWorkRequest` object with a `PeriodicWorkRequest` one. This cannot be done via the `updateWork(WorkRequest)` function. In these scenarios, we have no choice but to cancel the work and recreate it.

Exercise 8.03 – Aborting SCA deployment by canceling a worker

In *Exercise 8.02 – Tracking your SCA with a foreground worker*, you launched a worker that ran for 10 seconds while an SCA was deployed. However, what if you want to abort the mission while the SCA is still en route? In this exercise, you will implement a mechanism to cancel ongoing work:

1. Open the app you created in *Exercise 8.02*.
2. Open `MainActivity.kt`. This entire exercise will be performed within this file.

- Find the declaration of `trackingWorkId`, and add the following function after it:

```
fun cancelTracking() {
    trackingWorkId?.let(workManager::cancelWorkById)
}
```

- Find the `Button` composable within the `Scaffold` block. Wrap it in a `Column` composable:

```
Column {
    Button(...)
}
```

- Add another button after the existing button, which cancels tracking when tapped and has a **Cancel** label:

```
Button(...)
Button(
    onClick = {
        cancelTracking()
    },
    modifier = Modifier.padding(innerPadding)
) {
    Text(text = "Cancel")
}
```

- Update the `collectWorkInfo(UUID)` function, replacing the `info.state.isFinished` check with checks for two specific states – `SUCCEEDED` and `CANCELLED`:

```
info.state == WorkInfo.State.SUCCEEDED -> {
    Toast.makeText(
        this, "Agent arrived!", LENGTH_SHORT
    ).show()
info.state == WorkInfo.State.CANCELLED -> {
    Toast.makeText(
        this, "Deployment cancelled!", LENGTH_SHORT
    ).show()
```

- Run the app. Tap the **Start** button to start tracking, and then tap **Cancel** before the work completes. You should see a toast informing you that the work was canceled, and the notification will disappear.

In this exercise, you practiced canceling ongoing work. This knowledge is helpful when you want to stop playback, abort a long download, or cancel an upcoming alarm, to name a few cases.

With all the knowledge gained from this chapter, let's complete the following activity.

Activity 8.01 – A reminder to drink water

The average human loses about 2,500 milliliters of water per day. To stay healthy, we need to consume as much water as we lose. However, due to the busy nature of modern life, a lot of us forget to stay hydrated regularly.



See <https://packt.link/90nbQ> for more information on this.

Suppose you wanted to develop an app that keeps track of your water loss (statistically) and gives you a constant update on your fluid balance. Starting from a balanced state, the app would gradually decrease the user's tracked water level. The user could tell the app when they drank a glass of water, and it would update the water level accordingly.

The continuous updating of the water level will leverage your knowledge of running a background task, and you will also utilize your knowledge of communicating with a service to update a balance in response to user interaction.

The following steps will help you complete the activity:

1. Create an empty activity project and name your app `My Water Tracker`.
2. Add the required work manager dependency to your project:

```
androidx-work-runtime = {  
    group = "androidx.work",  
    name = "work-runtime",  
    version.ref = "androidxWorkRuntime"  
}
```

3. Add the data sync foreground service and post notification permissions to your `AndroidManifest.xml` file.
4. Add the `SystemForegroundService` service to the manifest.
5. Create a new `CoroutineWorker` class named `WaterConsumptionWorker`.

6. Define a static variable in your worker to track the water level. Use `AtomicInteger` to ensure thread safety. Note that to keep the needed precision of fractions of milliliters, this variable should store the water level in microliters. A microliter is 1,000th of a milliliter. In other words, 1,000 microliters is equal to 1 milliliter.
7. Define constants for a notification ID and for an extra Data work data key.
8. Set up the creation of the notification in the worker.
9. Create a `ForegroundInfo` instance and return it from `getForegroundInfo()`.
10. Make the `Worker` object a foreground worker by calling `setForeground(ForegroundInfo)`.
11. Add a button labeled **Start** to start tracking the user's water balance. Add the code to request the notification permissions if they are required.
12. Add the functionality to start the `WaterConsumptionWorker` worker. Make sure to launch it if no permission was needed and if permission was granted.
13. Add a label and a mutable state variable for storing and presenting the water level.
14. Add code to track the progress of work and update the water level on the UI.
15. In the worker, set the water balance to decrease by 0.144 ml every 5 seconds.
16. Update the progress and the notification as the water balance is updated.
17. Add a button to the main activity layout with a **Drank a glass of water** label.
18. When the user taps the button, increase the value of the static `AtomicInteger` variable storing the water balance by 250 ml. You can use a static function to handle the conversion from milliliters to microliters and the assignment.
19. Add a third button, **Cancel**, to the app. Have it cancel any ongoing work when tapped.



The solution to this activity can be found at <https://packt.link/60AKN>.

Summary

In this chapter, we learned how to execute long-running background tasks using `WorkManager` and foreground work. We discussed how to communicate progress to the user and how to get the user back into an app when they tap the notification associated with the work. All the topics covered in this chapter are quite broad, and you could explore building notifications and using the `WorkManager` class further.

For most common scenarios, you now have the tools you need. Common use cases include background downloads, the background cleaning up of cached assets, playing media while the app is not running in the foreground, and, combined with the knowledge we gained from *Chapter 7, Android Permissions and Google Maps*, tracking the user's location over time.

In the next chapter, we will look into making our apps more robust and maintainable by writing unit and integration tests. This is particularly helpful when the code you write runs in the background, and it is not immediately evident when something goes wrong.

Unlock this book's exclusive benefits now

Scan this QR code or go to packtpub.com/unlock, then search this book by name.

Note: Keep your purchase invoice ready before you start.

UNLOCK NOW



Part 3

Code Structure

In this part, we will look at how we can perform asynchronous tasks and then deliver the results to the user interface. We will then build upon that work to use Android Architecture Components to assist in code structuring by separating code that performs tasks (which can be tested) from code that interacts with the user interface (which is harder to test).

We will then look at the available options we have with regard to saving data on the device. Finally, we will explore how we can manage the dependencies inside the application with the help of dependency injection.

This part of the book includes the following chapters:

- *Chapter 9, Testing with JUnit, Mockito, MockK, and Compose*
- *Chapter 10, Coroutines and Flow*
- *Chapter 11, Android Architecture Components*
- *Chapter 12, Persisting Data*
- *Chapter 13, Dependency Injection with Dagger, Hilt, and Koin*

9

Testing with JUnit, Mockito, MockK, and Compose

In this chapter, we will explore how we can test an Android application and the variety of tools that are at our disposal for testing. We will begin by going over the types of tests that are available to us. Each Android project has the ability to run two types of tests. The first is local tests, which are placed in the `test` folder and run on your development machine. The second type is represented by instrumented tests, which are placed in the `androidTest` folder and run on a device or emulator.

We will go over the types of tests that can be written for an application and which folder they can be placed in. We'll be starting from unit tests that typically belong in the `test` folder before moving on to integration tests, which can be placed in either folder, and then to **user interface (UI)** tests, which typically go in the `androidTest` folder. We will then move on to the **test-driven development (TDD)** paradigm, which will help prevent defects during development.

By the end of this chapter, you will be able to write tests using `MockK`, `Robolectric`, and `Compose`, and you will have a good understanding of the TDD process.

We will cover the following topics in this chapter:

- Understanding the types of testing
- Writing a simple JUnit test
- Using Android Studio to run tests
- Mocking objects

- Writing integration tests
- Running UI tests
- Applying TDD

Technical requirements

The complete code for all the exercises and the activity in this chapter is available on GitHub at <https://packt.link/i4Thw>.

Understanding the types of testing

In this section, we will go over some of the ways of organizing and classifying tests inside an Android project and what types of libraries we can use for each category.

We will start at the bottom of the pyramid (see *Figure 9.1*), which is represented by unit tests, then move upward through integration tests, and finally, reach the top, which is represented by end-to-end tests (UI tests). Throughout this chapter, you will work with the tools that aid in wiring each of these types of tests; however, here's a brief overview of what each tool does:

- Mockito and MockK help mainly in unit tests and are useful for creating mocks or test doubles in which we can manipulate inputs so that we can assert different scenarios. A **mock** or **test double** is an object that mimics the implementation of another object. Every time a test interacts with mocks, you can specify the behavior of these interactions.
- Robolectric is an open source library that brings the Android framework onto your machine. It also allows you to test Android-specific components locally and not on the emulator. This can be used for both unit tests and integration tests.
- Jetpack Compose has two main features for testing. The first one allows developers to create interactions (clicking buttons, inserting text, and so on). The second one offers assertions (verifying that views display certain text, are currently being displayed to the user, are enabled, and so on).

Ideally, each application should have three types of tests:

- **Unit tests:** These are local tests that validate individual classes and methods. They should represent most of your tests, and they should be fast, easy to debug, and easy to maintain. They are also known as **small tests**.

- **Integration tests:** These are either local tests with Robolectric or instrumented tests that validate interactions between your app's modules and components. These are slower and more complex than unit tests. The increase in complexity is due to the interaction between the components. These are also known as **medium tests**.
- **UI tests (end-to-end tests):** These are instrumented tests that verify complete user journeys and scenarios. This makes them more complex and harder to maintain; they should represent the smallest number of your total test numbers. These are also known as **large tests**.

In the following figure, you can observe the **testing pyramid**. The recommendation from Google is to keep a ratio of 70:20:10 (unit tests: integration tests: UI tests) for your tests.

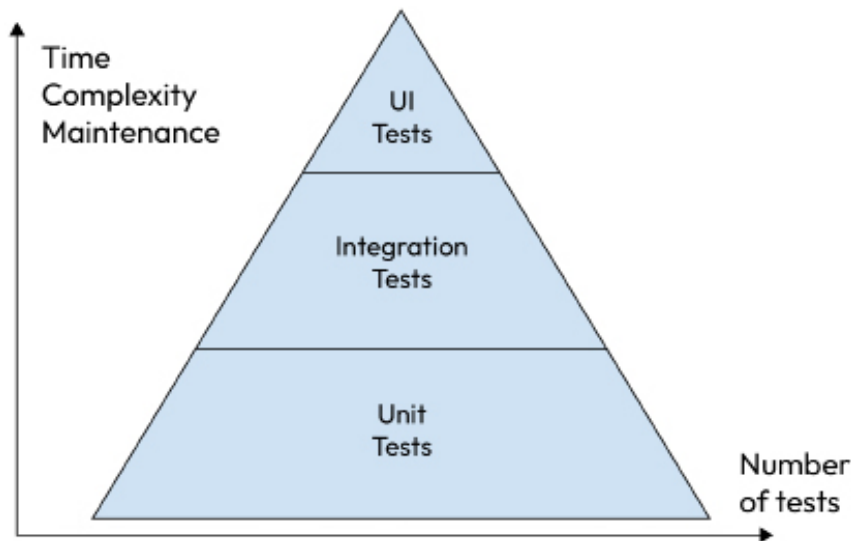


Figure 9.1 – Testing pyramid

As mentioned, a unit test is a test that verifies a small portion of your code, and most of your tests should be unit tests that cover all sorts of scenarios (success, errors, limits, and more). Ideally, these tests should be local, but there are a few exceptions where you can make them instrumented. Those cases are rare and should be limited to when you want to interact with specific hardware of the device.

We will next move on to JUnit, which represents the base framework for writing most types of tests in an Android application.

Writing a simple JUnit test

JUnit is a framework for writing unit tests in both Java and Android. It is responsible for how tests are executed, allowing developers to configure their tests. It offers a multitude of features, such as the following:

- **Setup and teardown:** These are called before and after each test method is executed, allowing developers to set up relevant data for the test and clear it once the test is executed. They are represented by the `@Before` and `@After` annotations, respectively.
- **Assertions:** These are used to verify the result of an operation against an expected value.
- **Rules:** These allow developers to set up inputs that are common for multiple tests.
- **Runners:** Using these, you can specify how the tests can be executed.
- **Parameters:** These allow a test method to be executed with multiple inputs.
- **Orderings:** These specify in which order the tests should be executed.
- **Matchers:** These allow you to define patterns that can then be used to validate the results of the subject of your tests or help you control the behavior of mocks.

In Android Studio, when a new project is created, the app module comes with the JUnit library in Gradle. This should be visible in `gradle/libs.version.toml` and `app/build.gradle.kts`:

```
[versions]
...
junit = "4.13.2"
...
[libraries]
...
junit = { group = "junit", name = "junit", version.ref = "junit" }
...
testImplementation(libs.junit)
```

Creating and running basic unit tests

Let's look at the following class that we need to test:

```
class MyClass {
    fun factorial(n: Int): Int {
        return IntArray(n) {
            it+1
        }.reduce { acc, i ->
```

```
        acc * i
    }
}
}
```

This method should return the factorial of the number *n*. We can start with a simple test that checks the value. To create a new unit test, you will need to create a new class in the test directory of your project.

The typical convention most developers follow is to add the `Test` suffix to the class name and place it under the same package in the test directory. For example, `com.mypackage.ClassA` will have the test in `com.mypackage.ClassATest`:

```
import org.junit.Assert.assertEquals
import org.junit.Test
class MyClassTest {
    private val myClass = MyClass()
    @Test
    fun computesFactorial() {
        val n = 3
        val result = myClass.factorial(n)
        assertEquals(6, result)
    }
}
```

In this test, you can see that we initialize the class under test, and the test method itself is annotated with the `@Test` annotation. The test method itself will assert $(3!) = 6$. The assertion is done using the `assertEquals` method from the JUnit library. A common practice in development is to split the test into three areas, also known as **arrange-act-assert (AAA)**:

- **Arrange:** Where the input is initialized
- **Act:** Where the method under test is called
- **Assert:** Where the verification is done

We can write another test to make sure that the value is correct, but we will end up duplicating the code. We can now attempt to write a parameterized test. To do this, we will need to use the parameterized test runner. The preceding test has its own built-in runner provided by JUnit.

The parameterized runner will run the test repeatedly for different values that we provide, and it will look like the following – please note that the import statements have been removed for brevity:

```
@RunWith(Parameterized::class)
class MyClassTest(
    private val input: Int,
    private val expected: Int
) {
    companion object {
        @Parameterized.Parameters
        @JvmStatic
        fun getData(): Collection<Array<Int>> = listOf(
            arrayOf(0, 1),
            arrayOf(1, 1),
            arrayOf(2, 2),
            arrayOf(3, 6),
            arrayOf(4, 24),
            arrayOf(5, 120)
        )
    }
    private val myClass = MyClass()
    @Test
    fun computesFactorial() {
        val result = myClass.factorial(input)
        assertEquals(expected, result)
    }
}
```

The preceding code will run six tests. The usage of the `@Parameterized` annotation tells JUnit that this is a test with multiple parameters and allows us to add a constructor for the test that will represent the input value for our factorial function and the output. We then defined a collection of parameters with the use of the `@Parameterized.Parameters` annotation.

Each parameter for this test is a separate list containing the input and expected output. When JUnit runs this test, it will run a new instance for each parameter and then execute the test method. This will produce five successes and one failure when we test 0!, meaning that we have found a bug.

Handling edge cases and improving test reliability

We never accounted for a situation when $n = 0$ is true. Now, we can go back to our code to fix the failure. We can do this by replacing the reduce function, which doesn't allow us to specify an initial value, with a fold function, which allows us to give the initial value of 1:

```
fun factorial(n: Int): Int {
    return IntArray(n) {
        it + 1
    }.fold(1, { acc, i -> acc * i })
}
```

Running the tests now, they will all pass. However, that doesn't mean we are done here. There are many things that can go wrong. What happens if n is a negative number? Since we are dealing with factorials, we may get large numbers. We are working with integers in our examples, which means that the integer will overflow after $12!$.

Normally, we would create new test methods in the `MyClassTest` class, but since the parameterized runner is used, all our new methods will be run multiple times. This will cost us time, so we will create a new test class to check our errors:

```
class MyClassTest2 {
    private val myClass = MyClass()
    @Test(expected =
        MyClass.FactorialNotFoundException::class)
    fun computeNegatives() {
        myClass.factorial(-10)
    }
}
```

This would lead to the following change in the class that was tested:

```
class MyClass {
    @Throws(FactorialNotFoundException::class)
    fun factorial(n: Int): Int {
        if (n < 0) {
            throw FactorialNotFoundException
        }

        return IntArray(n) {
            it + 1
        }.fold(1, { acc, i -> acc * i })
    }
}
```



```

    }
    object FactorialNotFoundException : Throwable()
}

```

Let's solve the issue with very large factorials. We can use the `BigInteger` class, which can hold large numbers. We can update the test as follows (the import statements are not shown):

```

@RunWith(Parameterized::class)
class MyClassTest(
    private val input: Int,
    private val expected: BigInteger
) {
    companion object {
        @Parameterized.Parameters
        @JvmStatic
        fun getData(): Collection<Array<Any>> = listOf(
            arrayOf(0, BigInteger.ONE),
            arrayOf(1, BigInteger.ONE),
            arrayOf(2, BigInteger.valueOf(2)),
            arrayOf(3, BigInteger.valueOf(6)),
            arrayOf(4, BigInteger.valueOf(24)),
            arrayOf(5, BigInteger.valueOf(120)),
            arrayOf(13, BigInteger("6227020800")),
            arrayOf(25, BigInteger(
                "15511210043330985984000000"
            ))
        )
    }
    private val myClass = MyClass()
    @Test
    fun computesFactorial() {
        val result = myClass.factorial(input)
        assertEquals(expected, result)
    }
}

```

The class under test now looks like this:

```
@Throws(FactorialNotFoundException::class)
fun factorial(n: Int): BigInteger {
    if (n < 0) {
        throw FactorialNotFoundException
    }
    return IntArray(n) {
        it + 1
    }.fold(BigInteger.ONE, {
        acc, i -> acc * i.toBigInteger()
    })
}
```

In the preceding example, we implemented the factorial with the help of `IntArray`. This implementation is based on Kotlin's ability to chain methods together, but it has one drawback: the fact that it uses memory for the array when it doesn't need to.

We only care about the factorial and not storing all the numbers from 1 to *n*. We can change the implementation to a simple for loop and use the tests to guide us during the refactoring process.

We can observe two benefits of having tests in our application:

- They serve as updated documentation of how the features should be implemented
- They guide us when refactoring code by maintaining the same assertion and detecting whether new changes to the code broke it

Let's update the code to get rid of `IntArray`:

```
@Throws(FactorialNotFoundException::class)
fun factorial(n: Int): BigInteger {
    if (n < 0) {
        throw FactorialNotFoundException
    }
    var result = BigInteger.ONE
    for (i in 1..n){
        result = result.times(i.toBigInteger())
    }
    return result
}
```

If we modify the `factorial` function, as in the preceding example, and run the tests, we should see them all pass.

In certain situations, your tests will use a resource that is common to the test or the application (databases, files, and so on). Ideally, this shouldn't happen for unit tests, but there can always be exceptions.

Let's analyze that scenario and see how JUnit can aid us with it. We will add a companion object, which will store the result, to simulate this behavior:

```
companion object {  
    var result: BigInteger = BigInteger.ONE  
}  
@Throws(FactorialNotFoundException::class)  
fun factorial(n: Int): BigInteger {  
    if (n < 0) {  
        throw FactorialNotFoundException  
    }  
    for (i in 1..n) {  
        result = result.times(i.toBigInteger())  
    }  
    return result  
}
```

If we execute the tests for the preceding code, we will start seeing that some will fail. That's because after the first tests execute the `factorial` function, the result will have the value of the executed tests, and when a new test is executed, the result of the `factorial` function will be multiplied by the previous value of the result.

Normally, this would be good because the tests tell us that we are doing something wrong, and we should remedy this. However, for this example, we will address the issue directly in the tests:

```
@Before  
fun setUp() {  
    MyClass.result = BigInteger.ONE  
}  
@After  
fun tearDown() {  
    MyClass.result = BigInteger.ONE  
}
```

```
@Test
fun computesFactorial() {
    val result = myClass.factorial(input)
    assertEquals(expected, result)
}
```

In the tests, we've added two methods with the `@Before` and `@After` annotations. When these methods are introduced, JUnit will change the execution flow as follows: all methods with the `@Before` annotation will be executed, a method with the `@Test` annotation will be executed, and then all methods with the `@After` annotation will be executed. This process will repeat for every `@Test` method in your class.

If you find yourself repeating the same statements in your `@Before` method, you can consider using the `@Rule` annotation to remove the repetition. We can set up a test rule for the preceding example. Test rules should be in the `test` or `androidTest` packages, as their usage is limited only to testing. They tend to be used in multiple tests, so you can place your rules in a rules package (as before, the import statements are not shown):

```
class ResultRule : TestRule {
    override fun apply(
        base: Statement,
        description: Description?
    ): Statement? {
        return object : Statement() {
            @Throws(Throwable::class)
            override fun evaluate() {
                MyClass.result = BigInteger.ONE
                try {
                    base.evaluate()
                } finally {
                    MyClass.result = BigInteger.ONE
                }
            }
        }
    }
}
```

In the preceding example, we can see that the rule will implement `TestRule`, which comes with the `apply()` method. We then create a new `Statement` object that will execute the base statement (the test itself) and reset the value of the result before and after the statement. We can now modify the test as follows:

```
@JvmField
@Rule
val resultRule = ResultRule()
private val myClass = MyClass()
@Test
fun computesFactorial() {
    val result = myClass.factorial(input)
    assertEquals(expected, result)
}
```

To add the rule to the test, we use the `@Rule` annotation. Since the test is written in Kotlin, we are using `@JvmField` to avoid generating getters and setters because `@Rule` requires a public field and not a method.

In this section, we have learned how we can use JUnit to write tests that can verify small units of our code by verifying the results, errors, or behavior for different parameters. We've also learned how each test is run when they are part of a testing class, and the order of operations being invoked. In the next section, we will look at how we can use Android Studio to understand how we can run tests and view the results.

Using Android Studio to run tests

In this section, we will look at how we can run one or multiple tests in Android Studio. Android Studio comes with a good set of shortcuts and visual tools to help with testing. If you want to create a new test for your class or go to existing tests for your class, you can use the `Ctrl + Shift + T` (Windows) or `Command + Shift + T` (macOS) shortcut. You will need to make sure that the contents of your class are currently in focus in the editor for the keyboard shortcut to take effect.

To run tests, there are two options. The first is to right-click your file or the package and select the **Run Tests in '...'** option. The other option comes in handy if you want to run a test suite independently. You can go to the particular test method and select the green icon near the name of the class, which will execute all the tests in the class.

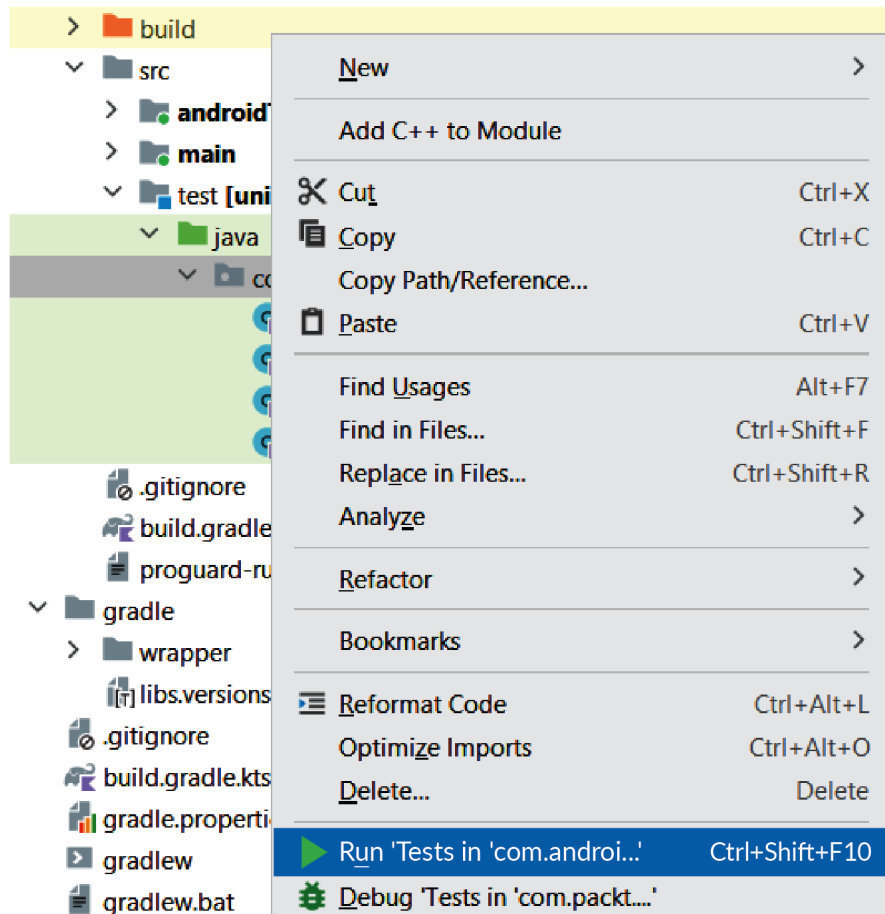


Figure 9.2 – Running a group of tests

For an individual test, you can click the green icon next to the `@Test` annotated methods:

```

2  @RunWith(MockitoJUnitRunner::class)
3  class StringProviderTest {
4
5      @InjectMocks
6      lateinit var stringProvider: StringProvider
7
8      @Mock
9      lateinit var context: Context
10
11     @Test
12     fun provideItemString() {
13         val number = 5
14         val expected = "expected"
15         whenever("Item {5}").thenReturn(expected)
16
17         val result = stringProvider.provideItemString(number)
18
19         assertEquals(expected, result)
20     }

```

Figure 9.3 – Icons for running individual tests

This will trigger the test execution, which will be displayed in the **Run** tab, as shown in the following screenshot. When the tests are completed, they will become either red or green, depending on their success state:

The screenshot shows the 'Run' tab in Android Studio. The top bar indicates the current test is 'NumberAdderParameterTest'. Below the toolbar, the 'Test Results' section is expanded, showing a tree view of the test execution. The root node 'NumberAdderParameterTest' is marked with a green checkmark and a duration of 4 sec 225 ms. It contains five sub-items, all marked with green checkmarks:

- sum[0]: 4 sec 210 ms
- sum[1]: 3 ms
- sum[2]: 3 ms
- sum[3]: 6 ms
- sum[4]: 3 ms

Figure 9.4 – Test output in Android Studio

Another important feature that can be found in tests is the **debug** one. This is important because you can debug both the test and the method under test, so if you find problems in fixing an issue, you can use this to view what the test used as input and how your code handles the input. The third feature you can find in the green icon next to a test is the **Run With Coverage** option.

This helps developers identify which lines of code are covered by the test and which ones are skipped. The higher the coverage, the higher the chances of finding crashes and bugs:

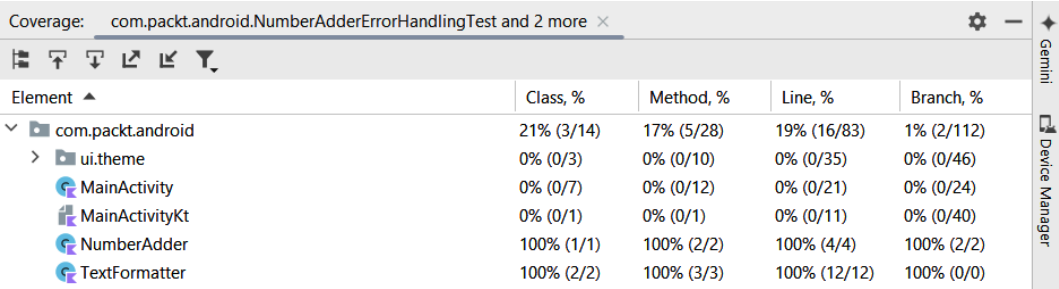


Figure 9.5 – Test coverage in Android Studio

In the preceding figure, you can see the coverage of our class broken down into the number of classes, methods, and lines under test.

Another way to run tests for your Android app is through the command line. This is usually handy in situations where your project has **continuous integration (CI)** set up, meaning that every time you upload your code to a repository in the cloud, a set of scripts will be triggered to test it and ensure functionality.

Since this is done in the cloud, there is no need for Android Studio to be installed. For simplicity, we will be using the **Terminal** tab in Android Studio to emulate that behavior. The **Terminal** tab is usually located in the bottom bar in Android Studio near the **Logcat** tab.

In every Android Studio project, a file called **gradlew** is present. This is an executable file that allows developers to execute Gradle commands. To run your local unit tests, you can use the following:

- `gradlew.bat test` (for Windows)
- `./gradlew test` (for macOS and Linux)

Once that command is executed, the app will be built and tested. You can find a variety of commands that you can input in **Terminal** in the **Gradle** tab, located on the right-hand side of Android Studio.

If you see a message saying **Task list has not been built**, click it and uncheck **Do not build Gradle task list during Gradle Sync**, click **OK**, and then sync the project's Gradle files. The task list should then appear in the list.

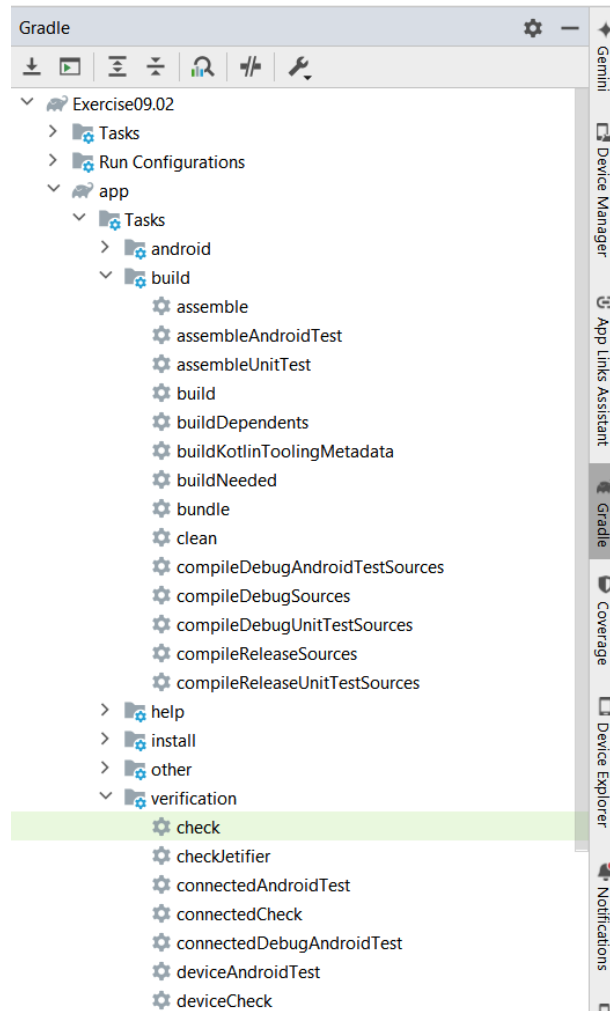


Figure 9.6 – Gradle commands in Android Studio

The output of the tests, when executed from either the **Terminal** or **Gradle** tab, can be found in the `app/build/reports` folder.

In this section, we have learned about the various options for testing that Android Studio provides and how we can visualize testing results. In the next section, we will look at how we can mock objects in tests and how we can use Mockito to do so.

Mocking objects

In the preceding examples, we looked at how to set up a unit test and use assertions to verify the result of an operation. What if we want to verify whether a certain method was called? Or what if we want to manipulate the test input to test a specific scenario? In these types of situations, we can use Mockito.

Mockito is a library that helps developers set up dummy objects that can be injected into the objects under test and allows them to verify method calls, set up inputs, and even monitor the test objects themselves.

The library should be added to your test Gradle setup, as follows:

```
[versions]
mockito = "5.12.0"
[libraries]
mockito = { group = "org.mockito", name = "mockito-core",
            version.ref = "mockito" }
testImplementation(libs.mockito)
```

Now, let's look at the following code example (please note that, for brevity, the import statements have been removed from the following code snippets):

```
class StringConcatenator(private val context: Context) {
    fun concatenate(
        @StringRes stringRes1: Int,
        @StringRes stringRes2: Int
    ): String {
        return context.getString(stringRes1)
            .plus(context.getString(stringRes2))
    }
}
```

In the previous code snippet, we have the Context object, which normally cannot be unit-tested because it's part of the Android framework. We can use the mock function to create a test double and inject it into the StringConcatenator object. Then, we can manipulate the call to getString() to return whatever input we choose. This process is referred to as **mocking**:

```
class StringConcatenatorTest {
    private val context = Mockito.mock(Context::class.java)
    private val stringConcatenator =
```

```

        StringConcatenator(context)

@Test
fun concatenate() {
    val stringRes1 = 1
    val stringRes2 = 2
    val string1 = "string1"
    val string2 = "string2"
    Mockito.`when`(context.getString(stringRes1))
        .thenReturn(string1)
    Mockito.`when`(context.getString(stringRes2))
        .thenReturn(string2)
    val result =
        stringConcatenator.concatenate(
            stringRes1, stringRes2
        )
    assertEquals(string1.plus(string2), result)
}
}

```

In the test, we created a mock context. When the concatenate method was tested, we used Mockito to return a specific string when the `getString()` method was called with a particular input. This allowed us to then assert the result.



``` is an escape character present in Kotlin and should not be confused with a quote mark. It allows the developer to give methods any name they want, including special characters or reserved words.

Mockito is not limited to mocking Android framework classes only. We can create a `SpecificStringConcatenator` class, which will use `StringConcatenator` to concatenate two specific strings from `strings.xml`:

```

class SpecificStringConcatenator(
 private val stringConcatenator: StringConcatenator
) {
 fun concatenateSpecificStrings(): String {
 return stringConcatenator.concatenate(
 R.string.string_1, R.string.string_2
)
 }
}

```

```
 }
}
```

We can write the test for it as follows:

```
class SpecificStringConcatenatorTest {
 private val stringConcatenator = Mockito
 .mock(StringConcatenator::class.java)
 private val specificStringConcatenator =
 SpecificStringConcatenator(stringConcatenator)
 @Test
 fun concatenateSpecificStrings() {
 val expected = "expected"
 Mockito.`when`(stringConcatenator.concatenate(
 R.string.string_1, R.string.string_2))
 .thenReturn(expected)
 val result = specificStringConcatenator
 .concatenateSpecificStrings()
 assertEquals(expected, result)
 }
}
```

In the preceding code snippet, we are mocking the previous `StringConcatenator` class and instructing the mock to return a specific result. If we run the test, it will fail because Mockito is not able to mock final classes. Here, it encounters a conflict with Kotlin that makes all classes *final* unless we specify them as *open*.

Luckily, there is a configuration we can apply that solves this dilemma without making the classes under test *open*:

1. Create a folder named `resources` in the test package.
2. In `resources`, create a folder named `mockito-extensions`.
3. In the `mockito-extensions` folder, create a file named `org.mockito.plugins.MockMaker`.
4. Inside the file, add the following line:

```
mock-maker-inline
```

In situations where you have callbacks or asynchronous work and cannot use the JUnit assertions, you can use Mockito to verify the invocation on the callback or lambdas:

```
class SpecificStringConcatenator(private val
stringConcatenator: StringConcatenator) {
 fun concatenateSpecificStrings(): String {
 return stringConcatenator.concatenate(
 R.string.string_1, R.string.string_2
)
 }
 fun concatenateWithCallback(callback: Callback) {
 callback.onStringReady(
 concatenateSpecificStrings()
)
 }
 interface Callback {
 fun onStringReady(input: String)
 }
}
```

In the preceding example, we have added the `concatenateWithCallback` method, which will invoke the callback with the result of the `concatenateSpecificStrings` method. The test for this method would look something like this:

```
@Test
fun concatenateWithCallback() {
 val expected = "expected"
 Mockito.`when`(stringConcatenator.concatenate(
 R.string.string_1, R.string.string_2))
 .thenReturn(expected)
 val callback = Mockito.mock(
 SpecificStringConcatenator.Callback::class.java
)
 specificStringConcatenator.concatenateWithCallback(
 callback
)
 Mockito.verify(callback).onStringReady(expected)
}
```

In the previous code snippet, we created a mock `Callback` object, which we can then verify at the end with the expected result. Notice that we had to duplicate the setup of the `concatenateSpecificStrings` method to test the `concatenateWithCallback` method. You should never mock the objects you are testing; however, you can use the `spy` method to change their behavior. We can use `spy` on the `stringConcatenator` object to change the outcome of the `concatenateSpecificStrings` method:

```
@Test
fun concatenateWithCallback() {
 val expected = "expected"
 val spy = Mockito.spy(specificStringConcatenator)
 Mockito.`when`(spy.concatenateSpecificStrings())
 .thenReturn(expected)
 val callback =
 Mockito.mock(
 SpecificStringConcatenator.Callback::
 class.java
)
 specificStringConcatenator.concatenateWithCallback(
 callback
)
 Mockito.verify(callback).onStringReady(expected)
}
```

Mockito also relies on dependency injection to initialize class variables and has a custom-built JUnit test runner. This can simplify the initialization of our variables, as follows:

```
@RunWith(MockitoJUnitRunner::class)
class SpecificStringConcatenatorTest {
 @Mock
 lateinit var stringConcatenator: StringConcatenator
 @InjectMocks
 lateinit var specificStringConcatenator:
 SpecificStringConcatenator
}
```

In the preceding example, `MockitoRunner` will inject the variables with the `@Mock` annotation with mocks. Next, it will create a new non-mocked instance of the field with the `@InjectMocks` annotation. When this instance is created, Mockito will try to inject the mock objects that match the signature of the constructor of that object.

In this section, we have looked at how we can mock objects when we write tests and how we can use Mockito to do so. In the next subsection, we will look at a specialized library for Mockito that is better suited to be used with the Kotlin programming language, MockK.

## Using MockK

In this subsection, we will analyze the MockK (`mockk`) library, which simplifies object mocking for Kotlin. You may have noticed when using Mockito that the `when` method from Mockito has escaped. This is because of a conflict with the Kotlin programming language. Mockito is built mainly for Java, and when Kotlin was created, it introduced the `this` keyword. Conflicts like this are escaped using the ``` character.

This, along with some other minor issues, causes some inconvenience when using Mockito in Kotlin. A few libraries were introduced to wrap Mockito and provide a nicer experience when using it. One of those is `mockk`. You can add this library to your module using the following command:

```
[versions]
mockk = "1.14.2"

[libraries]
mockk = { group = "io.mockk", name = "mockk", version.ref = "mockk" }
testImplementation(libs.mockk)
```

A big visible change this library adds is replacing the `when` method with `every`. Another useful change is replacing the `mock` method to rely on generics, rather than class objects. The rest of the syntax is like the Mockito syntax.

We can now update the previous tests with the new library, starting with `StringConcatenatorTest` (the import statements have been removed for brevity):

```
class StringConcatenatorTest {
 private val context = mockk<Context>()
 private val stringConcatenator =
 StringConcatenator(context)

 @Test
 fun concatenate() {
 val stringRes1 = 1
 val stringRes2 = 2
 val string1 = "string1"
 val string2 = "string2"
 every {
```

```

 context.getString(stringRes1)
 } returns string1
 every {
 context.getString(stringRes2)
 } returns string2
 val result = stringConcatenator.concatenate(
 stringRes1, stringRes2
)
 assertEquals(string1.plus(string2), result)
}
}

```

As you can observe, the ``` character has disappeared, and our mock initialization for the Context object has been simplified. We can apply the same thing for the `SpecificStringConcatenatorTest` class (the import statements have been removed for brevity):

```

@RunWith(MockitoJUnitRunner::class)
class SpecificStringConcatenatorTest {

 private val stringConcatenator =
 mockk<StringConcatenator>()
 private val specificStringConcatenator =
 SpecificStringConcatenator(stringConcatenator)

 @Test
 fun concatenateSpecificStrings() {
 val expected = "expected"
 every {
 stringConcatenator.concatenate(
 R.string.string_1, R.string.string_2
) returns expected
 }
 val result =
 specificStringConcatenator
 .concatenateSpecificStrings()
 assertEquals(expected, result)
 }

 @Test
 fun concatenateWithCallback() {

```



```

 val expected = "expected"
 val spy = spyk(specificStringConcatenator)
 every {
 spy.concatenateSpecificStrings()
 } returns expected
 val callback =
 mockk<SpecificStringConcatenator.Callback>(
 relaxed = true
)
 spy.concatenateWithCallback(callback)
 verify { callback.onStringReady(expected) }
 }
}

```

Here, the novelty was the introduction of the relaxed mock. This will make the mock return a simple value for all its methods if no mock value was set up. In our case, we don't care about what the mock should return. We only care that it is being invoked.

In this subsection, we have looked at how we can use the mockk library and how it can simplify the Mockito functions in Kotlin. Next, we will do an exercise on how we can write unit tests with JUnit and MockK.

## Exercise 9.01 – Testing the sum of numbers

Using the JUnit and MockK libraries, write a set of tests for the following class that should cover the following scenarios:

- Assert the values for 0, 1, 5, 20, and `Int.MAX_VALUE`
- Assert the outcome for a negative number
- Fix the code and replace the sum of numbers with the  $n*(n+1)/2$  formula



Throughout this exercise, the import statements are not shown. To see the full code files, refer to <https://packt.link/QxbSs>.

The code to test is as follows:

```

class NumberAdder {
 @Throws(InvalidNumberException::class)
 fun sum(n: Int, callback: (BigInteger) -> Unit) {

```

```

 if (n < 0) {
 throw InvalidNumberException
 }
 var result = BigInteger.ZERO
 for (i in 1..n) {
 result = result.plus(i.toBigInteger())
 }
 callback(result)
 }
 object InvalidNumberException : Throwable()
}

```

Perform the following steps to complete this exercise:

1. Let's make sure the necessary libraries are added to the `gradle/libs.versions.toml` and `app/build.gradle.kts` files:

```

[versions]
junit = "4.13.2"
mockk = "1.14.2"

[libraries]
junit = { group = "junit", name = "junit", version.ref = "junit" }
mockk = { group = "io.mockk", name = "mockk", version.ref = "mockk" }
testImplementation(libs.junit)
testImplementation(libs.mockk)

```

2. Create a class named `NumberAdder` and copy the preceding code inside it.
3. Move the cursor inside the newly created class and, with *Command + Shift + T* or *Ctrl + Shift + T*, create a test class called `NumberAdderParameterTest`.
4. Create a parameterized test inside this class that will assert the outcomes for the 0, 1, 5, 20, and `Int.MAX_VALUE` values:

```

@RunWith(Parameterized::class)
class NumberAdderParameterTest(
 private val input: Int,
 private val expected: BigInteger
) {
 companion object {
 @Parameterized.Parameters

```

```

 @JvmStatic
 fun getData(): List<Array<out Any>> = listOf(
 arrayOf(0, BigInteger.ZERO),
 arrayOf(1, BigInteger.ONE),
 arrayOf(5, 15.toBigInteger()),
 arrayOf(20, 210.toBigInteger()),
 arrayOf(Int.MAX_VALUE, BigInteger(
 "2305843008139952128"
))
)
}

private val numberAdder = NumberAdder()

@Test
fun sum() {
 val callback =
 mockk<(BigInteger) -> Unit>(relaxed = true)
 numberAdder.sum(input, callback)
 verify { callback.invoke(expected) }
}
}

```

5. Create a separate test class that handles the exception thrown when there are negative numbers, named `NumberAdderErrorHandlingTest`:

```

@RunWith(MockitoJUnitRunner::class)
class NumberAdderErrorHandlingTest {
 @InjectMocks
 lateinit var numberAdder: NumberAdder

 @Test(expected =
 NumberAdder.InvalidNumberException::class)
 fun sum() {
 val input = -1
 val callback = mockk<(BigInteger) -> Unit>()
 numberAdder.sum(input, callback)
 }
}

```

6. Since  $1 + 2 + \dots + n = n * (n + 1) / 2$ , we can use the formula in the code, and this would make the execution of the method run faster:

```
class NumberAdder {
 @Throws(InvalidNumberException::class)
 fun sum(n: Int, callback: (BigInteger) -> Unit) {
 if (n < 0) {
 throw InvalidNumberException
 }
 callback(
 n.toBigInteger()
 .times((n.toBigInteger() +
 1.toBigInteger()))
 .divide(2.toBigInteger()))
 }
 object InvalidNumberException : Throwable()
}
```

7. Run the tests by right-clicking the package in which the tests are located and selecting **Run Tests in [package\_name]**. An output similar to the following will appear, signifying that the tests have passed:

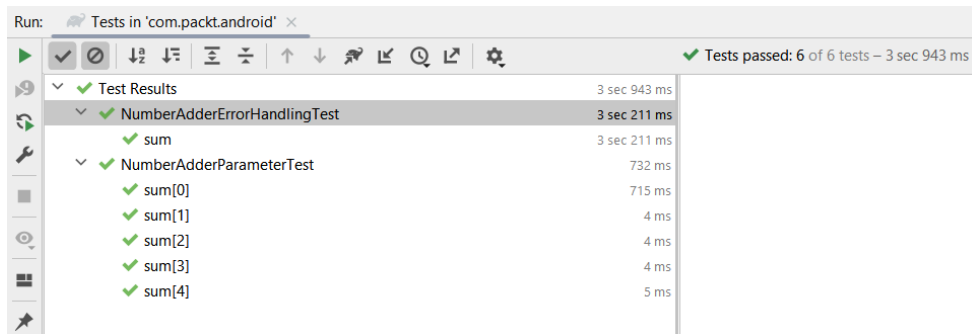


Figure 9.7 – Output of Exercise 9.01

By completing this exercise, we have taken the first steps into unit testing, managed to create multiple test cases for a single operation, taken the first steps into understanding mocking, and used tests to guide us on how to refactor code without introducing any new issues.

We have now looked at how we can write simple tests using JUnit and MockK. In the next section, we will build upon this with the introduction of new testing libraries, such as Robolectric.

## Writing integration tests

In this section, we will look at the next step in the testing pyramid: integration tests. We will explore Robolectric to write more complex tests, and later on, we will learn how we can integrate `ComposeTestRule` with Robolectric to test how we display UI elements.

Let's assume your project is covered by unit tests, where a lot of your logic is held. You now must add these tested classes to an activity or a fragment and require them to update your UI. How can you be certain that these classes will work well with each other? The answer to that question is integration testing.

The idea behind this type of testing is to ensure that different components within your application integrate well with each other. Some examples include the following:

- Your networking components work well
- The storage components can store and retrieve the data correctly
- The UI components load and display the appropriate data
- The transition between different screens in your application is smooth

To aid with integration testing, the requirements are sometimes written in the **given-when-then** format. These usually represent acceptance criteria for a user story and can form the structure of the integrations we want to write. Take the following example:

```
Given I am not logged in
And I open the application
When I enter my credentials
And click Login
Then I see the Main screen
```

On the Android platform, integration testing can be achieved with two libraries:

- **Robolectric:** This library gives developers the ability to test Android components as unit tests – that is, executing integration tests without an actual device or emulator
- **Jetpack ComposeTestRule:** This rule is helpful with testing Compose components when running instrumentation tests on an Android device or emulator

We'll have a look at these libraries in detail in the next sections.

## Robolectric

Robolectric started as an open source library that was meant to give users the ability to unit test classes from the Android framework as part of their local tests instead of the instrumented tests. Recently, it has been endorsed by Google and integrated with AndroidX Jetpack components.

One of the main benefits of this library is the simplicity of testing activities and fragments. This is a benefit when it comes to integration tests because we can use this feature to make sure that our components integrate well with each other.

Some of Robolectric's features are as follows:

- The possibility to instantiate and test the activity and fragment lifecycle
- The possibility to provide configurations for different Android APIs, orientations, screen sizes, layout directions, and so on
- The possibility to change the `Application` class, which then helps to change the modules to permit data mocks to be inserted

To add Robolectric along with the AndroidX integration, we will need the following libraries in `gradle/libs.versions.toml` and `build.gradle.kts`:

```
[versions]
...
junit = "4.13.2"
junitVersion = "1.2.1"
robolectric = "4.14.1"
[libraries]
...
junit = { group = "junit", name = "junit", version.ref = "junit" }
androidx-junit = { group = "androidx.test.ext", name = "junit",
 version.ref = "junitVersion" }
robolectric = { group = "org.robolectric", name = "robolectric",
 version.ref = "robolectric" }
...
testImplementation(libs.junit)
testImplementation(libs.robolectric)
testImplementation(libs.androidx.junit)
```

Let's assume we must deliver a feature in which we display the Result x text, where x is the factorial function for a number that the user will insert in the TextField element. We will assume that we will use a Text element and a Button element. When the button is clicked, we display the factorial result of the number entered in the TextField element in the Text element.

To achieve this, we have two classes, one that computes the factorial and another that concatenates the word Result with the factorial if the number is positive, or returns the Error text if the number is negative.

The factorial class will look something like this (throughout this example, the import statements have been removed for brevity):

```
class FactorialGenerator {
 @Throws(FactorialNotFoundException::class)
 fun factorial(n: Int): BigInteger {
 if (n < 0) {
 throw FactorialNotFoundException
 }
 var result = BigInteger.ONE
 for (i in 1..n) {
 result = result.times(i.toBigInteger())
 }
 return result
 }
 object FactorialNotFoundException : Throwable()
}
```

The TextFormatter class will look like this:

```
class TextFormatter(
 private val factorialGenerator: FactorialGenerator,
 private val context: Context
) {
 fun getFactorialResult(n: Int): String {
 return try {
 context.getString(R.string.result,
 factorialGenerator.factorial(n).toString())
 } catch (e: FactorialGenerator
 .FactorialNotFoundException) {
 context.getString(R.string.error)
 }
 }
}
```

```

 }
}
}

```

Assume that we have the following screen, where we will define the `TextField`, `Button`, and `Text` functions:

```

@Composable
fun MainScreen(
 textFieldText: String,
 onTextFieldValueChange: (String) -> Unit,
 resultText: String,
 onClick: () -> Unit,
 modifier: Modifier = Modifier
) {
 Column(modifier) {
 TextField(
 value = textFieldText,
 onChange = onTextFieldValueChange,
 keyboardOptions = KeyboardOptions(
 keyboardType = KeyboardType.Number
),
 label = { Text(text = "Text Field") }
)
 Button(onClick = onClick) {
 Text(text = "Press Me")
 }
 Text(text = resultText)
 }
}

```

We can combine these components in the `MainActivity` class and have something like this:

```

class MainActivity : ComponentActivity() {
 override fun onCreate(savedInstanceState: Bundle?) {
 super.onCreate(savedInstanceState)
 enableEdgeToEdge()
 val factorialGenerator = FactorialGenerator()
 val textFormatter = TextFormatter(
 factorialGenerator, applicationContext

```



```

)
 setContent {
 Scaffold(
 modifier = Modifier.fillMaxSize()
) { innerPadding ->
 var textFieldText by remember {
 mutableStateOf("")
 }
 var resultText by remember {
 mutableStateOf("")
 }
 MainScreen(
 textFieldText = textFieldText,
 onTextFieldValueChange = {
 textFieldText = it
 },
 resultText = resultText,
 onButtonClick = {
 resultText =
 textFormatter
 .getFactorialResult(
 textFieldText.toInt()
)
 },
 modifier = Modifier.padding(
 innerPadding
)
)
 }
 }
}
}
}

```

We can observe three components interacting with each other in this case. We can use Robolectric to test our activity:

```

@RunWith(AndroidJUnit4::class)
class MainActivityTest {

```

```

@Test
fun `show factorial result in text view`() {
 val scenario = launch(MainActivity::class.java)
 scenario.moveToState(Lifecycle.State.RESUMED)
 scenario.onActivity {
 ...
 }
}

```

In the preceding example, we can see the AndroidX support for the activity test. The `AndroidJUnit4` test runner will set up Robolectric and create the necessary configurations, while the `launch` method will return a scenario object, which we can then play with to achieve the necessary conditions for the test. We can also observe how we can use the ``` character to provide longer names for our functions, in which we can include whitespace characters. The `onActivity` block will be filled with interactions from the Compose testing library.

If we want to add configurations for the test, we can use the `@Config` annotation on both the class and each of the test methods:

```

@Config(
 sdk = [Build.VERSION_CODES.TIRAMISU],
 minSdk = Build.VERSION_CODES.KITKAT,
 maxSdk = Build.VERSION_CODES.TIRAMISU,
 application = Application::class,
 assetDir = "/assetDir/"
)
@RunWith(AndroidJUnit4::class)
class MainActivityTest

```

We can also specify global configurations in the `test/resources` folder in the `robolectric.properties` file, like so:

```

sdk=33
minSdk = 14
maxSdk = 33

```

Another important feature that has recently been added to Robolectric is support for the Jetpack `ComposeTestRule` library.

## Jetpack ComposeTestRule library

Jetpack Compose offers the ability to test `@Composable` functions. If we are using Robolectric, we can write our testing code in the `test` folder, and if not, we can use the `androidTest` folder, and our tests will be viewed as instrumented tests. The testing library is the following:

```
androidx-ui-test-manifest = { group = "androidx.compose.ui",
 name = "ui-test-manifest" }
androidx-ui-test-junit4 = { group = "androidx.compose.ui",
 name = "ui-test-junit4" }
debugImplementation(libs.androidx.ui.test.manifest)
androidTestImplementation (libs.androidx.ui.test.junit4)
```

To test, we would need to use a test rule that provides a set of methods used for interacting with the `@Composable` elements and performing assertions on them. We have multiple ways of obtaining that rule through the following approaches:

```
class MyTest {
 @get:Rule
 var composeTestRuleForActivity =
 createAndroidComposeRule(MyActivity::class.java)
 @get:Rule
 var composeTestRuleForNoActivity = createComposeRule()
 @Test
 fun testNoActivityFunction(){
 composeTestRuleForNoActivity.setContent {
 // Set method you want to test here
 }
 }
}
```

In the preceding snippet, we have two test rules. The first one, `composeTestRuleForActivity`, will start the activity that holds the `@Composable` function that we want to test, and will hold all the nodes we want to assert.

The second one, `composeTestRuleForNoActivity`, provides the ability to set the function we want to test as content. This will then allow the rule to have access to all the `@Composable` elements.

If we want to identify elements from our function, we can use the following methods:

```
composeTestRule.onNodeWithText("My text")
composeTestRule.onNodeWithContentDescription(
 "My content description"
)
composeTestRule.onNodeWithTag("My test tag")
```

In the preceding snippet, we have the `onNodeWithText` method, which will identify a particular UI element using a text label that's visible to the user. The `onNodeWithContentDescription` method will identify an element using the content description set, and `onNodeWithTag` will identify an element using the test tag, which is set using the `Modifier.testTag` method.

Once we identify the element we want to interact with or perform assertions on, we have similar methods for both situations. For interacting with the element, we have methods such as the following:

```
composeTestRule.onNodeWithText("My text")
 .performClick()
 .performScrollTo()
 .performTextInput("My new text")
 .performGesture {
 }
```

In the preceding snippet, we perform a click, scroll, text input, and gesture on the element. For assertions, some examples are as follows:

```
composeTestRule.onNodeWithText("My text")
 .assertIsDisplayed()
 .assertIsNotDisplayed()
 .assertIsEnabled()
 .assertIsNotEnabled()
 .assertIsSelected()
 .assertIsNotSelected()
```

In the preceding example, we assert whether an element is displayed, not displayed, enabled, not enabled, selected, or not selected.

If our UI has multiple elements with the same text, we have the option to extract all of them by using the following:

```
composeTestRule.onAllNodesWithText("My text")
composeTestRule.onAllNodesWithContentDescription(
```

```
 "My content description"
)
 composeTestRule.onAllNodesWithTag("My test tag")`
```

In the previous code snippet, we extract all the nodes that have `My text` as a text, `My content description` as a content description, and `My test tag` as a test tag. The return is a collection, which allows us to assert each element of the collection individually, like so:

```
composeTestRule.onAllNodesWithText("My text")[0]
 .assertIsDisplayed()
```

In the previous code snippet, we assert that the first element that has `My text` is displayed. We can also perform assertions on the collection, like so:

```
composeTestRule.onAllNodesWithText("My text")
 .assertCountEquals(3)
 .assertAll(SemanticsMatcher.expectValue(
 SemanticsProperties.Selected, true))
 .assertAny(SemanticsMatcher.expectValue(
 SemanticsProperties.Selected, true))
```

In the previous code snippet, we assert that the number of elements that have `My text` as a text set is three, assert whether all elements match a `SemanticsMatcher` type, or assert whether any of the elements match a `SemanticsMatcher` type. In this case, it would assert that all the elements are selected and at least one element is selected.

Compose provides `IdlingResource`, which will prevent any interactions or assertions until certain tasks are finished. For example, if you need to perform tasks asynchronously, you can use `IdlingResource` to block the test until your tasks are done:

```
@Before
fun setUp() {
 composeTestRule.registerIdlingResource(
 idlingResource
)
}

@After
fun tearDown() {
 composeTestRule.unregisterIdlingResource(
```

```
 idlingResource
)
}
```

In the preceding snippet, we register `IdlingResource` in the `@Before` annotated method and unregister it in the `@After` method.



A useful cheatsheet is provided by the Android team. It shows all the useful `ComposeTestRule` use cases, and can be found at <https://packt.link/Jldqg>.

`ComposeTestRule` uses the same mechanism as used for accessibility (such as screen readers) to determine how to interact and verify the UI elements. This mechanism is called **semantics**, and what happens is that a **semantic tree** is generated for every layout displayed on the screen. The tree will have simple UI elements such as `Text` and `Button` as leaves, and containers such as `Row` and `Column` as parents. If semantic properties aren't explicitly set, then they will inherit the properties that are visible to the user, such as the text property in the case of the `Text` and `Button` fields. Let's look at an example:

```
@Composable
fun MyScreen() {
 Column {
 Text(text = "Title")
 Row {
 Text(text = "Subtitle")
 Text(text = "Date")
 }
 }
}
```

In the previous code snippet, we have the three leaves in the form of the three `Text` elements and two containers in the form of the `Column` and `Row` elements. The semantic tree for this would look like the following figure:

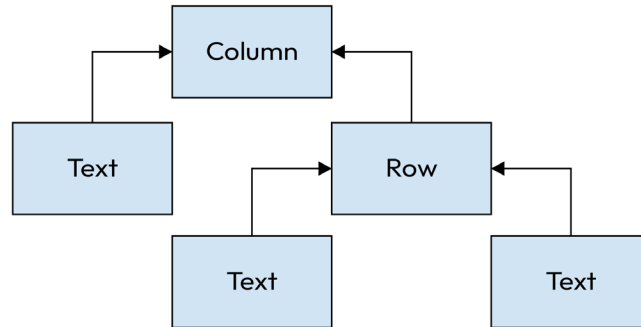


Figure 9.8 – Semantic tree example

In *Figure 9.8*, we can see a semantic tree for the code snippet provided. This is what `ComposeTestRule` will parse to identify the appropriate UI elements. If we want to add descriptions for our UI elements so that we differentiate them for the UI tests to avoid ambiguity, we can use the `Modifier.semantics` function. More information can be found at <https://packt.link/ldPJb>.

We can now combine Robolectric with Compose testing in the following way:

```

@RunWith(AndroidJUnit4::class)
class MainActivityTest {
 @get:Rule
 val composeTestRule = createComposeRule()

 @Test
 fun `show factorial result in text view`() {
 val scenario = launch(MainActivity::class.java)
 scenario.moveToState(Lifecycle.State.RESUMED)
 scenario.onActivity {
 // Write "5" in a TextField that has the hint "Text Field"
 composeTestRule.onNodeWithText("Text Field")
 .performTextInput("5")
 // Click on an element that has the "Press Me" text
 composeTestRule.onNodeWithText("Press Me")
 .performClick()
 // Verify that "Result 120" is displayed on the screen
 composeTestRule.onNodeWithText("Result 120")
 .assertIsDisplayed()
 }
 }
}

```

In the previous code snippet, we combine the `scenario` object created by Robolectric with the `composeTestRule` function, so that Robolectric deals with creating the activity and Compose deals with the screen interactions. In this case, it would simulate the user entering the value of 5 and pressing the button, and then verify that the `Result 120` text is displayed.



For the following exercise, you will need an emulator or a physical device with USB debugging enabled. You can do so by selecting **Tools | AVD Manager** in Android Studio. Then, you can create one with the **Create Virtual Device** option by selecting the type of emulator, clicking **Next**, and then selecting an x86 image. Any image larger than Lollipop should be alright for this exercise. Next, you can give your image a name and click **Finish**.

## Exercise 9.02 – Double integration

In this subsection, we will develop an application that observes the following requirements:

```
Given I open the application And I insert the number n
When I press the Calculate button
Then I should see the text "The sum of numbers from 1 to n is [result]"
Given I open the application And I insert the number -n
When I press the Calculate button
Then I should see the text "Error: Invalid number"
```

You should implement both unit tests and integration tests using Robolectric and Compose, and migrate the integration tests to become instrumentation tests.



Throughout this exercise, the `import` statements are not shown. To see the full code files, refer to <https://packt.link/J712V>.

Implement the following steps to complete this exercise:

1. Let's start by adding the necessary test libraries to `gradle/libs.versions.toml`:

```
[versions]
...
junit = "4.13.2"
junitVersion = "1.2.1"
espressoCore = "3.6.1"
...
```



```

robolectric = "4.14.1"
mockk = "1.14.2"

[libraries]
...
junit = { group = "junit", name = "junit", version.ref = "junit" }
androidx-junit = { group = "androidx.test.ext",
 name = "junit", version.ref = "junitVersion" }
androidx-espresso-core = { group = "androidx.test.espresso",
 name = "espresso-core", version.ref = "espressoCore" }
...
androidx-ui-test-manifest = { group = "androidx.compose.ui",
 name = "ui-test-manifest" }
androidx-ui-test-junit4 = { group = "androidx.compose.ui",
 name = "ui-test-junit4" }
...
robolectric = { group = "org.robolectric", name = "robolectric",
 version.ref = "robolectric"}
mockk = { group = "io.mockk", name = "mockk", version.ref = "mockk" }

[plugins]
...

```

2. Add the following dependencies to app/build.gradle.kts:

```

...
testImplementation(libs.junit)
testImplementation(libs.robolectric)
testImplementation(libs.androidx.junit)
testImplementation(platform(libs.androidx.compose.bom))
testImplementation(libs.androidx.ui.test.junit4)
testImplementation(libs.mockk)
androidTestImplementation(libs.androidx.junit)
androidTestImplementation(libs.androidx.espresso.core)
androidTestImplementation(platform(libs.androidx.compose.bom))
androidTestImplementation(libs.androidx.ui.test.junit4)
debugImplementation(libs.androidx.ui.tooling)
debugImplementation(libs.androidx.ui.test.manifest)

```

3. For Robolectric, we will need to add extra configurations, the first of which is to add the following line to `app/build.gradle.kts` in the `android` block:

```
android {
 ...
 testOptions {
 unitTests {
 isIncludeAndroidResources = true
 }
 }
}
```

4. Create a `resources` directory in the test package. You will need to switch your Android Studio project view from **Android** to **Project**.
5. Add the `robolectric.properties` file and add the following configuration to that file:

```
sdk=35
```

6. Create the `NumberAdder` class. This is like the one in *Exercise 9.01 – Testing the sum of numbers*:

```
class NumberAdder {
 @Throws(InvalidNumberException::class)
 fun sum(n: Int, callback: (BigInteger) -> Unit) {
 if (n < 0) {
 throw InvalidNumberException
 }
 callback(n.toBigInteger().times(
 n.toLong() + 1).toBigInteger())
 .divide(2.toBigInteger()))
 }
 object InvalidNumberException : Throwable()
}
```

7. Create the tests for `NumberAdder` in the test folder. First, create `NumberAdderParameterTest`. This will be the same as in *Exercise 9.01 – Testing the sum of numbers*.
8. Then, create the `NumberAdderErrorHandlingTest` test. This will be the same as in *Exercise 9.01 – Testing the sum of numbers*.

9. In the main folder in the root package, create a class that will format the sum and concatenate it with the necessary strings:

```
class TextFormatter(
 private val numberAdder: NumberAdder,
 private val context: Context
) {
 fun getSumResult(
 n: Int, callback: (String) -> Unit
) {
 try {
 numberAdder.sum(n) {
 callback(context.getString(
 R.string
 .the_sum_of_numbers_from_1_to_is,
 n, it.toString()
))
 }
 } catch (e: NumberAdder.InvalidNumberException)
 {
 callback(context.getString(
 R.string.error_invalid_number
))
 }
 }
}
```

10. Unit test this class for both the success and error scenarios. Start with the success scenario:

```
class TextFormatterTest {
 private val numberAdder = mockk<NumberAdder>()
 private val context = mockk<Context>()
 private val textFormatter = TextFormatter(
 numberAdder, context
)

 @Test
```

```

fun getSumResult_success() {
 val n = 10
 val sumResult = BigInteger.TEN
 val expected = "expected"
 every {
 numberAdder.sum(eq(n), any())
 } answers {
 (it.invocation.args[1] as (BigInteger) ->
 Unit).invoke(sumResult)
 }
 every {
 context.getString(
 R.string
 .the_sum_of_numbers_from_1_to_is,
 n,
 sumResult.toString()
)
 } returns expected
 val callback = mockk<(String) ->
 Unit>(relaxed = true)
 textFormatter.getSumResult(n, callback)
 verify { callback.invoke(expected) }
}

```

11. Then, create the test for the error scenario:

```

@Test
fun getSumResult_error() {
 val n = 10
 val expected = "expected"
 every {
 numberAdder.sum(eq(n), any())
 } throws NumberAdder.InvalidNumberException
 every {
 context.getString(
 R.string.error_invalid_number
)
 } returns expected
}

```

```

 val callback =
 mockk<(String) -> Unit>(relaxed = true)
 textFormatter.getSumResult(n, callback)
 verify { callback.invoke(expected) }
 }

```

12. In `main/res/values/strings.xml`, add the following strings:

```

<string name="the_sum_of_numbers_from_1_to_is">
 The sum of numbers from 1 to %1$d is:
 %2$s
</string>
<string name="error_invalid_number">
 Error: Invalid number
</string>
<string name="calculate">Calculate</string>

```

13. Create the `@Composable` method, which will be responsible for rendering your screen:

```

@Composable
fun MainScreen(
 textFieldText: String,
 onTextFieldValueChange: (String) -> Unit,
 resultText: String,
 onButtonClick: () -> Unit,
 modifier: Modifier = Modifier
) {
 Column(
 modifier = modifier.fillMaxSize(),
 verticalArrangement = Arrangement.Center,
 horizontalAlignment = Alignment.CenterHorizontally
) {
 TextField(
 value = textFieldText,
 onValueChange = onTextFieldValueChange,
 keyboardOptions = KeyboardOptions(
 keyboardType = KeyboardType.Number
),
 label = { Text(text = "Text Field") }
)
 }
}

```

```

 Button(onClick = onButtonClick) {
 Text(
 text = "Press Me"
)
 }
 Text(text = resultText)
 }
}

```

14. In the main folder in the root package, in the MainActivity class, add the NumberAdder and TextFormatter variables in the onCreate function:

```

class MainActivity : ComponentActivity() {
 override fun onCreate(savedInstanceState: Bundle?) {
 super.onCreate(savedInstanceState)
 enableEdgeToEdge()
 val numberAdder = NumberAdder()
 val textFormatter = TextFormatter(
 numberAdder, applicationContext
)
 }
}

```

15. In the setContent block, call the MainScreen function defined in the previous steps:

```

setContent {
 Exercise0902Theme {
 Scaffold(modifier = Modifier.fillMaxSize()) {
 innerPadding ->
 var textFieldText by remember {
 mutableStateOf("")
 }
 var resultText by remember { mutableStateOf("") }
 MainScreen(
 textFieldText = textFieldText,
 onTextFieldValueChange = { textFieldText = it
 },
 resultText = resultText,
 onButtonClick = {textFormatter.
 getSumResult(textFieldText.toInt()) {
 resultText = it
 }
 }
)
 }
 }
}

```

```

 }
 },
 modifier = Modifier.padding(innerPadding)
)
}
}
}

```

16. Create a test for MainActivity and place it in the test directory. It will contain two test methods, one for success and one for errors:

```

@RunWith(AndroidJUnit4::class)
class MainActivityTest {
 @get:Rule
 val composeRule = createComposeRule()

 @Test
 fun `show sum result in text view`() {
 }

 @Test
 fun `show error in text view`() {
 }
}

```

17. In MainActivityTest, implement the success scenario for your test:

```

class MainActivityTest {
 ...

 @Test
 fun `show sum result in text view`() {
 val scenario = launch(MainActivity::class.java)
 scenario.moveToState(Lifecycle.State.RESUMED)
 scenario.onActivity { activity ->
 composeRule.onNodeWithText("Text Field")
 .performTextInput("5")
 composeRule.onNodeWithText("Press Me")
 .performClick()
 }
 }
}

```

```
 composeRule.onNodeWithText(
 activity.getString(
 R.string
 .the_sum_of_numbers_from_1_to_is,
 5, 15)
)
 .assertIsDisplayed()
 }
}
...
}
```

18. In MainActivityTest, implement the error scenario for your test:

```
class MainActivityTest {

 ...

 @Test
 fun `show error in text view`() {
 val scenario = launch(MainActivity::class.java)
 scenario.moveToState(Lifecycle.State.RESUMED)
 scenario.onActivity { activity ->
 composeRule.onNodeWithText("Text Field")
 .performTextInput("-5")
 composeRule.onNodeWithText("Press Me")
 .performClick()
 composeRule.onNodeWithText(
 activity.getString(
 R.string.error_invalid_number
)
)
 .assertIsDisplayed()
 }
 }
}
```



19. At the time of writing, Robolectric doesn't provide support for Android 16, so we will need to configure it to use Android 15. To do this, you will need to create a file called `robolectric.properties` inside the resources folder, which is inside the test folder, and add the following configuration inside:

```
sdk=35
```

You can check <https://packt.link/3ibVu> to see whether a newer version of Robolectric is out that provides support for SDK 36 (Android 16).

If you run the tests by right-clicking the package in which the tests are located and selecting **Run Tests in [package\_name]**, then an output like the following will appear:

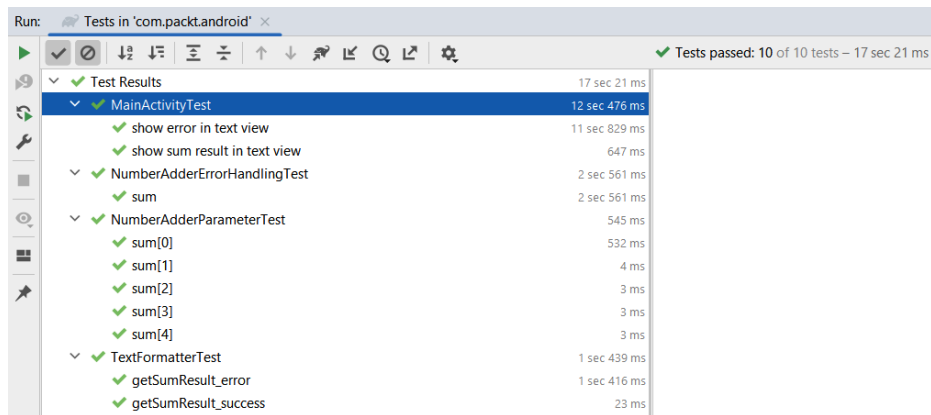


Figure 9.9 – Result of executing the tests in the test folder for Exercise 9.02

The Robolectric test is executed in the same way as a regular unit test; however, there is an increase in the execution time.

20. Let's now migrate the preceding test to an instrumented integration test. To do this, we will copy the preceding test from the test package into the `androidTest` package. Make sure that in the `androidTest` folder, there is a Java folder that contains a package with the same name as the `main/java` folder. You will need to move your tests to this package:

```
@RunWith(AndroidJUnit4::class)
class MainActivityUITest {
 @get:Rule
 val composeRule = createComposeRule()

 @Test
 fun showSumResultInTextView() {
```

```
...
}

@Test
fun showErrorInTextView() {
 ...
}
}
```

If you run the tests by right-clicking the package in which the tests are located and selecting **Run Tests in [package\_name]**, then an output like the following will appear:

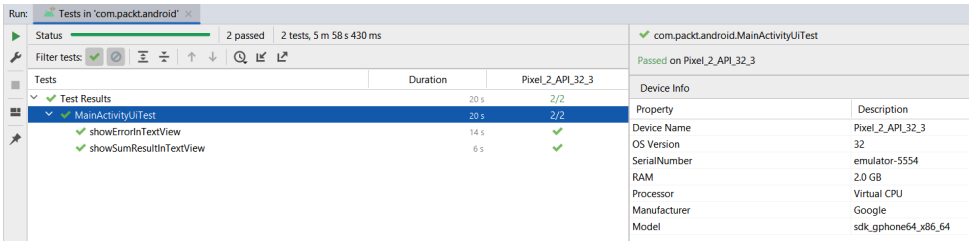


Figure 9.10 – Result of executing the tests in the androidTest folder for Exercise 9.02

In Figure 9.10, we can see what Android Studio displays as an output for the result. If you pay attention to the emulator while the tests are executing, you can see that, for each test, your activity will be opened, the input will be set in the field, and the button will be clicked.

Both of our integration tests (on the workstation and the emulator) try to match the accepted criteria of the requirement. The integration tests verify the same behavior. The only difference is that one checks it locally and the other checks it on an Android device or emulator.

In this section, we have implemented an exercise in which we have written tests with the Robolectric library combined with the Compose testing library and looked at how we can migrate our Robolectric tests from the test folder to the androidTest folder. In the next section, we will look at how we can build upon the existing testing suite with instrumented tests that run on physical devices or emulators.

## Running UI tests

UI tests are instrumented tests where developers can simulate user journeys and verify the interactions between different modules of the application. They are also referred to as end-to-end tests. For small applications, you can have one test suite, but for larger applications, you should split your test suites to cover user journeys (logging in, creating an account, setting up flows, and so on).

Since they are executed on the device, you will need to write them in the `androidTest` package, which means they will run with the Instrumentation framework. Instrumentation works as follows:

- The app is built and installed on the device
- A testing app will also be installed on the device that will monitor your app
- The testing app will execute the tests on your app and record the results

One of the drawbacks of this is the fact that the tests will share persisted data, so if a test stores data on the device, then the second test can have access to that data, which means that there is a risk of failure. Another drawback is that if a test encounters a crash, this will stop the entire testing process because the application under test is stopped.

These issues were solved when the Jetpack libraries were introduced, mainly the orchestrator framework. Orchestrators give you the ability to clear the data after each test is executed, sparing developers the need to make any adjustments. The orchestrator is represented by another application that will manage how the testing app coordinates the tests and the data between the tests.

To add it to your project, you need a configuration like this in the `app/build.gradle.kts` file:

```
android {
 ...
 defaultConfig {
 ...
 testInstrumentationRunner = "androidx.test.runner.AndroidJUnitRunner"
 testInstrumentationRunnerArguments["clearPackageData"] = "true"
 }
 testOptions { execution = "'ANDROIDX_TEST_ORCHESTRATOR'"
 }
}
dependencies {
 ...
 androidTestUtil(libs.androidx.orchestrator)
}
```

Here, `libs.androidx.orchestrator` is defined in `gradle/libs.versions.toml` as follows:

```
[versions]
orchestrator = "1.5.1"
[libraries]
```

```
androidx-orchestrator = { group = "androidx.test", name = "orchestrator",
 version.ref = "orchestrator" }
```

You can execute the orchestrator test on a connected device using Gradle's `connectedCheck` command, either from **Terminal** or from the list of Gradle commands.

In the configuration, you will notice the `testInstrumentationRunner` line. This allows us to create a custom configuration for the test, which gives us the opportunity to inject mock data into the modules:

```
testInstrumentationRunner = "com.android.CustomTestRunner"
```

`CustomTestRunner` looks like this (the import statements are not shown in the following code snippets):

```
class CustomTestRunner: AndroidJUnitRunner() {
 @Throws(Exception::class)
 override fun newApplication(
 cl: ClassLoader?,
 className: String?,
 context: Context?
): Application? {
 return super.newApplication(
 cl,
 MyApplication::class.java.name,
 Context
)
 }
}
```

The test classes themselves can be written by applying the `AndroidJUnit4` syntax with the help of the `androidx.test.ext.junit.runners.AndroidJUnit4` test runner:

```
@RunWith(AndroidJUnit4::class)
class MainActivityUiTest {
}
```

The `@Test` methods themselves run in a dedicated test thread:

```
@Test
fun myTest() {
}
```

Typically, in UI tests, you will find interactions and assertions that may get repetitive. In order to avoid duplicating multiple scenarios in your code, you can apply a pattern called **Robot**. Each screen will have an associated Robot class in which the interactions and assertions can be grouped into specific methods. Your test code will use the Robot classes and assert them. A typical Robot class will look something like this:

```
class MyScreenRobot(val composeRule: ComposeContentTestRule)
{

 fun setText(): MyScreenRobot {
 composeRule.onNodeWithText("Text Field")
 .performTextInput("My text")
 return this
 }

 fun pressButton(): MyScreenRobot {
 composeRule.onNodeWithText("Button")
 .performClick()
 return this
 }

 fun assertText(): MyScreenRobot {
 composeRule.onNodeWithText("Text")
 .assertIsDisplayed()
 return this
 }
}
```

The test will look like this:

```
@Test
fun myTest() {
 MyScreenRobot()
 .setText()
 .pressButton()
 .assertText()
}
```

In this section, we have looked at how we can write instrumented tests and how we can structure them. In the next section, we will look at applying these in an exercise.

## Exercise 9.03 – Dealing with random events

Write an application that will have a button and a text. When the user presses the button, it will generate a number between 1 and 5 and then display the text `Generated number x`. Write a UI test that will cover this scenario. Instead of generating a random number, your test will make sure that 2 is always generated.



Throughout this exercise, the `import` statements are not shown. To see the full code files, refer to <https://packt.link/OL0g3>.

Take the following steps to complete this exercise:

1. Create a new Android Studio project with an **Empty** activity.
2. Make sure that the following are present in your `gradle/libs.versions.toml` file:

```
[versions]
...
junit = "4.13.2"
junitVersion = "1.2.1"
espressoCore = "3.6.1"
...

[libraries]
...
junit = { group = "junit", name = "junit", version.ref = "junit" }
androidx-junit = { group = "androidx.test.ext", name = "junit",
 version.ref = "junitVersion" }
androidx-espresso-core = { group = "androidx.test.espresso",
 name = "espresso-core", version.ref = "espressoCore" }
...
androidx-ui-test-manifest = { group = "androidx.compose.ui",
 name = "ui-test-manifest" }
androidx-ui-test-junit4 = { group = "androidx.compose.ui",
 name = "ui-test-junit4" }
...
```

```
[plugins]
```

```
...
```

3. Add the following libraries to `app/build.gradle.kts`:

```
...
testImplementation(libs.junit)
androidTestImplementation(libs.androidx.junit)
androidTestImplementation(libs.androidx.espresso.core)
androidTestImplementation(platform(libs.androidx.compose.bom))
androidTestImplementation(libs.androidx.ui.test.junit4)
debugImplementation(libs.androidx.ui.tooling)
debugImplementation(libs.androidx.ui.test.manifest)
```

4. In the main folder in the root package, create a class; start with a `Randomizer` class:

```
open class Randomizer(private val random: Random) {
 open fun getTimeToWait(): Int {
 return random.nextInt(5) + 1
 }
}
```

5. Now, create an `Application` class, which will be responsible for creating all the instances of the preceding classes:

```
class MyApplication : Application() {
 val randomizer = Randomizer(Random())
}
```

6. Add the `MyApplication` class to `AndroidManifest.xml` in the application tag with the `android:name` attribute:

```
<application
 android:name=".MyApplication"
...>
```

7. Create the `@Composable` method of `MainScreen`:

```
@Composable
fun MainScreen(
 resultText: String,
```

```
 onClick: () -> Unit,
 modifier: Modifier
) {
 Column(modifier = modifier) {
 Button(onClick = onClick) {
 Text(text = "Press Me")
 }
 Text(text = resultText)
 }
 }
}
```

8. Add your screen to the MainActivity content:

```
class MainActivity : AppCompatActivity() {
 override fun onCreate(savedInstanceState: Bundle?) {
 super.onCreate(savedInstanceState)
 enableEdgeToEdge()
 val randomizer = (application as MyApplication).randomizer
 setContent {
 Exercise0903Theme {
 Scaffold(
 modifier = Modifier.fillMaxSize()
) { innerPadding ->
 var result by remember {
 mutableStateOf("")
 }
 MainScreen(
 resultText = result,
 onClick = {
 result = getString(
 R.string
 .generated_number,
 randomizer.getNumber()
)
 },
 modifier = Modifier.padding(
 innerPadding
)
)
 }
 }
 }
 }
}
```



```

)
 }
}
}
}
}

```

9. Make sure that the relevant strings are added to `strings.xml`:

```

<string name="generated_number">
 Generated number %d
</string>

```

10. Create a `MainActivityUiTest` class in the `androidTest` directory:

```

class MainActivityUiTest {

 @get:Rule
 val composeRule = createComposeRule()

 @Test
 fun verifyGeneratedNumber() {
 val scenario = launch(MainActivity::class.java)
 scenario.moveToState(Lifecycle.State.RESUMED)
 composeRule.onNodeWithText("Press Me") .performClick()
 composeRule.onNodeWithText(
 getApplicationContext<Application>()
 .getString(R.string.generated_number, 2)
)
 .assertIsDisplayed()
 }
}

```

11. Run the test multiple times and check the test results. Notice that the test will have a 20% chance of success. This means that we are dealing with randomness that we need to eliminate from our tests.

12. Tests don't like randomness because it will cause failures and unnecessary maintenance costs, so we need to eliminate it by making the `Randomizer` class open and creating a subclass in the `androidTest` directory. We can do the same for the `MyApplication` class and provide a different randomizer called `TestRandomizer`:

```
class TestRandomizer(random: Random) :
 Randomizer(random)
{
 override fun getNumber(): Int {
 return 2
 }
}
```

13. Now, modify the `MyApplication` class such that we can override the randomizer from a subclass:

```
open class MyApplication : Application() {
 lateinit var randomizer: Randomizer
 override fun onCreate() {
 super.onCreate()
 randomizer = generateRandomizer()
 }
 open fun generateRandomizer():
 Randomizer = Randomizer(Random())
}
```

14. In the `androidTest` directory, create `MyTestApplication`, which will extend `MyApplication` and override the `generateRandomizer` method:

```
class MyTestApplication : MyApplication() {
 override fun generateRandomizer():
 Randomizer = TestRandomizer(Random())
}
```

15. Finally, in the `androidTest/java` folder in the root package, create an instrumentation test runner that will use this new `Application` class inside the test:

```
class MyApplicationTestRunner : AndroidJUnitRunner() {
 @Throws(Exception::class)
 override fun newApplication(
 cl: ClassLoader?,
```

```

 className: String?,
 context: Context?
): Application? {
 return super.newApplication(
 cl,
 MyTestApplication::class.java.name,
 Context
)
 }
}

```

16. Add the new test runner to the Gradle configuration:

```

android {
 ...
 defaultConfig {
 ...
 testInstrumentationRunner =
 "com.packt.android.MyApplicationTestRunner"
 }
}

```

If we run the test now, the test should pass; however, we have a couple of problems with our dependencies. For the Randomizer class, we had to make our class open so that it could be extended in the androidTest folder.

17. In the main/java folder in the root package, create an interface called Randomizer:

```

interface Randomizer {
 fun getNumber(): Int
}

```

18. Rename the previous Randomizer class RandomizerImpl and implement the Randomizer interface as follows:

```

class RandomizerImpl(
 private val random: Random
) : Randomizer {
 override fun getNumber(): Int {
 return random.nextInt(5) + 1
 }
}

```

```
 }
}
```

19. In `MyApplication`, modify the `generateRandomizer` method to have the `Randomizer` return type, which will return an instance of `RandomizerImpl`:

```
open class MyApplication : Application() {
 ...
 open fun generateRandomizer() : Randomizer =
 RandomizerImpl(Random())
}
```

20. Modify `TestRandomizer` to implement the `Randomizer` interface:

```
class TestRandomizer : Randomizer {
 override fun getNumber(): Int {
 return 2
 }
}
```

21. Modify `TestMyApplication` to correct the compilation errors:

```
class TestMyApplication : MyApplication() {
 override fun generateRandomizer(): Randomizer =
 TestRandomizer()
}
```

When running the test now, everything should pass, as shown in *Figure 9.11*:

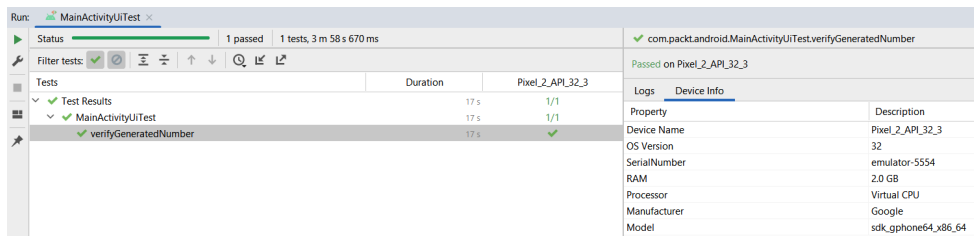


Figure 9.11 – Output of Exercise 9.03

This type of exercise shows how to avoid randomness in a test and provides concrete and repeatable input to make our tests reliable. Similar approaches are taken with dependency injection frameworks, where entire modules can be replaced in the test suite to ensure the test's reliability.

One of the most common things to be replaced is API communication. Another issue this approach solves is the decrease in waiting time. If this type of scenario were to have been repeated across your tests, then their execution time would have increased because of this.

In this exercise, we have looked at how we can write instrumented tests and execute them on an emulator or physical device, as well as how we can switch dependencies that cause flakiness and provide stub data instead.

In the next section, we will look at TDD and how it can be deployed as a process to increase development efficiency and decrease the number of potential bugs.

## Applying TDD

In this section, we will analyze the TDD process, which can be used to improve a team's development process and decrease the number of defects in a project.

Let's assume that you are tasked with building an activity that displays a calculator with the add, subtract, multiply, and divide options. You must also write tests for your implementation. Typically, you would build your UI and your activity, and a separate Calculator class. Then, you would write the unit tests for your Calculator class and then for your activity class.

If you were to translate the TDD process to implementing features on an Android app, you would have to write your UI test with your scenarios first. To achieve this, you can create a skeleton UI to avoid compile-time errors. After your UI test, you would need to write your Calculator test. Here, you would also need to create the necessary methods in the Calculator class to avoid compile-time errors.

If you ran your tests in this phase, they would fail. This would force you to implement your code until the tests pass. Once your Calculator tests pass, you can connect your calculator to your UI until your UI tests pass. While this seems like a counterintuitive approach, it solves two issues once the process is mastered:

- Less time will be spent writing code because you will ensure that your code is testable, and you will need to write only the amount of code necessary for the test to pass
- Fewer bugs will be introduced because developers will be able to analyze different outcomes

Have a look at the following diagram, which shows the TDD cycle:

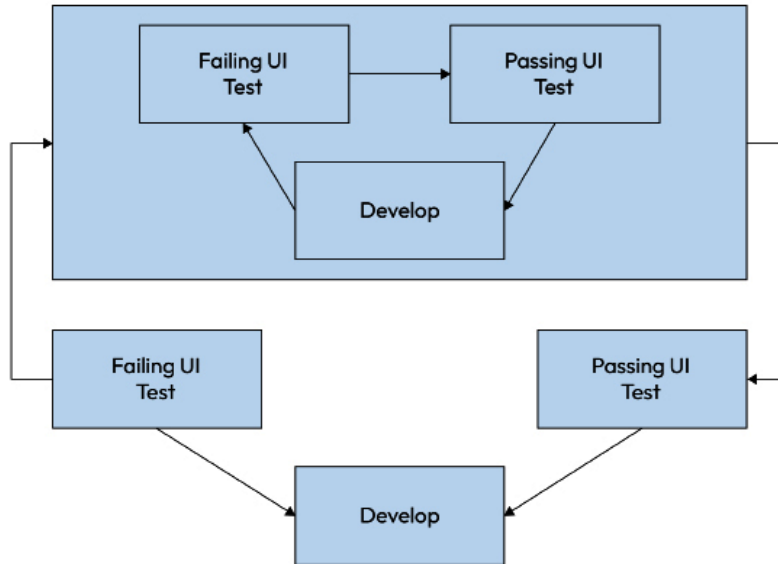


Figure 9.12 – The TDD cycle

In the preceding figure, we can see the development cycle in a TDD process. You should start from a point where your tests are failing, and then implement changes for the tests to pass. When you update or add new features, you can repeat the process.

Going back to our factorial examples, we started with a factorial function that didn't cover all our scenarios and had to keep updating the function every time a new test was added. TDD is built with that idea in mind. You start with an empty function. You first define your testing scenarios: what are the conditions for success? What's the minimum? What's the maximum? Are there any exceptions to the main rule? What are they? These questions can help developers define their test cases. Then, these cases can be written. Let's now see how this can be done practically through the next exercise.

## Exercise 9.04 – Using TDD to calculate the sum of numbers

Write a function that has the integer  $n$  as input and will return the sum of numbers from 1 to  $n$ . The function should be written with a TDD approach, and the following criteria should be satisfied:

- For  $n \leq 0$ , the function will return a -1 value
- The function should be able to return the correct value for `Int.MAX_VALUE`
- The function should be quick, even for `Int.MAX_VALUE`

Perform the following steps to complete this exercise:

1. Create a new Android Studio project with no activity.
2. In the main/java folder in the root package, create an `Adder` class with the `sum` method, which will return `0`, to satisfy the compiler:

```
class Adder {
 fun sum(n: Int): Int = 0
}
```

3. Create an `AdderTest` class in the test directory and define the test cases. We will have the following test cases: `n=1`, `n=2`, `n=0`, `n=-1`, `n=10`, `n=20`, and `n=Int.MAX_VALUE`. We can split the successful scenarios into one method and the unsuccessful ones into a separate method:

```
class AdderTest {
 private val adder = Adder()
 @Test
 fun sumSuccess() {
 assertEquals(1, adder.sum(1))
 assertEquals(3, adder.sum(2))
 assertEquals(55, adder.sum(10))
 assertEquals(210, adder.sum(20))
 assertEquals(
 2305843008139952128L,
 adder.sum(Int.MAX_VALUE)
)
 }
 @Test
 fun sumError(){
 assertEquals(-1, adder.sum(0))
 assertEquals(-1, adder.sum(-1))
 }
}
```

If we run the tests for the `AdderTest` class, we will see an output like the following figure, meaning that all our tests failed:

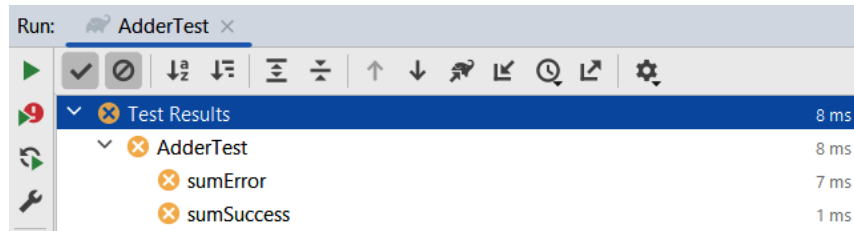


Figure 9.13 – Initial test status for Exercise 9.04

4. Let's first address the success scenarios by implementing the sum in a loop from 1 to n:

```
class Adder {
 fun sum(n: Int): Long {
 var result = 0L
 for (i in 1..n) {
 result += i
 }
 return result
 }
}
```

If we run the tests now, you will see that one will pass and the other will fail, as in the following screenshot:

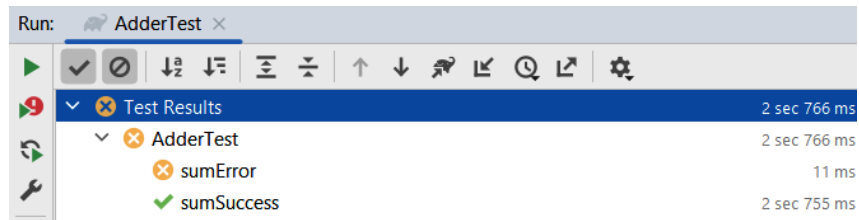


Figure 9.14 – Test status after resolving the success scenario for Exercise 9.04

5. If we look at the time it took to execute the successful test, it seems a bit long. This can add up when thousands of unit tests are present in one project. We can now optimize our code to deal with the issue by applying the  $n(n+1)/2$  formula:

```
class Adder {
 fun sum(n: Int): Long {
 return (n * (n.toLong() + 1)) / 2
 }
}
```



Running the tests now will drastically reduce the execution time to a few milliseconds.

6. Now, let's focus on solving our failure scenarios. We can do this by adding a condition for when  $n$  is smaller than or equal to 0:

```
class Adder {
 fun sum(n: Int): Long {
 return if (n > 0)
 (n * (n.toLong() + 1)) / 2
 else -1
 }
}
```

If we run the tests now, we should see them all passing, as in the following screenshot:

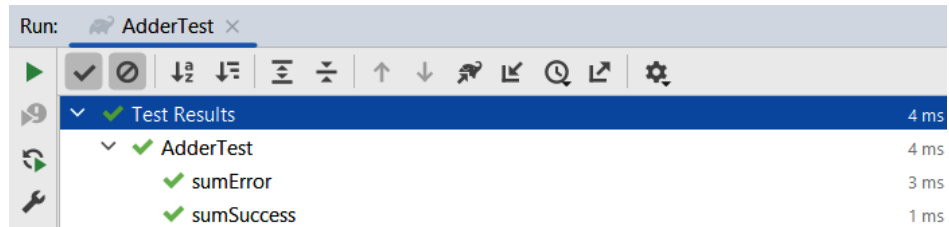


Figure 9.15 – Passing tests for Exercise 9.04

In this exercise, we have applied the concept of TDD to a very small example to demonstrate how the technique can be used. We have observed how, starting from the skeleton code, we can create a suite of tests to verify our conditions, and how, by constantly running tests, we improved the code until a point where all the tests pass. As you have probably noticed, the concept isn't intuitive. Some developers find it hard to define how big the skeleton code should be to start creating the test cases, while others, out of habit, focus on writing the code first and then developing the test. In either case, developers will need a lot of practice with the technique until it's properly mastered.

In the next section, we will develop an application with the TTD approach.

## Activity 9.01 – Developing with TDD

Using the TDD approach, develop an application that will initially show one button. When the button is clicked, a list will be displayed on the screen. Each item in the list will display `Item x`, where  $x$  is the current item. The total number of items is a random number between 1 and 10. When an item is clicked, `Clicked Item y` will display, where  $y$  is the item clicked in the list.

The following tests need to be implemented:

- Unit tests with MockK
- UI tests with Jetpack Compose

To complete this activity, you need to take the following steps:

1. Create a new Android Studio project with an **Empty** activity.
2. In the `androidTest` folder, create a UI test where you will click on the button, verify that 10 items will be created, and then the ninth item will be clicked. Run the test and make sure it fails for the preceding scenario.
3. Create the UI of the application.
4. Create an interface that will contain a method to generate a random number.
5. Create an implementation of this where you will return -1.
6. Create a unit test for your randomizing function using MockK, which will verify that you will always return the number 5.
7. Change the implementation of *step 4* so that you will generate a number between 1 and 10.
8. Run the test from *step 6* to make sure it passes.
9. Connect your randomizing component to the screen from *step 3*.
10. In `androidTest`, create an implementation of the interface from *step 4*, where you will return 9.
11. Create the `Application` class, the `TestApplication` class, and the `TestRunner` class to ensure that your test will provide the test implementation from *step 9*.
12. Run the test from *step 2* and fix any bugs found so that your test passes.



The solution to this activity can be found at <https://packt.link/Px0eQ>.

## Summary

In this chapter, we looked at the ways we can test our Android applications. We also looked at the tools that can help with this and the different types of tests we can deploy.

We started with unit tests and how we can use JUnit, Mockito, and MockK to test small blocks of code in isolation. We then moved on to integration tests with Robolectric and Compose, and then we looked at UI tests for Compose. Finally, we looked at TDD and how it can help us prevent potential bugs by prioritizing the tests.

In the *Activity 9.01 – Developing with TDD* subsection, we combined the concepts and developed a simple Android application using TDD.

In the next chapter, we will tackle how to perform asynchronous work in an Android application through the use of coroutines and flows. We will also look at how they can be used to load data in the background.

### Unlock this book's exclusive benefits now

Scan this QR code or go to [packtpub.com/unlock](https://packtpub.com/unlock), then search this book by name.

*Note: Keep your purchase invoice ready before you start.*

UNLOCK NOW



# 10

## Coroutines and Flow

Some Android applications work locally on the device without external connections, such as a simple calculator app or an app that only saves data on a local database. However, most apps would need a backend server to retrieve or process data. These operations may take a while, depending on the internet connection, device settings, and server specifications. If long-running operations are run in the main **user interface (UI)** thread, the application will be blocked until the tasks are completed. The application might become unresponsive and prompt the user to close and stop using it.

To avoid this, tasks that can take an indefinite amount of time must be run asynchronously. An asynchronous task means it can run in parallel to another task or in the background. For example, while fetching data from a data source asynchronously, your UI can still be displayed and user interaction can occur.

You can use libraries such as Coroutines for asynchronous operations and Flow for reactive operations. We'll discuss both in this chapter.

This chapter introduces you to background operations and data manipulations with Coroutines and Flow. You'll also learn how to manipulate and display the data using Kotlin Flow operators.

By the end of this chapter, you will be able to use Coroutines and Flow to manage network calls in the background. You will also be able to manipulate data with Flow operators.

We will cover the following key topics in this chapter:

- Using Coroutines on Android
- Using Flow on Android

## Technical requirements

The complete code for all the exercises and the activity in this chapter is available on GitHub at <https://packt.link/gb40J>.

Let's get started with Coroutines.

## Using Coroutines on Android

**Coroutines** is a Kotlin library you can use to manage background tasks, such as making network calls and accessing files or databases. Kotlin Coroutines simplifies your asynchronous programming in Android by allowing you to code sequentially the tasks that will be run in a suspending function and then resume afterward. Google has updated its libraries to support Coroutines, and most third-party libraries have added support too.

You can use Coroutines in your Android project by including the latest version of `org.jetbrains.kotlinx: kotlinx-coroutines-core` (the core library for Coroutines) and `org.jetbrains.kotlinx: kotlinx-coroutines-android` (the library that adds support for Android's main thread) in your version catalog file and add it to your app module's `build.gradle` file's dependencies block.

If you want to make a function a suspending function, add the `suspend` keyword to it. Let's say you have the following function in your code:

```
fun getMovies(): List<Movies> { ... }
```

💡 **Quick tip:** Enhance your coding experience with the **AI Code Explainer** and **Quick Copy** features. Open this book in the next-gen Packt Reader. Click the **Copy** button (1) to quickly copy code into your coding environment, or click the **Explain** button (2) to get the AI assistant to explain a block of code to you.

```
function calculate(a, b) {
 return {sum: a + b};
};
```

Copy

1

Explain

2



🔒 **The next-gen Packt Reader** is included for free with the purchase of this book. Scan the QR code OR go to [packtpub.com/unlock](https://packtpub.com/unlock), then use the search bar to find this book by name. Double-check the edition shown to make sure you get the right one.



You can make the `getMovies()` function a suspending function by adding the `suspend` keyword:

```
suspend fun getMovies(): List<Movies> { ... }
```

Then, you can create a coroutine to call the `getMovies` suspending function and display the list of movies returned:

```
CoroutineScope(Dispatchers.IO).launch {
 val movies = movieService.getMovies()
 withContext(Dispatchers.Main) {
 displayMovies(movies)
 }
}
```

The coroutine will fetch the list of movies in the background and suspend the thread, allowing it to do other tasks. When the list of movies is returned, it will resume and display them. An instance of `CoroutineScope` was used for the coroutine's scope. It starts a coroutine using the `launch` builder on the `Dispatchers.IO` dispatcher to specify that it will run in the background. The `withContext` function then switches it to `Dispatchers.Main` to display the list on the main thread. You will explore these components in detail in the next sections.

## Coroutine builders

In the previous section, you started a coroutine with `launch`. It creates coroutine and returns a `Job` object that represents the coroutine. There are other coroutine builders you can use to create coroutines:

- `async`: Returns a value that you can get later with the `await` function
- `runBlocking`: Creates a coroutine outside a coroutine or coroutine scope

The `launch` builder returns a `Job` object that represents the coroutine. It is ideal for running a task and not waiting for its result. The `async` builder is similar to `launch`, but it returns a `Deferred` object for the result of the task done by the coroutine. `runBlocking` starts a new coroutine and blocks the current thread until the task has finished executing its task. This is useful for cases such as unit testing. The following example shows how to use these coroutine builders:

```
scope.launch {
 saveMovies(movies)
}
...
val result = async {
 getTopMovie()
}
displayTopMovie(result.await())
...
```

In the next section, you will explore the coroutine scope.

## Coroutine scope

`CoroutineScope` is the scope wherein the coroutine will run, defining its life cycle. If you cancel a scope, it will cancel all coroutines and their child coroutines. You can create a coroutine scope using `MainScope` (for the main UI thread), a `CoroutineScope` factory function, or other built-in coroutine scopes.

The `CoroutineScope` function requires you to pass in a `CoroutineContext`. `CoroutineContext` is a collection of elements for the coroutines that specify how the coroutine should run. In the previous examples, you have passed a dispatcher for the coroutine as the context:

```
val scope = CoroutineScope(Dispatchers.IO)
scope.launch { ... }
```

Jetpack's Lifecycle library also has `lifecycleScope`, which you can use to create coroutines. It is tied to the activity or fragment (the `Lifecycle` object), so it is canceled when the life cycle is destroyed. To use `lifecycleScope`, you must include the latest version of `androidx.lifecycle:lifecycle-runtime-ktx` in your version catalog file and add it to your app module's `build.gradle` file's dependencies block.

You can use `lifecycleScope` as follows:

```
lifecycleScope.launch { ... }
```

Aside from `launch`, it has additional coroutine builders, depending on the life cycle's state:

- `launchWhenCreated`
- `launchWhenStarted`
- `launchWhenResumed`

The `launchWhenCreated` builder launches the coroutine when the life cycle is created, `launchWhenStarted` launches the coroutine when the life cycle is started, and `launchWhenResumed` launches the coroutine when the life cycle goes back to the `Resumed` state.

`ViewModel` has a default `CoroutineScope` for creating coroutines: `viewModelScope`. The `viewModelScope` scope is canceled when `ViewModel` has been cleared. To use `viewModelScope`, you must include the latest version of `androidx.lifecycle:lifecycle-viewmodel-ktx` in your version catalog file and add it to your app module's `build.gradle` file's dependencies block.

You can use `viewModelScope` as follows:

```
viewModelScope.launch { ... }
```

The `coroutineScope{}` suspending builder allows you to create a `CoroutineScope` scope with the `CoroutineContext` from its outer scope. When a child coroutine fails, it will cancel the parent coroutine and the sibling coroutines. If you do not want this to happen, you can use `supervisorScope{}` instead. The `supervisorScope{}` builder is similar to the `coroutineScope{}` builder, but the coroutine's scope has a `SupervisorJob` job. This allows the children of `supervisorScope` to fail independently of each other.



## Creating coroutines

In your composable functions, you can also use `rememberCoroutineScope` to create a `CoroutineScope` scope in a composable function, which will be canceled automatically when the screen leaves the composition:

```
val scope = rememberCoroutineScope()
scope.launch { ... }
```

To cancel a coroutine, you can call the `cancel` function from the coroutine scope:

```
scope.cancel()
```

In the next section, you will learn about coroutine dispatchers.

## Coroutine dispatchers

Coroutines have a `CoroutineContext`, which includes the coroutine's dispatcher. **Dispatchers** specify what thread the coroutine will use to perform the task. There are four dispatchers you can use:

- `Dispatchers.Main`: Used to run on Android's main thread. A variant called `Dispatchers.Main.immediate` can be used to immediately execute the coroutine in the main thread. The `viewModelScope` and `lifecycleScope` coroutine scopes use `Dispatchers.Main.immediate` by default.
- `Dispatchers.IO`: Used for network operations and reading or writing to files or databases.
- `Dispatchers.Default`: Used for CPU-intensive work such as complex computations and processing images or videos. This is also the dispatcher used when the scope has no dispatcher set.
- `Dispatchers.Unconfined`: This is a special dispatcher not confined to any specific threads. It executes the coroutine in the current thread and resumes it in the thread that is used by the suspending function. This is usually used for unit testing coroutines.

You can specify the dispatcher when setting the context in `CoroutineScope` or when using a coroutine builder. For example, the following shows how to use the `Dispatchers.IO` dispatcher for the coroutine:

```
CoroutineScope(Dispatchers.IO).launch { ... }
...
scope.launch(Dispatchers.IO) { ... }
```

To change the context for your coroutine, you can wrap with a `withContext` function the code that you want to use a different dispatcher. For example, in your suspending function, `getMovies`, which gets movies from your endpoint, you can change the coroutine to use `Dispatchers.IO`:

```
suspend fun getMovies(): List<Movies> {
 withContext(Dispatchers.IO) { ... }
}
```

In the next section, you will explore coroutine contexts.

## Coroutine contexts

A coroutine runs in a **CoroutineContext**, a collection of elements for the coroutine that specifies how the coroutine should run. Coroutine scopes have a default Coroutine context or `EmptyCoroutineContext`. You can pass in `CoroutineContext` values when you create a `CoroutineScope` scope or use a coroutine builder. In the previous examples, you were passing a dispatcher.

The following are some of the `CoroutineContext` elements you can use:

- `CoroutineDispatcher`
- `Job`
- `CoroutineName`
- `CoroutineExceptionHandler`

Dispatchers specify the thread where the coroutine runs, while the `Job` element of the coroutine allows you to manage the coroutine's task. `CoroutineExceptionHandler` can be used to handle exceptions. `CoroutineName` allows you to set a string to represent your coroutine for debugging purposes:

```
viewModelScope.launch(CoroutineName("NetworkCoroutine")) {
 ...
}
```

This gives the coroutine the name of `NetworkCoroutine`.

As the `CoroutineContext` is a collection of elements for the coroutine, you can combine multiple items by using the `+` symbol, as shown here:

```
val context = Dispatchers.main.immediate + SupervisorJob
```

This is similar to what `MainScope`, `viewModelScope`, and `lifecycleScope` use as context for their coroutine scope.

## Coroutine Job elements

Job is a ContextCoroutine element that you can use to manage the coroutine's tasks and its life cycle. Jobs can be canceled or joined together. The launch coroutine builder creates a new job, while the async coroutine builder returns a Deferred<T> object. Deferred is itself a Job object that has a result. To access the job from the coroutine, you can set it to a variable:

```
val job = viewModelScope.launch(Dispatchers.IO) { ... }
```

If a parent job is canceled or failed, its children are also automatically canceled. When a child job is canceled or failed, its parent will also be canceled. SupervisorJob is a special version of a job that allows its children to fail independently of each other.

For example, in the following coroutine, when either the parent job (CoroutineScope(Job())), childJob1, or childJob2 fails, the other coroutines will be canceled as well:

```
CoroutineScope(Job()).launch {
 launch { childJob1() }
 launch { childJob2() }
}
```

If we replace the parent job with SupervisorJob, any failures between the child jobs will not cause the other to be canceled:

```
CoroutineScope(SupervisorJob()).launch {
 launch { childJob1() }
 launch { childJob2() }
}
```

Let's try to use Coroutines in an Android project.

## Exercise 10.01 – using Coroutines in an Android app

For this chapter, you will work with an application that displays popular movies using the The Movie Database API. Go to <https://developers.themoviedb.org> and register for an API key. In this exercise, you will be using Coroutines to fetch a list of popular movies:

1. Open the Popular Movies project in Android Studio in the Chapter10 directory from this book's code repository.

2. Open the version catalog file and add the latest versions of `org.jetbrains.kotlin:kotlinx-coroutines-core` and `org.jetbrains.kotlin:kotlinx-coroutines-android`:

```
[versions]
...
kotlinxCoroutines = "1.9.0"
...
[libraries]
...
kotlinx-coroutines-android = {
 group = "org.jetbrains.kotlin",
 name = "kotlinx-coroutines-android",
 version.ref = "kotlinxCoroutines"
}
kotlinx-coroutines-core = {
 group = "org.jetbrains.kotlin",
 name = "kotlinx-coroutines-core",
 version.ref = "kotlinxCoroutines"
}
```

3. Open your app module, build the gradle file, and add the dependencies for Kotlin Coroutines:

```
implementation(libs.kotlinx.coroutines.android)
implementation(libs.kotlinx.coroutines.core)
```

These will allow you to use Coroutines in your project.

4. Open `MovieApplication` and update `apiKey` with the value from the The Movie Database API:

```
private val apiKey = "your_api_key_here"
```

5. Go to `MovieViewModel` and update the class declaration:

```
class MovieViewModel(
 private val movieRepository: MovieRepository,
 private val dispatcher: CoroutineDispatcher =
```

```
 Dispatchers.IO
) : ViewModel() {
 ...
 }
```

This adds a dispatcher that will be used for the coroutine to get the movies from the repository.

6. Update `getPopularMovies` with the following code:

```
fun getPopularMovies() {
 viewModelScope.launch(dispatcher) {
 try {
 _popularMovies.value =
 movieRepository.getPopularMovies()
 } catch (exception: Exception) {
 _error.value =
 "An error occurred:
 ${exception.message}"
 }
 }
}
```

The `getPopularMovies` function has a coroutine, using `viewModelScope`, that will fetch the movies from `movieRepository`.

7. Run the application. You will see that the app will display a list of popular movie titles:



Figure 10.1 – The app displaying popular movies

8. Click on a movie, and you will see its details, such as its release date and an overview:

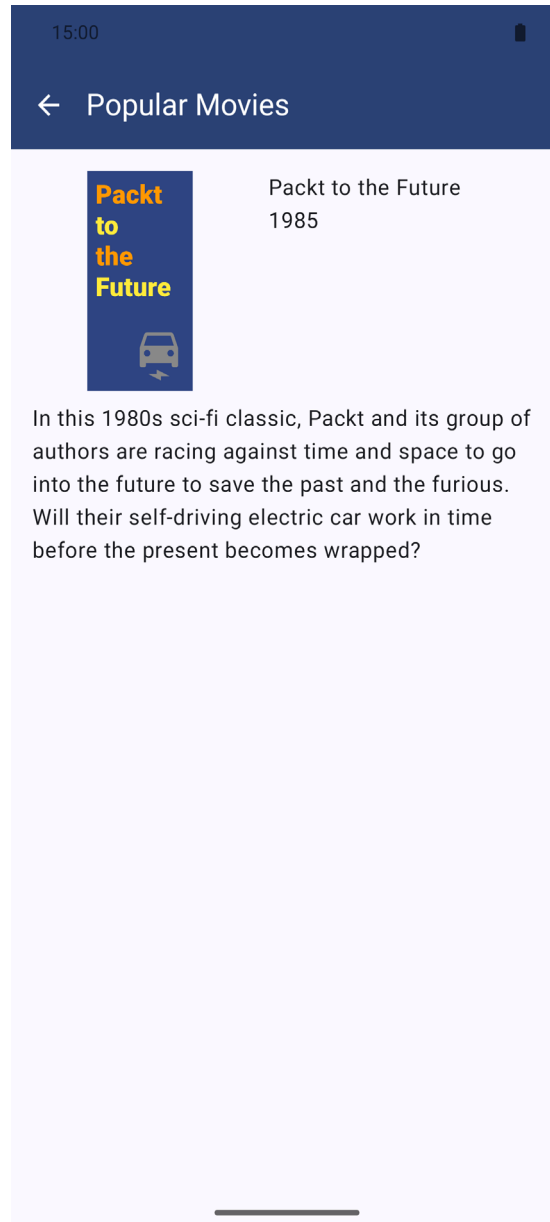


Figure 10.2 – The movie details screen

You have used Coroutines to retrieve and display a list of popular movies from a remote data source without blocking the main thread.

In the next section, you will learn about Kotlin Flow.

## Using Flow on Android

In this section, you will look into using Flow for asynchronous programming in Android. Flow, an asynchronous stream library built on top of Kotlin Coroutines, is ideal for live data updates in your application. Android Jetpack libraries include Room, WorkManager, and Jetpack Compose, and third-party libraries support Flow.

A flow of data is represented by the `kotlinx.coroutines.flow.Flow` interface. Flows emit multiple values of the same type one at a time. For example, `Flow<String>` is a flow that emits string values.

A flow starts to emit values when you call the suspending `collect` function from a coroutine or another suspending function. In the following example, the `collect` function was called from the coroutine created using the `launch` builder of `lifecycleScope`:

```
class MainActivity : AppCompatActivity() {
 ...
 override fun onCreate(savedInstanceState: Bundle?) {
 ...
 lifecycleScope.launch {
 viewModel.fetchMovies().collect { movie ->
 Log.d("movies", "${movie.title}")
 }
 }
 }
}

class MovieViewModel : ViewModel() {
 ...
 fun fetchMovies(): Flow<Movie> { ... }
}
```

Here, the `collect{}` function was called on `viewModel.fetchMovies()`. This will start the flow's emission of movies; each movie title is then logged.



To change the `CoroutineContext` context where the flow runs, you can use the `flowOn()` function to change the dispatcher. The previous example can be updated with a different dispatcher, as shown in the following code:

```
override fun onCreate(savedInstanceState: Bundle?) {
 ...
 lifecycleScope.launch {
 viewModel.fetchMovies()
 .flowOn(Dispatchers.IO)
 .collect { movie ->
 Log.d("movies", "${movie.title}")
 }
 }
}
```

In this example, the dispatcher for the flow will be changed to `Dispatchers.IO`. Calling `flowOn` will only change the functions before it, not the functions and operators after. In the next section, you will learn about collecting flows on Android.

## Collecting flows on Android

In Android, flows are usually collected in the activity or fragment for display in the UI. Moving the app in the background will not stop the data collection. The app must not continue updating the screen to avoid memory leaks and prevent wasting resources.

You can safely collect flows in the UI layer by manually handling life cycle changes or by using `Lifecycle.repeatOnLifecycle` and `Flow.flowWithLifecycle`, available in the `lifecycle-runtime-ktx` library, starting with version 2.4.0.

To use it in your project, add the latest version of `androidx.lifecycle:lifecycle-runtime-ktx` to your version catalog file and add it to your app module's `build.gradle` file's dependencies block. This adds the `lifecycle-runtime-ktx` library to your project, so you can use both `Lifecycle.repeatOnLifecycle` and `Flow.flowWithLifecycle`.

`Lifecycle.repeatOnLifecycle(state, block)` will suspend the parent coroutine until the life cycle is destroyed and execute the suspending block code when the life cycle is at least in the state provided. The flow will stop when the life cycle moves out of the state and restart when the life cycle moves back to the state. `Lifecycle.repeatOnLifecycle` must be called on the activity's `onCreate` function or on the fragment's `onViewCreated` function.

When using `Lifecycle.State.STARTED` for state, the `repeatOnLifecycle` function will start the flow collection when the life cycle is started and stop when the life cycle is stopped (`onStop()` is called).

If you use `Lifecycle.State.RESUMED`, the start will be when the life cycle is resumed, and the stop will be when `onPause` is called or when the life cycle is paused.

The following example shows how you can use `Lifecycle.repeatOnLifecycle`:

```
class MainActivity : AppCompatActivity() {
 ...

 override fun onCreate(savedInstanceState: Bundle?) {
 ...
 lifecycleScope.launch {
 repeatOnLifecycle(Lifecycle.State.STARTED) {
 viewModel.fetchMovies()
 .collect { movie ->
 Log.d("movies", "${movie.title}")
 }
 }
 }
 }
}
```

In this class, `repeatOnLifecycle` with `Lifecycle.State.STARTED` starts collecting the flow of movies when the life cycle is started and stops when the life cycle is stopped.

`Flow.flowWithLifecycle` is another way to safely collect flows in Android. It emits values from the flow and operators preceding the call (the upstream flow) when the life cycle is at least in the state you set or the default, `Lifecycle.State.STARTED`. Internally, it uses `Lifecycle.repeatOnLifecycle`. The following example shows how you can use `Flow.flowWithLifecycle`:

```
class MainActivity : AppCompatActivity() {
 ...

 override fun onCreate(savedInstanceState: Bundle?) {
 ...
 lifecycleScope.launch {
 viewModel.fetchMovies()
 .flowWithLifecycle(Lifecycle.State.STARTED)
 .collect { movie ->
 Log.d("movies", "${movie.title}")
 }
 }
 }
 }
}
```

```
 .flowWithLifecycle(
 lifecycle,
 Lifecycle.State.STARTED
)
 .collect { movie ->
 Log.d("movies", "${movie.title}")
 }
 }
}
```

Here, we used `flowWithLifecycle` with `Lifecycle.State.STARTED` to collect the flow of movies when the life cycle is started and stop when the life cycle is stopped.

In the following section, you will learn how to create flows.

## Creating flows with flow builders

You can create flows using the flow builders from the Kotlin Flow API. The following are the flow builders you can use:

- `flow{}`: This creates a new flow from a suspendable lambda block. You can send values using the `emit` function.
- `flowOf()`: This creates a flow from the specified value or the vararg values.
- `asFlow()`: This is an extension function used to convert a type (sequence, array, range, or collection) into a flow.

The following example shows how to use the flow builders in an application:

```
class MovieViewModel : ViewModel() {
 ...

 fun fetchMovies: Flow<List<Movie>> = flow {
 fetchMovieList().forEach { movie -> emit(movie) }
 }

 fun fetchTop3Titles: Flow<List<String>> {
```

```
 val movies = fetchTopMovies()
 return flowOf(
 movies[0].title,
 movies[1].title,
 movies[2].title
)
}

fun fetchMovieIds: Flow<Int> {
 return fetchMovies().map { it.id }.asFlow()
}
}
```

In this example, `fetchMovies` created a flow using `flow{}` and emitted each movie from the list. The `fetchTop3Titles` function uses `flowOf` to create a flow with the titles of the first three movies. Finally, `fetchMovieIds` converted the list of IDs into a flow of movie IDs using the `asFlow` function.

In the next section, you will learn about the Kotlin flow operators you can use with flows.

## Using operators with flows

There are built-in flow operators you can use with flows. You can collect flows with terminal operators and transform flows with intermediate operators.

Terminal operators, such as the `collect` function used in the previous examples, are used to collect flows. The following are the other terminal operators you can use:

- `count`
- `first` and `firstOrNull`
- `last` and `lastOrNull`
- `fold`
- `reduce`
- `single` and `singleOrNull`
- `toCollection`, `toList`, and `toSet`

These operators work similarly to the Kotlin `Collection` function with the same name.

You can use intermediate operators to modify a flow and return a new one. They can also be chained. The following intermediate operators work the same as the Kotlin Collection functions with the same name:

- `filter`, `filterNot`, `filterNotNull`, and `filterIsInstance`
- `map` and `mapNotNull`
- `onEach`
- `runningReduce` and `runningFold`
- `withIndex`

Additionally, there is a transform operator you can use to apply your own operation. For example, this class has a flow that uses the transform operator:

```
class MovieViewModel : ViewModel() {
 ...
 fun fetchTopRatedMovie(): Flow<Movie> {
 return fetchMoviesFlow()
 .transform {
 if(it.voteAverage > 0.6f) emit(it)
 }
 }
}
```

Here, the transform operator was used in the flow of movies to only emit the ones whose voteAverage property is higher than 0.6 (60%).

There are also size-limiting Kotlin flow operators, such as `drop`, `dropWhile`, `take`, and `takeWhile`, which function similarly to the Kotlin collection functions of the same name.

Let's add Kotlin Flow to an Android project.

## Exercise 10.02 – using Flow in an Android application

In this exercise, you will update the Popular Movies app to use Kotlin Flow in fetching the list of movies:

1. Open the Popular Movies project from *Exercise 10.01 – using Coroutines in an Android app*.

2. Go to the `MovieRepository` class and replace the `getPopularMovies` function with the following:

```
suspend fun getPopularMovies(): Flow<List<Movie>> {
 return flow {
 emit(movieService.getPopularMovies())
 }.flowOn(Dispatchers.IO)
}
```

This changes the `fetchMovies` function to use Kotlin Flow. The flow will emit the list of movies from `movieService.getPopularMovies`, and it will flow on the `Dispatchers.IO` dispatcher.

3. Open the `MovieViewModel` class. Change the content of the `getPopularMovies` function to the following:

```
fun getPopularMovies() {
 viewModelScope.launch(dispatcher) {
 movieRepository.getPopularMovies()
 .catch {
 _error.value =
 "An error occurred:
 ${it.message}"
 }
 .collect {
 _popularMovies.value = it
 }
 }
}
```

This will collect the list of movies from `movieRepository` and set it to `MutableStateFlow` in `_popularMovies` (and `StateFlow` in `popularMovies`).

4. Run the application. The app will display the list of movies, as shown in the following screenshot:



Figure 10.3 – The app displaying popular movies

5. In this exercise, you added Kotlin Flow to an Android project. `MovieRepository` returns the list of movies as a flow, which was collected in `MovieViewModel`. The `MovieViewModel` class uses `StateFlow`, which was then collected in `MainActivity` for displaying.

Let's move on to the activity.

## Activity 10.01 – creating a TV guide app

A lot of people watch television. Most of the time, though, they are not sure what TV shows are currently airing. Suppose you wanted to develop an app that can display a list of these shows from the The Movie Database API's `tv/on_the_air` endpoint using Kotlin Flow.

The app will have two screens: the main screen and the details screen. On the main screen, you will display a list of the TV shows that are on the air. Clicking on a TV show will open the details screen, which displays more information about the selected TV show.

The following are the steps for the completion of the activity:

1. Create a new project in Android Studio and name it `TV Guide`. Set its package name.
2. Add the `INTERNET` permission to the `AndroidManifest.xml` file.
3. Add the dependencies for `Ktor`, `Coroutines`, `kotlinx.serialization`, and other libraries in your `app/build.gradle.kts` file.
4. Create a `TVShow` model class. Create another class, named `TVResponse`, for the response you get from the API endpoint for the TV shows on air.
5. Create a new activity, named `DetailsActivity`, with the UI for the TV show details.
6. Create a `TelevisionService` class for getting the list of TV shows.
7. Create a `TVShowRepository` class with a constructor for `tvService`, and `apiKey tvShows`. Create a function to retrieve the list of TV shows from the endpoint.
8. Create a `TVShowViewModel` class with a constructor for `TVShowRepository`. Add `tvShows` and error state flows and a function that collects the flow from the repository.
9. Create an application class named `TVShowApplication` with a property for `TVShowRepository`.
10. Set `TVShowApplication` as the value for the application in the `AndroidManifest.xml` file.
11. Create a composable class for the UI of the TV show list item.
12. Open `MainActivity` and add the code to display the list of TV shows and a function that will open the details screen when clicking on a TV show from the list.
13. Run your application. The app will display a list of TV shows. Clicking on a TV show will open the details activity, which displays the show details. The main screen and details screen will be similar to the following screenshot:



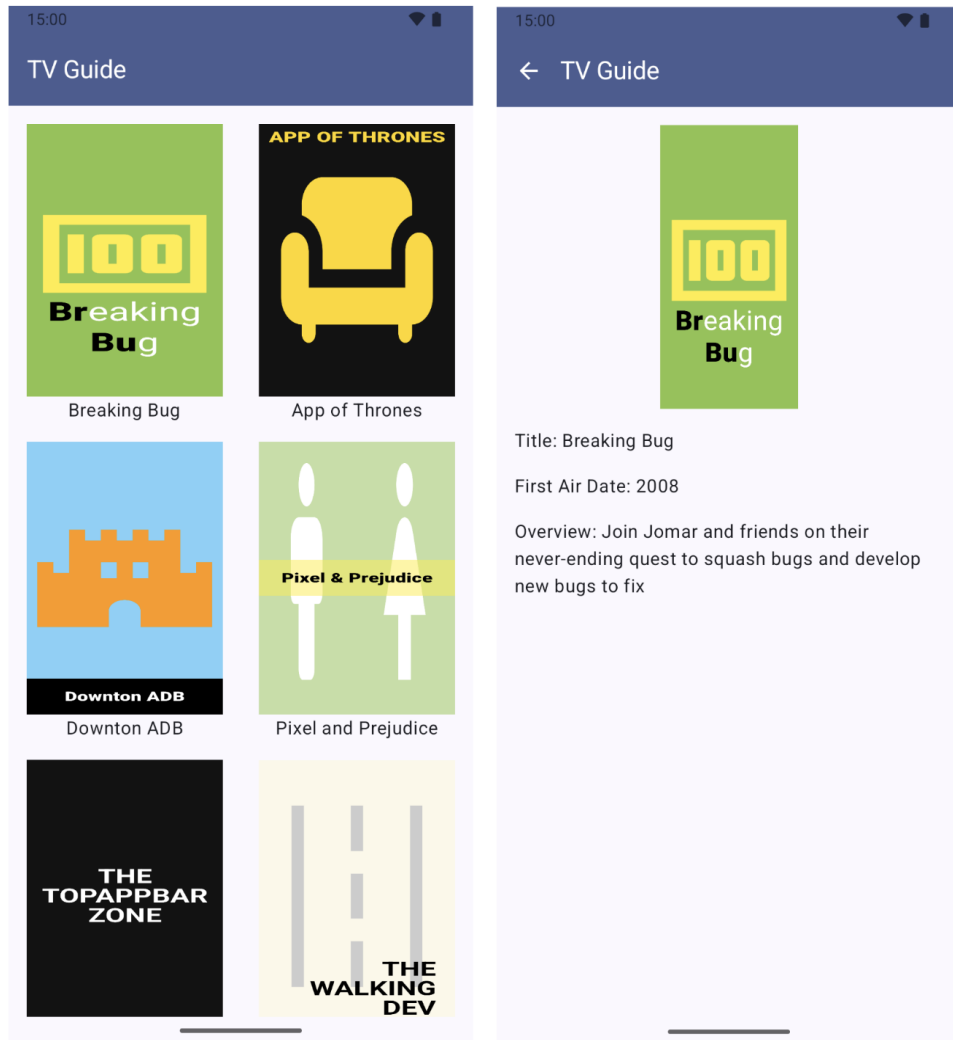


Figure 10.4 – The main screen and details screen of the TV guide app



The solution to this activity can be found at <https://packt.link/EpOE5>.

## Summary

This chapter focused on doing background operations with Coroutines and Flow. Background operations are used for long-running tasks such as accessing data from the local database or a remote server.

You started with the basics of using Kotlin Coroutines, Google's recommended solution for asynchronous programming. You learned that you can make a background task into a suspending function with the `suspend` keyword. Coroutines can be started with the `async` or `launch` keyword.

You learned how to create suspending functions and how to start coroutines. You also used dispatchers to change the thread where a coroutine runs. Then, you used Coroutines to perform network calls.

You then moved on to using Kotlin Flow in an Android app to load the data in the background. To safely collect flows in the UI layer, prevent memory leaks, and avoid wasting resources, you can use `Lifecycle.repeatOnLifecycle` and `Flow.flowWithLifecycle`.

You learned about using flow builders to create flows. The `flow` and `flowOf` builder functions allow you to create new flows from a function that emits values. You can use the `asFlow()` extension function to convert collections and functional types into a flow.

Finally, you explored flow operators and learned how to use them with Kotlin flows. Terminal operators are used to start the collection of the flow. With intermediate operators, you can transform a flow into another flow.

In the next chapter, you will learn about Android architecture components. You will focus on Android Jetpack libraries and other components that you can use in the Model-View-ViewModel pattern as you build more complex applications.

### Unlock this book's exclusive benefits now

Scan this QR code or go to [packtpub.com/unlock](https://packtpub.com/unlock), then search this book by name.

*Note: Keep your purchase invoice ready before you start.*

UNLOCK NOW





# 11

## Android Architecture Components

In this chapter, we will look at some of the most important architecture components used in the creation of Android apps. We will first look at `ViewModel` components, which help to decouple the business logic that our application uses from the **user interface (UI)** that we present to our users. When looking at `ViewModel` components, we will also analyze how we can combine them with additional libraries to handle more expensive tasks. We will also look at how we can use `ViewModel` components and the `SavedStateHandle` class to save the instance state.

We will then continue by looking at the Room persistence library, which we will use to save data on a device in a structured way, such as a database. Here, we will also look at how we can integrate third-party libraries to observe changes in our data.

By the end of the chapter, you should be able to create an Android app that can store data on a device and show it to the user, using the studied components.

In this chapter, we will cover the following topics:

- Understanding Android components' background
- Exploring `ViewModel`
- Combining `ViewModel` with data streams
- Saving instance states inside `ViewModels`
- Persisting data with Room

## Technical requirements

The complete code for all the exercises and the activity in this chapter is available on GitHub at <https://packt.link/ruDQ2>.

## Understanding Android components' background

In many situations, you must use `onSaveInstanceState` to save the current state of your activity or fragment, and then, in `onCreate` or `onRestoreInstanceState`, you need to restore the state of your activity or fragment. This adds extra complexity to your code and makes it repetitive, especially if the processing code is part of your activity or fragment.

These scenarios are where `ViewModel` and `LiveData` come in. `ViewModel` components are built with the express goal of holding data in case of life cycle changes. They also separate the logic from views, which makes them very easy to unit test. `LiveData` is a component used to hold data and notify observers when changes occur while taking their life cycle into account.

In simpler terms, the fragment only deals with views, `ViewModel` does the heavy lifting, and `LiveData` deals with delivering the results to the fragment, but only when the fragment is there and ready.

If you've ever used WhatsApp or a similar messaging app and you've turned off the internet, you'll have noticed that you are still able to use the application. The reason for this is that the messages are stored locally on your device. This is achieved with a database file called SQLite in most cases.

The Android framework already allows you to use this feature for your application. However, it requires a lot of boilerplate code to read and write data. Every time you want to interact with the local storage, you must write a SQL query. When you read the SQLite data, you must convert it into a Java/Kotlin object.

All of this requires a lot of code, time, and unit testing. What if someone else were to handle the SQLite connection, and all you had to do was focus on the code part? This is where **Room** comes in. This is a library that is a wrapper over SQLite. All you need to do is define how your data should be saved and let the library take care of the rest.

`ViewModel` components, `LiveData`, and `Room` are all part of the Android architecture components, which are part of the Android Jetpack libraries. The architecture components are designed to help developers structure their code, write testable components, and reduce boilerplate code.

Other architecture components include `Databinding` (which binds views with models or `ViewModel` components, allowing the data to be directly set in views), `WorkManager` (which allows developers to handle background work with ease), `Navigation` (which allows developers to create visual navigation graphs and specify relationships between activities and fragments), and `Paging` (which allows developers to load paginated data, which helps in situations where infinite scrolling is required).

In the following sections, we will go over these components in more detail, starting with `ViewModel` and the benefits it offers when writing an Android app.

## Exploring `ViewModel`

The `ViewModel` component is responsible for holding and processing data required by the UI. It has the benefit of surviving configuration changes that destroy and recreate fragments and activities, which allows it to retain the data that can then be used to re-populate the UI.

`ViewModel` components will eventually be destroyed when the activity or fragment is destroyed without being recreated or when the application process is terminated. This allows `ViewModel` to serve its responsibility and be garbage collected when it is no longer necessary. The only method `ViewModel` has is `onCleared()`, which is called when `ViewModel` terminates. You can use this method to terminate ongoing tasks and deallocate resources that will no longer be required.

Migrating data processing from activities into `ViewModel` helps create better and faster unit tests. Testing an activity requires an Android test to be executed on a device. Activities also have states, which means that your test should get the activity into the proper state for the assertions to work. `ViewModel` can be unit tested locally on your development machine and can be stateless, meaning that your data processing logic can be tested individually.

One of the most important features of `ViewModel` is that it allows communication between fragments. To communicate between fragments without `ViewModel`, you must make your fragment communicate with the activity, which will then call the fragment you wish to communicate with.

To achieve this with `ViewModel`, you can just attach it to the parent activity and use the same `ViewModel` component in the fragment you wish to communicate with. This will reduce the boilerplate code that was required previously.

In the following diagram, you can see that a `ViewModel` component can be created at any point in an activity's life cycle (in practice, they are normally initialized in `onCreate` for activities and `onCreateView` or `onViewCreated` for fragments because these represent the points where views are created and ready to be updated) and that once created, it will live as long as the activity does:

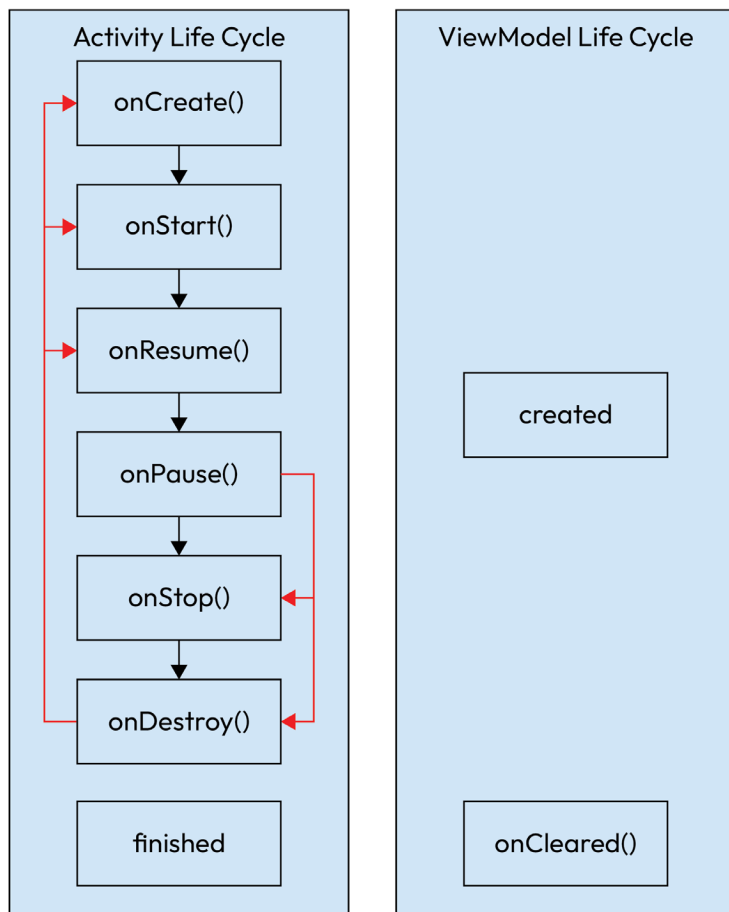


Figure 11.1 – The life cycle of an activity compared to the `ViewModel` life cycle

In the preceding diagram, we can see how the life cycle of `Activity` compares to that of `ViewModel`. The arrows on the sides indicate what happens when `Activity` is recreated, starting from the `onPause` method, ending in `onDestroy`, and then going from `onCreate` to `onResume` in a new instance of `Activity`.

The following diagram shows how `ViewModel` connects to a fragment:

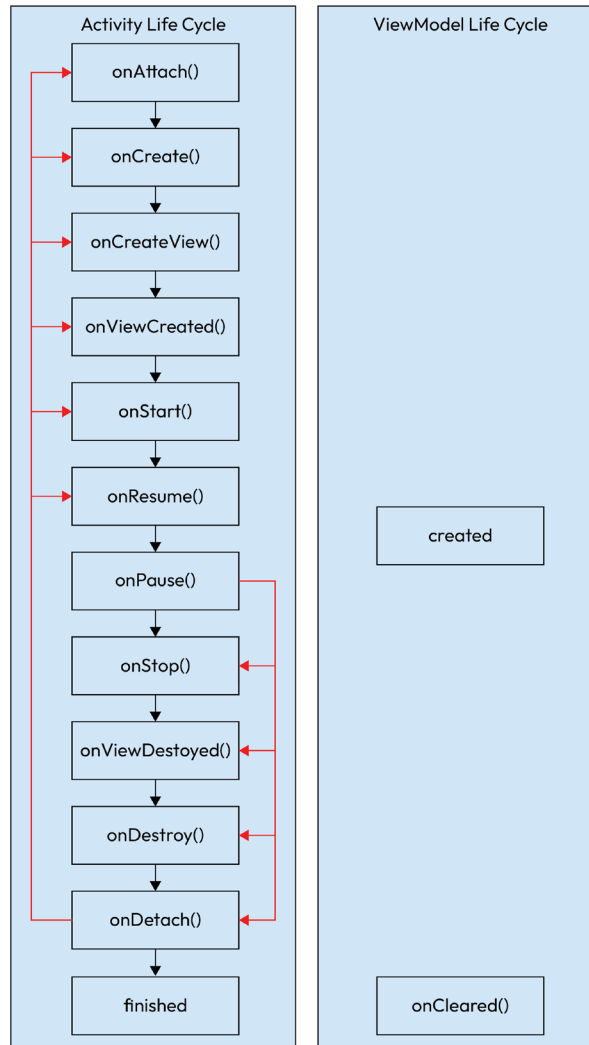


Figure 11.2 – The life cycle of a fragment compared to the `ViewModel` life cycle

In the preceding diagram, we can see how the life cycle of `Fragment` compares to that of `ViewModel`. The red lines indicate what happens when `Fragment` is recreated, starting from the `onPause` method, ending in `onDetach`, and then going from `onAttach` to `onResume` in a new instance of `Fragment`.

The `ViewModel` life cycle component is also compatible with `Compose`. The `ViewModel` component will be linked to the life cycle owner, which manages the `Compose` tree. So, if you have an `Activity` instance managing your `Compose` layout, then the `ViewModel` component will be linked to the



Activity instance, or if you have a Fragment instance, then the ViewModel component is linked to the Fragment instance. To make this available, you will need a dedicated library (androidx.lifecycle:lifecycle-viewmodel-compose:x.x.x), and then use the viewModel function:

```
@Composable
fun Screen(
 viewModel: ViewModel = viewModel()
) {
 viewModel.fetchData()
}
```

In this section, we learned what a ViewModel component is and the benefits it provides regarding testing and performing logic, which survives the recreation of the activity and fragment.

## Exercise 11.01 – Compose and ViewModel components

Create an application that displays one Text element and one Button element. When the button is pressed, the Text element will then display Total count x, where x is the number of clicks pressed. When the screen is rotated, then the total count text should retain the number of clicks. This will be done by holding the value of the total inside a ViewModel class.

Perform the following steps to solve the problem:

1. Create a new empty Activity component.
2. Make sure you added the ViewModel library for Compose to gradle/libs.versions.toml:

```
[versions]
...
viewModelCompose = "2.8.7"

[libraries]
...
viewmodel-compose = { group = "androidx.lifecycle", name =
 "lifecycle-viewmodel-compose", version.ref = "viewModelCompose"
}
```

3. Add the library to the app/build.gradle.kts file as a dependency:

```
dependencies {
 ...
 implementation(libs.viewmodel.compose)
}
```

4. Create a `TotalViewModel` class that will hold the total value and will increment the value for each click:

```
class TotalViewModel : ViewModel() {
 var result: Int = 0
 fun incrementResult() {
 result++
 }
}
```

5. Add the following strings to `res/values/strings.xml`:

```
<string name="click_me">Click Me</string>
<string name="total_count">Total count %s</string>
```

6. Create a screen holding a `Text` element and a `Button` element:

```
@Composable
fun MainScreen(
 modifier: Modifier,
 count: Int = 0,
 onButtonClick: () -> Unit
) {
 Column(
 modifier = modifier.fillMaxSize(),
 verticalArrangement = Arrangement.Center,
 horizontalAlignment = Alignment.CenterHorizontally
) {
 Text(text = stringResource(id = R.string.total_count,
 count))
 Button(onClick = onButtonClick) {
 Text(text = stringResource(id = R.string.click_me))
 }
 }
}
```

In the preceding code block, we have defined the screen for our UI elements. One thing to note is the fact that we have the `result` parameter in the signature of our function. This will be used later when we connect it to the `ViewModel` component.

7. Create a `main@Composable` function that will connect `TotalViewModel` with `MainScreen`, defined in the previous step:

```
@Composable@Composable
fun Main(
 totalViewModel: TotalViewModel = viewModel(),
 modifier: Modifier
) {
 var count by remember {
 mutableIntStateOf(totalViewModel.result)
 }
 MainScreen(modifier = modifier,
 count = count,
 onClick = {
 totalViewModel.incrementResult()
 count = totalViewModel.result
 })
}
```

In the preceding example, we have used the `viewModel` function to obtain an instance of `TotalViewModel`. Then, we have defined the `count` state, which will be initialized by whatever value is found in `ViewModel`. This is what allows the value to survive when the screen is rotated. In the `onClick` lambda, we then increment the value of the total and update the count state.

8. Finally, connect your `Main` function to that of `MainActivity`:

```
class MainActivity : AppCompatActivity() {
 override fun onCreate(savedInstanceState: Bundle?) {
 super.onCreate(savedInstanceState)
 enableEdgeToEdge()
 setContent {
 Exercise1101Theme {
 Scaffold(
 modifier = Modifier.fillMaxSize()
) { innerPadding ->
 Main(
 modifier = Modifier.padding(
 innerPadding
)
)
 }
 }
 }
 }
}
```

```
}
}
}
}
}
}
}
```

If we run the preceding example, we should see the following output:

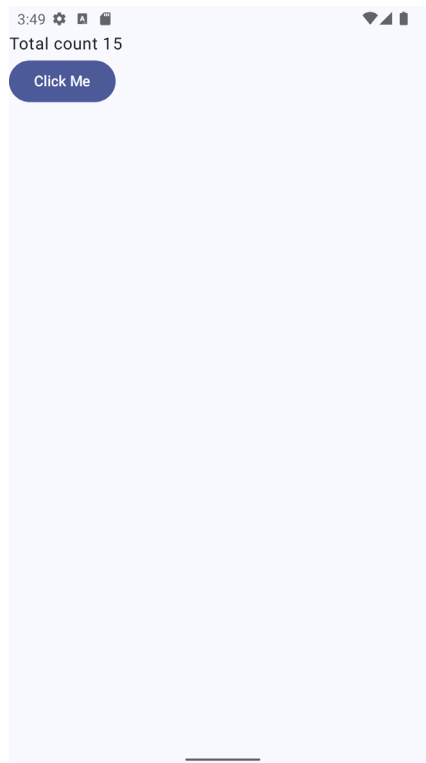


Figure 11.3 – Output of Exercise 11.01

We should now be able to see how clicking the button updates the value of the text, and if we rotate the screen, we should see the value preserved across configuration changes.

In this section, we have looked at how we can use `ViewModel` components to separate an application's logic from its UI and how the Android architecture component's `ViewModel` class can also help with configuration changes. In the following section, we will combine `ViewModel` with asynchronous data streams such as `LiveData` and coroutines to perform longer-running operations.

## Combining ViewModel with data streams

When it comes to data observability, we have multiple approaches for implementation, whether manually built mechanisms, components from the Java language, third-party components, or solutions developed particularly for Android. When it comes to Android, some of the most common solutions are LiveData, flows from coroutines' components, and RxJava.

The first one we will look at is LiveData, as it is part of the Android architecture components, which means that it is tailored specially to Android. We will then look at how we can use other types of data streams, such as coroutines and flows.

### LiveData

LiveData is a life cycle-aware component that permits updates to your UI, but only if the UI is in an active state (for example, if the activity or fragment is in one of the STARTED or RESUMED states). To monitor changes on LiveData, you need an observer combined with a LifecycleOwner instance. When the activity is set to an active state, the observers will be notified when changes occur.

If the activity is recreated, then the observer will be destroyed, and a new one will be reattached. Once this happens, the last value of LiveData will be emitted to allow us to restore the state. Activities and fragments are LifecycleOwner instances, but fragments have a separate LifecycleOwner instance for the view states. Fragments have this particular LifecycleOwner instance due to their behavior in the BackStack fragment.

When fragments are replaced within the back stack, they are not fully destroyed; only their views are. Some common callbacks that developers use to trigger processing logic are onCreateView(), onActivityResumed(), and onCreateView(). If we were to register observers for LiveData in these methods, we might end up with scenarios where multiple observers will be created every time our fragment pops back onto the screen.

When updating a LiveData model, we are presented with two options: setValue() and postValue(). The setValue() method will deliver the result immediately and is meant to be called only on the UI thread. On the other hand, postValue() can be called on any thread. When postValue() is called, LiveData will schedule an update of the value on the UI thread and update the value when the UI thread becomes free.

In the LiveData class, these methods are protected, which means that there are subclasses that allow us to change the data. MutableLiveData makes the methods public, which gives us a simple solution for observing data in most cases. MediatorLiveData is a specialized implementation of LiveData that allows us to merge multiple LiveData objects into one (this is useful in situations where our data is kept in different repositories and we want to show a combined result).

`TransformLiveData` is another specialized implementation that allows us to convert one object into another. This helps us in situations where we grab data from one repository and we want to request data from another repository that depends on the previous data, as well as in situations where we want to apply extra logic to a result from a repository.

`CustomLiveData` allows us to create our own `LivData` implementations (usually when we periodically receive updates, such as the odds in a sports betting app, stock market updates, and Facebook and Twitter/X feeds).



It is a common practice to use `LivData` in `ViewModel`. Holding `LivData` in a fragment or activity will cause losses in data when configuration changes occur.

The following diagram shows how `LivData` is connected to the life cycle of `LifecycleOwner`:

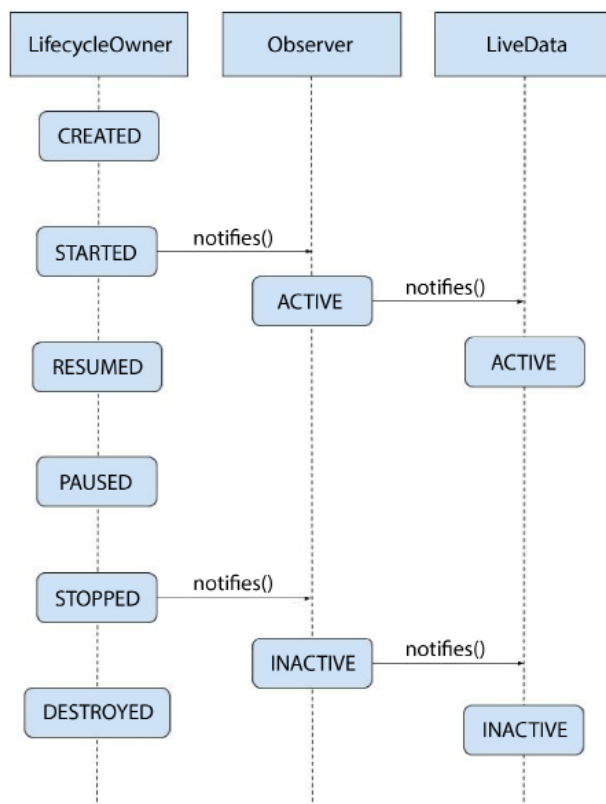


Figure 11.4 – The relationship between `LivData` and life cycle observers with `LifecycleOwner`



We can register multiple observers for LiveData, and each observer can be registered for a different LifecycleOwner instance. In this situation, LiveData will become inactive, but only when all the observers are inactive.

In terms of compatibility with Compose, the `observeAsState` extension function can be used on your LiveData data stream. This will convert the data stream into a Compose State object, meaning that changes on your LiveData stream will propagate a change to the state of the UI.

In this section, we looked at how the LiveData component works and the benefits it provides for observing data from activities and fragments about their life cycles. While it was commonly used in the past, it has since been phased out by developers and replaced with coroutines and flows, which we will look at in the next section.

## Coroutines and flows

Coroutines and flows can also be combined with a ViewModel component in the following way:

```
class TotalsViewModel : ViewModel() {
 private val _total = MutableStateFlow(0)
 val total: StateFlow<Int> = _total
 fun increaseTotal() {
 _total.value = _total.value + 1
 }
}
```

In the preceding snippet, we have the two total declarations for public and private usage. Instead of LiveData, we use StateFlow, which will emit the current value and all subsequent new values when we subscribe to it. Because it emits the last value, we must always set an initial value when we initialize it. If we want to subscribe to changes in the total value, we can use the following:

```
val totalsViewModel =
 ViewModelProvider(requireActivity())
 .get(TotalsViewModel::class.java)
viewLifecycleOwner.lifecycleScope.launch {
 repeatOnLifecycle(Lifecycle.State.CREATED) {
 totalsViewModel.total.collect {
 updateText(it)
 }
 }
}
```

```
 }
}
```

The preceding snippet will subscribe to `StateFlow` every time `viewLifecycleOwner` enters the `CREATED` stage. This will connect `StateFlow` with the life cycle of `Fragment` to prevent any possible leaks.

In terms of compatibility with `Compose`, the `collectAsState` extension function can be used on your `Flow` data stream. This will convert the data stream into a `Compose State` object, meaning that changes on your `Flow` stream will propagate a change to the state of the UI.

## Exercise 11.02 – Compose, ViewModel components, and flows

Modify *Exercise 11.01, Compose and ViewModel components*, in the following way. When the button is clicked to display the total count, we should also change the color of the text to red for odd numbers and blue for even numbers. Introduce a `MutableStateFlow` instance in `TotalViewModel` that will now manage and hold the count and the current text color.

Perform the following steps to solve the problem:

1. Add a `UiState` class in `TotalViewModel`, which will hold the total count and the current color:

```
data class UiState(
 val result: Int = 0,
 val textColor: Color = Color.Blue
)
```

2. Modify `TotalViewModel` to expose a `MutableStateFlow` instance that will hold `UiState`. Then, update the `incrementResult` method to increment the value of the result and change the text color if the number is odd or even:

```
class TotalViewModel : ViewModel() {

 private val _state = MutableStateFlow(UiState())
 val state: StateFlow<UiState> = _state

 fun incrementResult() {
 val newResult = _state.value.result + 1
 }
}
```



```

viewModelScope.launch {
 _state.emit(
 _state.value.copy(
 result = newResult,
 textColor =
 if (newResult.mod(2) == 0)
 Color.Blue
 else Color.Red
)
)
}
...
}

```

In the preceding snippet, we defined two states: `state` and `_state`. The `_state` state is meant to be internal to `TotalViewModel` so that no component other than `ViewModel` can change the state value. The state value is not a mutable variable, so external components can read the value and not modify it.

3. Modify the `MainScreen` function to use the new `UiState` class and set the right `Text` properties from the `UiState` object:

```

@Composable
fun MainScreen(
 modifier: Modifier,
 state: TotalViewModel.UiState,
 onClick: () -> Unit
) {
 Column(modifier = modifier) {
 Text(
 text = stringResource(
 id = R.string.total_count, state.result
),
 color = state.textColor
)
 Button(onClick = onClick) {

```

```
 Text(
 text = stringResource(
 id = R.string.click_me
)
)
 }
}
}
```

4. Modify the Main function to now consume StateFlow:

```
@Composable
fun Main(
 totalViewModel: TotalViewModel = viewModel(),
 modifier: Modifier
) {
 MainScreen(
 modifier = modifier,
 state = totalViewModel.state
 .collectAsState()
 .value,
 onClick = {
 totalViewModel.incrementResult()
 }
)
}
```

If we run the exercise, we should see a similar screen to the one from *Exercise 11.01, Compose and ViewModel components*:

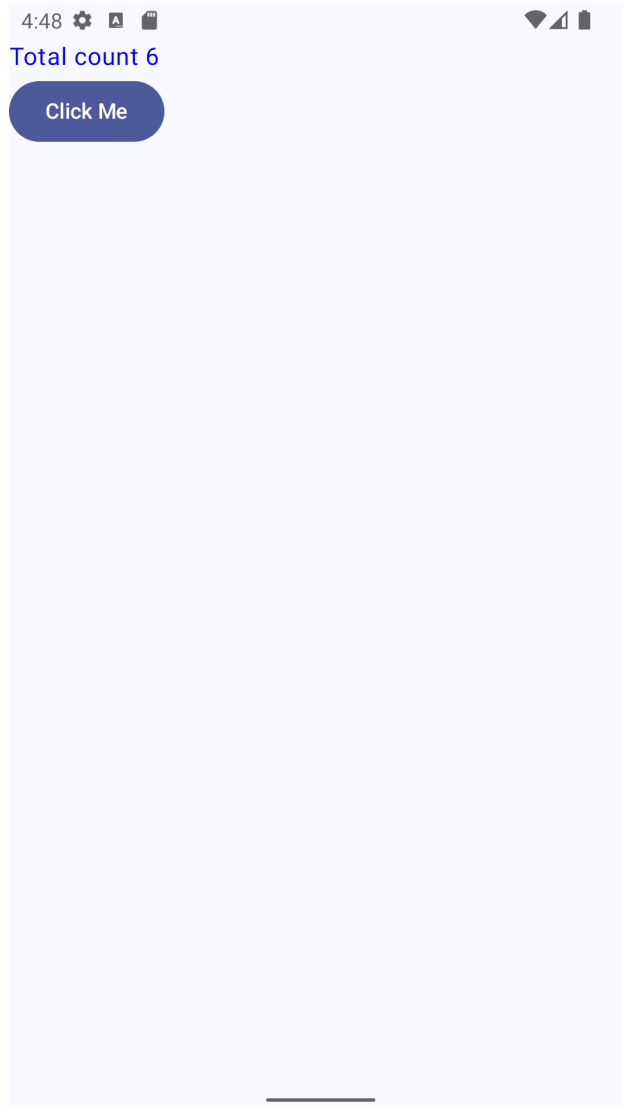


*Figure 11.5 – Output of Exercise 11.02*

Clicking the button will retain the same functionality as before, with the added feature of changing the color of the text.

In this section, we have looked at how we can combine data streams with `ViewModel` components and `Compose` to support asynchronous operations. In the following section, we will look at how we can use `ViewModel` to save the instance state.

## Saving instance states inside ViewModels

ViewModel components have the capability of saving the instance state, like how Activity and Fragment components can save their state through `onSaveInstanceState`, or how in Compose, we can use `rememberSaveable`. This can be done through the `SavedStateHandle` class and used in a ViewModel constructor, like in the following example:

```
class MyViewModel(
 private val savedStateHandle: SavedStateHandle
) : ViewModel() {
}
```

To store and retrieve data, we are presented with the `get` and `set` functions, each requiring a key for the data we wish to save:

```
private const val MY_KEY = "my_key"

class MyViewModel(
 private val savedStateHandle: SavedStateHandle
) : ViewModel() {

 var myValue: String = ""
 set(value) {
 field = value
 savedStateHandle[MY_KEY] = value
 }
 get() {
 return savedStateHandle
 .get<String>(MY_KEY).toString()
 }
}
```

In the preceding snippet, we have defined a variable called `myValue`, which will save its value in `savedStateHandle` when the value changes and will return the value from `savedStateHandle` when it's read.

Because `SavedStateHandle` objects are managed by the system, we will need to make sure that the system passes the appropriate `SavedStateHandle` instance to our ViewModels components. To facilitate this, we are provided with the `AbstractSavedStateViewModelFactory` class, instead of the usual `ViewModelProvider.Factory` class. In the `create` method from this new class, we

have the `SavedStateHandle` parameter, which we can pass to our `ViewModel` component. For example, in `Compose`, we would have the following:

```
val myViewModel = viewModel(
 factory = object: AbstractSavedStateViewModelFactory()
{
 override fun <T : ViewModel> create(
 key: String,
 modelClass: Class<T>,
 handle: SavedStateHandle
): T {
 return MyViewModel(handle) as T
 }
})
```

In the preceding snippet, we can see how we create `MyViewModel` in the `create` method, using the `handle` parameter. The limitations of what can be saved and restored from the instance state are the same ones as for `rememberSaveable` and `onSaveInstanceState`.

In this section, we have looked at how we can save the instance state inside a `ViewModel` component using `SavedStateHandle`. In the section that follows, we will look at how we can persist data on the device using `Room`.

## Persisting data with Room

The `Room` persistence library acts as a wrapper between your application code and the `SQLite` storage. You can think of `SQLite` as a database that runs without its own server and saves all the application data in an internal file that's only accessible to your application (if the device is not rooted).

`Room` sits between the application code and the `SQLite` Android framework and handles the necessary **create, read, update, and delete (CRUD)** operations while exposing an abstraction that your application can use to define the data and how you want the data to be handled.

This abstraction comes in the form of the following objects:

- **Entities:** You can specify how you want your data to be stored and the relationships between your data
- **Data access object (DAO):** The operations that can be done on your data
- **Database:** You can specify the configurations that your database should have (the name of the database and migration scenarios)

Let us assume you want to make a messaging app and store each message in your local storage. In this case, Entity would be a `Message` object, which will have an ID and will contain the contents of the message, the sender, the time, the status, and so on.

To access messages from the local storage, you will need `MessageDao`, which will contain methods such as `insertMessage()`, `getMessagesFromUser()`, `deleteMessage()`, and `updateMessage()`. In addition, since it is a messaging application, you will need a `Contact` entity to hold information about the senders and receivers of a message.

The `Contact` entity will contain information such as name, last time online, phone number, and email. To access the contact information, you will need a `ContactDao` interface, which will contain the `createUser()`, `updateUser()`, `deleteUser()`, and `getAllUsers()` methods. Both entities will create a matching table in SQLite, which contains the fields we defined inside the entity classes as columns. To achieve this, we will have to create a `MessagingDatabase` class in which we will reference both entities.

In a world without Room or similar DAO libraries, we would need to use the Android framework's SQLite components. This typically involves code when setting up our database, such as a query to create a table, and applying similar queries for every table we would have. Every time we would query a table for data, we would need to convert the resulting object into a Java or Kotlin one.

By default, Room does not allow any operations on the UI thread to enforce the Android standards related to **input-output (I/O)** operations. To make asynchronous calls to access data, Room is compatible with a number of libraries and frameworks, such as Kotlin coroutines, RxJava, and LiveData, on top of its default definitions.

We should now have an idea of how Room works and its main components. We will next look over each of these components and how we can use them for data persistence.

## Entities

Entities serve two purposes: to define the structure of tables and to hold the data from a table row. Let's use our scenario of the messaging app and define two entities: one for the user and one for the message.

The `User` entity will contain information about who sent messages, while the `Message` entity will contain information about the contents of a message, the time it was sent, and a reference to the sender of the message. The following code snippet provides an example of how entities are defined with Room:

```
@Entity(tableName = "messages")
data class Message(
```

```

 @PrimaryKey(autoGenerate = true)
 @ColumnInfo(name = "message_id") val id: Long,
 @ColumnInfo(name = "text", defaultValue = "") val text: String,
 @ColumnInfo(name = "time") val time: Long,
 @ColumnInfo(name = "user") val userId: Long,
)
@Entity(tableName = "users")
data class User(
 @PrimaryKey @ColumnInfo(name = "user_id") val id: Long,
 @ColumnInfo(name = "first_name") val firstName: String,
 @ColumnInfo(name = "last_name") val lastName: String,
 @ColumnInfo(name = "last_online") val lastOnline: Long
)

```

As you can see, entities are just *data classes* with annotations, which will tell Room how the tables should be built in SQLite. The annotations we used are as follows:

- The `@Entity` annotation defines the table. By default, the table name will be the name of the class. We can change the name of the table through the `tableName` method in the Entity annotation. This is useful in situations where we want our code to be obfuscated but wish to keep the consistency of the SQLite structure.
- `@ColumnInfo` defines configurations for a certain column. The most common one is the name of the column. We can also specify a default value, the SQLite type of the field, and whether the field should be indexed.
- `@PrimaryKey` indicates what in our entity will make it unique. Every entity should have at least one primary key. If your primary key is an integer or a long, then we can add the `autoGenerate` field. This means that every entity that gets inserted into the Primary Key field is automatically generated by SQLite.

Usually, this is done by incrementing the previous ID. If you wish to define multiple fields as primary keys, then you can adjust the `@Entity` annotation to accommodate this, such as the following:

```

@Entity(tableName = "messages", primaryKeys = ["id", "time"])

```

Let's assume that our messaging application wants to send locations. Locations have latitude, longitude, and name. We can add them to the `Message` class, but that would increase the complexity of the class. What we can do is create another entity and reference the ID in our class.

The problem with this approach is that we would then query the `Location` entity every time we queried the `Message` entity. Room has a third approach through the `@Embedded` annotation. Now, let's look at the updated `Message` entity:

```
@Entity(tableName = "messages")
data class Message(
 @PrimaryKey(autoGenerate = true) @ColumnInfo(name =
 "message_id") val id: Long,
 @ColumnInfo(name = "text", defaultValue = "") val text: String,
 @ColumnInfo(name = "time") val time: Long,
 @ColumnInfo(name = "user") val userId: Long,
 @Embedded val location: Location?
)
data class Location(
 @ColumnInfo(name = "lat") val lat: Double,
 @ColumnInfo(name = "long") val log: Double,
 @ColumnInfo(name = "location_name") val name: String
)
```

This code adds three columns (`lat`, `long`, and `location_name`) to the `messages` table. This allows us to avoid having objects with many fields while keeping our tables consistent.

If we look at our entities, we'll see that they exist independently. The `Message` entity has a `userId` field, but nothing is preventing us from adding messages from invalid users. This may lead to situations where we collect data without any purpose. If we want to delete a particular user, along with their messages, we must do so manually. Room provides us with a way to define this relationship using `ForeignKey`:

```
@Entity(
 tableName = "messages",
 foreignKeys = [ForeignKey(
 entity = User::class,
 parentColumns = ["user_id"],
 childColumns = ["user"],
 onDelete = ForeignKey.CASCADE
)]
)
data class Message(
 @PrimaryKey(autoGenerate = true) @ColumnInfo(name =
```



```

 "message_id") val id: Long,
 @ColumnInfo(name = "text", defaultValue = "") val text: String,
 @ColumnInfo(name = "time") val time: Long,
 @ColumnInfo(name = "user") val userId: Long,
 @Embedded val location: Location?
)

```

In the preceding example, we added the `foreignKeys` field and created a new `ForeignKey` instance to the `User` entity, which we set as the parent column, the `user_id` field in the `User` class, and for the child column, the `user` field in the `Message` class.

Every time we add a message to the table, there needs to be a `User` entry in the users table. If we try to delete a user and any messages from that user still exist, then, by default, this will not work because of the dependencies. However, we can tell Room to do a cascade delete, which will erase the user and the associated messages.

## DAO

If entities specify how we define and hold our data, then DAOs specify what to do with that data. A DAO class is a place where we define our CRUD operations. Ideally, each entity should have a corresponding DAO, but there are situations where crossovers occur (usually, this happens when we deal with JOIN operations between two tables).

Continuing with our previous example, let's build some corresponding DAOs for our entity:

```

@Dao
interface MessageDao {
 @Insert(onConflict = OnConflictStrategy.REPLACE)
 fun insertMessages(vararg messages: Message)

 @Update
 fun updateMessages(vararg messages: Message)

 @Delete
 fun deleteMessages(vararg messages: Message)

 @Query("SELECT * FROM messages")
 fun loadAllMessages(): List<Message>

 @Query("SELECT * FROM messages WHERE user=:userId AND
 time>=:time")
 fun loadMessagesFromUserAfterTime(
 userId: String, time: Long): List<Message>
}

```

```

@Dao
interface UserDao {
 @Insert(onConflict = OnConflictStrategy.REPLACE)
 fun insertUser(user: User)
 @Update
 fun updateUser(user: User)
 @Delete
 fun deleteUser(user: User)
 @Query("SELECT * FROM users")
 fun loadAllUsers(): List<User>
}

```

In the case of our messages, we have defined the following functions: insert one or more messages, update one or more messages, delete one or more messages, and retrieve all the messages from a certain user that are older than a particular time. For our users, we can insert one user, update one user, delete one user, and retrieve all users.

If you look at our `@Insert` methods, you'll see we have defined that in the case of a conflict (when we try to insert something with an ID that already exists), it will replace the existing entry. The `@Update` field has a similar configuration, but in our case, we have chosen the default. This means that nothing will happen if the update cannot occur.

The `@Query` annotation stands out from all the others. This is where we use SQLite code to define how our read operations work. `SELECT *` means we want to read all the data for every row in the table, which will populate all our entity fields. The `WHERE` clause indicates a restriction that we want to apply to our query. We can also define a method such as this:

```

@Query("SELECT * FROM messages WHERE user IN (:userIds) AND time>=:time")
fun loadMessagesFromUserAfterTime(userIds: List<String>, time: Long):
 List<Message>

```

This allows us to filter messages from multiple users. We can define a new class like this:

```

data class TextWithTime(
 @ColumnInfo(name = "text") val text: String,
 @ColumnInfo(name = "time") val time: Long
)

```

Now, we can define the following query:

```
@Query("SELECT text,time FROM messages")
fun loadTextsAndTimes(): List<TextWithTime>
```

This will allow us to extract information from certain columns at a time, not the entire row.

Now, let's say that you want to add the user information of the sender to every message. Here, we'll need to use a similar approach to the one we used previously for `TextWithTime` and create a new data class to hold the information we wish to extract:

```
data class MessageWithUser(
 @Embedded val message: Message,
 @Embedded val user: User
)
```

By using the new data class, we can define this query:

```
@Query("SELECT * FROM messages INNER JOIN users on
 users.user_id=messages.user")
fun loadMessagesAndUsers(): List<MessageWithUser>
```

We now have the user information for every message we want to display. This will come in handy in scenarios such as group chats, where we should display the name of the sender of every message.

## Setting up the database

What we have learned about so far is a bunch of DAOs and entities. Now, it's time to put them together. First, let's define our database:

```
@Database(entities = [User::class, Message::class],
 version = 1)
abstract class ChatDatabase : RoomDatabase() {
 companion object {
 private lateinit var chatDatabase: ChatDatabase
 fun getDatabase(
 applicationContext: Context
): ChatDatabase {
 if (!(::chatDatabase.isInitialized)) {
 chatDatabase = Room.databaseBuilder(
 applicationContext,
 chatDatabase::class.java,
```

```

 "chat-db"
).build()
}
return chatDatabase
}
}
abstract fun userDao(): UserDao
abstract fun messageDao(): MessageDao
}

```

In the `@Database` annotation, we specify what entities go in our database and our version. Then, for every DAO, we define an abstract method in `RoomDatabase`. This allows the build system to build a subclass of our class in which it provides the implementations for these methods. The build system will also create tables related to our entities.

The `getDatabase` method in the companion object illustrates how we create an instance of the `ChatDatabase` class. Ideally, there should be one instance of the database for our application due to the complexity involved in building a new database object. However, this can be better achieved through a **dependency injection (DI)** framework.

Let's assume you've released your chat application. Your database is currently version 1, but your users are complaining that the message status feature is missing. You decide to add this feature in the next release. This involves changing the database structure, which can impact databases that have already built their structures.

Luckily, Room offers something called a migration. In the migration, we can define how our database changed between versions 1 and 2. So, let's look at our example:

```

data class Message(
 @PrimaryKey(autoGenerate = true) @ColumnInfo(name =
 "message_id") val id: Long,
 @ColumnInfo(name = "text", defaultValue = "") val text: String,
 @ColumnInfo(name = "time") val time: Long,
 @ColumnInfo(name = "user") val userId: Long,
 @ColumnInfo(name = "status") val status: Int,
 @Embedded val location: Location?
)

```

In the preceding snippet, we added the status flag to the Message entity. Now, let's look at ChatDatabase:

```
Database(entities = [User::class, Message::class],
 version = 2)
abstract class ChatDatabase : RoomDatabase() {
 companion object {
 private lateinit var chatDatabase: ChatDatabase
 private val MIGRATION_1_2 = object : Migration(1,2) {
 override fun migrate(
 database: SupportSQLiteDatabase
) {
 database.execSQL(
 "ALTER TABLE messages ADD
 COLUMN status INTEGER"
)
 }
 }
 fun getDatabase(
 applicationContext: Context
): ChatDatabase {
 if (!::chatDatabase.isInitialized) {
 chatDatabase = Room.databaseBuilder(
 applicationContext,
 chatDatabase::class.java,
 "chat-db"
)
 .addMigrations(MIGRATION_1_2)
 .build()
 }
 return chatDatabase
 }
 }
}

abstract fun userDao(): UserDao
abstract fun messageDao(): MessageDao
}
```

In our database, we've increased the version to 2 and added a migration between versions 1 and 2. Here, we added the `status` column to the table. We'll add this migration when we build the database.

Once we've released the new code, when the updated app is opened and the code to build the database is executed, it will compare the version of the stored data with the one specified in our class and notice a difference. Then, it will execute the specified migrations until it reaches the latest version. This allows us to maintain an application for years without impacting the user's experience.

If you look at our `Message` class, you may have noticed that we defined the time as `Long`. In Java and Kotlin, we have the `Date` object, which may be more useful than the timestamp of the message. Luckily, Room has a solution for this in the form of `TypeConverter`.

The following table shows what data types we can use in our code and the SQLite equivalent. Complex data types need to be brought down to these levels using `TypeConverter` instances:

| Java/Kotlin                         | SQLite  |
|-------------------------------------|---------|
| String                              | TEXT    |
| Byte, Short, Integer, Long, Boolean | INTEGER |
| Double, Float                       | REAL    |
| Array<Byte>                         | BLOB    |

Figure 11.6 – The relationship between Kotlin/Java data types and SQLite data types

Here, we've modified the `lastOnline` field so that it's of the `Date` type:

```
data class User(
 @PrimaryKey @ColumnInfo(name = "user_id") val id: Long,
 @ColumnInfo(name = "first_name") val firstName: String,
 @ColumnInfo(name = "last_name") val lastName: String,
 @ColumnInfo(name = "last_online") val lastOnline: Date
)
```

Here, we've defined a couple of methods that convert a `Date` object into `Long` and vice versa. The `@TypeConverter` annotation helps Room identify where the conversion takes place:

```
class DateConverter {
 @TypeConverter
 fun from(value: Long?): Date? {
 return value?.let { Date(it) }
 }
}
```

```

 @TypeConverter
 fun to(date: Date?): Long? {
 return date?.time
 }
}

```

Finally, we'll add our converter to Room using the `@TypeConverters` annotation:

```

@Database(entities = [User::class, Message::class], version = 2)
@TypeConverters(DateConverter::class)
abstract class ChatDatabase : RoomDatabase() {

```

In the next section, we will look at some third-party frameworks.

## Third-party frameworks

Room works well with third-party frameworks such as LiveData, RxJava, and coroutines. This solves two issues: multithreading and observing data changes.

LiveData will make the `@Query`-annotated methods in your DAOs reactive, which means that if new data is added, LiveData will notify the observers of this:

```

@Query("SELECT * FROM users")
fun loadAllUsers(): LiveData<List<User>>

```

Kotlin coroutines complement LiveData by making the `@Insert`, `@Delete`, and `@Update` methods asynchronous:

```

@Insert(onConflict = OnConflictStrategy.REPLACE)
suspend fun insertUser(user: User)
@Update
suspend fun updateUser(user: User)
@Delete
suspend fun deleteUser(user: User)

```

RxJava solves both issues, making the `@Query` methods reactive through components such as Publisher, Observable, or Flowable and making the rest of the methods asynchronous through Completable, Single, or Maybe:

```

@Insert(onConflict = OnConflictStrategy.REPLACE)
fun insertUser(user: User) : Completable
@Update
fun updateUser(user: User) : Completable

```

```
@Delete
fun deleteUser(user: User) : Completable

@Query("SELECT * FROM users")
fun loadAllUsers(): Flowable<List<User>>
```

In this section, we have looked at how we can integrate third-party frameworks with Room, which can assist with concurrency and observing data changes for our tables. In the following section, we will implement an app that will use Room.

## Exercise 11.03 – making a little room

You have been hired by a news agency to build a news application. The application will display a list of articles written by journalists. An article can be written by one or more journalists, and each journalist can write one or more articles. The data information for each article includes the article's title, content, and date.

The journalist's information includes their first name, last name, and job title. You will need to build a Room database that holds this information so that it can be tested.

Before we start, let's look at the relationship between the entities. In the chat application example, we defined the rule that one user can send one or multiple messages.

This relationship is known as a one-to-many relationship. That relationship is implemented as a reference between one entity and another (the user was defined in the message table to be connected to the sender).

In this case, we have a many-to-many relationship. To implement a many-to-many relationship, we need to create an entity that holds references that will link the other two entities. Let's get started:

1. Create a new Android project with no activity.
2. Let's start by adding the symbol processing plugin to `gradle/libs.versions.toml`. This will read the annotations used by Room and generate the code necessary for interacting with the database:

```
[versions]
...
ksp = "2.0.21-1.0.25"
...
[plugins]
...
ksp = { id = "com.google.devtools.ksp", version.ref = "ksp" }
```



3. In the root `build.gradle.kts` file, add the `ksp` plugin:

```
plugins {
 ...
 alias(libs.plugins.ksp) apply false
}
```

4. In `app/build.gradle.kts`, add the `ksp` plugin:

```
plugins {
 ...
 alias(libs.plugins.ksp)
}
```

5. In `gradle/libs.versions.toml`, add the Room dependencies:

```
[versions]
...
room = "2.7.1"

[libraries]
...
room-runtime = { group = "androidx.room", name = "room-runtime",
 version.ref = "room" }
room-compiler = { group = "androidx.room", name = "room-compiler",
 version.ref = "room" }
```

6. Next, let's add the Room libraries in `app/build.gradle.kts`:

```
dependencies {
 ...
 implementation(libs.room.runtime)
 ksp(libs.room.compiler)
 ...
}
```

The first line defines the library version, the second line brings in the Room library for Java and Kotlin, and the last line is for the Kotlin annotation processor. This allows the build system to generate boilerplate code from the Room annotations. After these changes to your Gradle files, you should get a prompt to sync your project, which you should click.

7. Let's define our entities in the main/java folder and the root package:

```
@Entity(tableName = "article")
data class Article(
 @PrimaryKey(autoGenerate = true)
 @ColumnInfo(name = "id") val id: Long = 0,
 @ColumnInfo(name = "title") val title: String,
 @ColumnInfo(name = "content") val content: String,
 @ColumnInfo(name = "time") val time: Long
)
@Entity(tableName = "journalist")
data class Journalist(
 @PrimaryKey(autoGenerate = true)
 @ColumnInfo(name = "id") val id: Long = 0,
 @ColumnInfo(name = "first_name") val firstName: String,
 @ColumnInfo(name = "last_name") val lastName: String,
 @ColumnInfo(name = "job_title") val jobTitle: String
)
```

8. Now, define an entity that connects the journalist to the article and the appropriate constraints in the main/java folder and the root package:

```
@Entity(
 tableName = "joined_article_journalist",
 primaryKeys = ["article_id", "journalist_id"],
 foreignKeys = [ForeignKey(
 entity = Article::class,
 parentColumns = arrayOf("id"),
 childColumns = arrayOf("article_id"),
 onDelete = ForeignKey.CASCADE
), ForeignKey(
 entity = Journalist::class,
 parentColumns = arrayOf("id"),
 childColumns = arrayOf("journalist_id"),
 onDelete = ForeignKey.CASCADE
)]
)
data class JoinedArticleJournalist(
 @ColumnInfo(name = "article_id") val articleId: Long,
```

```
 @ColumnInfo(name = "journalist_id") val journalistId: Long
)
}
```

In the preceding code, we defined our connecting entity. As you can see, we haven't defined an ID for uniqueness, but both the article and the journalist will be unique when used together. We also defined foreign keys for each of the other entities referred to by our entity.

9. Create an ArticleDao DAO in the main/java folder and the root package:

```
@Dao
interface ArticleDao {
 @Insert(onConflict = OnConflictStrategy.REPLACE)
 fun insertArticle(article: Article)
 @Update
 fun updateArticle(article: Article)
 @Delete
 fun deleteArticle(article: Article)
 @Query("SELECT * FROM article")
 fun loadAllArticles(): List<Article>
 @Query("SELECT * FROM article INNER JOIN
 joined_article_journalist ON
 article.id=joined_article_journalist
 .article_id WHERE
 joined_article_journalist.journalist_id=
 :journalistId")
 fun loadArticlesForAuthor(journalistId: Long): List<Article>
}
```

10. Now, create a JournalistDao DAO in the main/java folder and the root package:

```
@Dao
interface JournalistDao {
 @Insert(onConflict = OnConflictStrategy.REPLACE)
 fun insertJournalist(journalist: Journalist)
 @Update
 fun updateJournalist(journalist: Journalist)
 @Delete
 fun deleteJournalist(journalist: Journalist)
 @Query("SELECT * FROM journalist")
 fun loadAllJournalists(): List<Journalist>
}
```

```

@Query("SELECT * FROM journalist INNER JOIN
 joined_article_journalist ON
 journalist.id=joined_article_journalist
 .journalist_id WHERE
 joined_article_journalist.article_id=
 :articleId")

fun getAuthorsForArticle(articleId: Long): List<Journalist>
}

```

11. Create a JoinedArticleJournalistDao DAO in the main/java folder and the root package:

```

@Dao
interface JoinedArticleJournalistDao {
 @Insert(onConflict = OnConflictStrategy.REPLACE)
 fun insertArticleJournalist(
 joinedArticleJournalist: JoinedArticleJournalist
)
 @Delete
 fun deleteArticleJournalist(
 joinedArticleJournalist: JoinedArticleJournalist
)
}

```

Let's analyze our code a little bit. For the articles and journalists, we can add, insert, delete, and update queries. For articles, we can extract all of the articles, but also extract articles from a certain author.

We also have the option to extract all journalists who wrote an article. This is done through a JOIN operation with our intermediary entity. For that entity, we define options to insert (which will link an article to a journalist) and delete (which will remove that link).

12. Finally, let's define our Database class in the main/java folder and the root package:

```

@Database(
 entities = [
 Article::class,
 Journalist::class,
 JoinedArticleJournalist::class
],
 version = 1
)

```

```

)
abstract class NewsDatabase : RoomDatabase() {
 abstract fun articleDao(): ArticleDao
 abstract fun journalistDao(): JournalistDao
 abstract fun joinedArticleJournalistDao():
 JoinedArticleJournalistDao
}

```

We avoided defining the `getInstance` method here because we won't be calling the database anywhere. But if we don't do that, how will we know whether it works? The answer to this is that we'll test it. This won't be a test that will run on your machine but one that will run on the device. This means that we will create it in the `androidTest` folder.

13. Let's start by setting up the test data. Here, we will add some articles and journalists to the database and then test retrieving, updating, and deleting the entries:

```

@RunWith(AndroidJUnit4::class)
class NewsDatabaseTest {
 @Test
 fun updateArticle() {
 val article = articleDao.loadAllArticles()[0]
 articleDao.updateArticle(
 article.copy(title = "new title")
)
 assertEquals(
 "new title",
 articleDao.loadAllArticles()[0].title
)
 }
 @Test
 fun updateJournalist() {
 val journalist = journalistDao.loadAllJournalists()[0]
 journalistDao.updateJournalist(
 journalist.copy(jobTitle = "new job title")
)
 assertEquals(
 "new job title",
 journalistDao.loadAllJournalists()[0]
 .jobTitle
)
 }
}

```

```
)
 }
}
```

The complete code for this step can be found at <https://packt.link/8F4wA>.

In this exercise, we have defined a few examples of how to test a Room database. What's interesting is how we build the database. Our database is an in-memory database. This means that all the data will be kept if the test is run and discarded afterward.

This allows us to start with a clean slate for each new state and avoids the consequences of each of our testing sessions affecting each other. In our test, we've set up 5 articles and 10 journalists. The first article was written by the top two journalists, while the second article was written by the first journalist.

The rest of the articles have no authors. By doing this, we can test our update and delete methods. For the delete method, we can test our foreign key relationship as well. In the test, we can see that if we delete article 1, it will delete the relationship between the article and the journalists who wrote it.

When testing your database, you should add the scenarios that your app will use. Feel free to add other testing scenarios and improve the preceding tests in your own database. Note that if you are using the `androidTest` folder, then this will be an instrumented test, meaning that you will need an emulator or a device to test.

In the following activity, we will create an app that will combine all the topics from the chapter to show how we can go from storing data to displaying it to the user.

## Activity 11.01 – shopping notes app

Create an Android app that will display a `TextField` component, a `Button` element, a `Text` element, and a list. When the `Button` element is clicked, a new note will be created and persisted using Room. The note will contain the text from `TextInput`. The `Text` element will display the total count of notes, and the list will display all notes created.

Perform the following steps to complete the activity:

1. Create a new empty `Activity` component.
2. Add the Room, `ViewModel`, and **Kotlin Symbol Processing (KSP)** configurations to the Gradle files.
3. Create an entity, a DAO, and the database responsible for managing the notes.

4. Create `NoteRepository` and `NoteImplementation` methods, which will deal with interacting with `NoteDao`.
5. Create a `NoteViewModel` method that will be responsible for loading the data from the repository and converting it into a `UiState` object, which will then be set on the UI.
6. Create a `NoteScreen` method that will show the UI elements specified in the requirement based on the `UiState` object defined previously.
7. Create a `Note` method that will take the data from `NoteViewModel` and then invoke the `NoteScreen` method defined previously.
8. In your `Activity` component, create a `NoteDatabase` method, then call the `Note` method defined previously. You will need to use the factory parameter of the `viewModel` function in your `@Composable` function to pass the `NoteDao` instance from `NoteDatabase` to `NoteViewModel`.



The solution to this activity can be found at <https://packt.link/swrAS>.

## Summary

In this chapter, we analyzed the building blocks required to build a maintainable application. We also investigated one of the most common issues that developers come across when using the Android framework, which is maintaining the states of objects during life cycle changes.

We started by analyzing `ViewModel` components and how they solve the issue of holding data during orientation changes. We added `Flow` data streams to `ViewModel` components to show how the two complement each other, and looked at how we can use other data streams with `ViewModel` components and compare those with `Flow` data streams.

We also looked at how we can save the instance state using `ViewModel` components in combination with `SavedStateHandle`, which transfers the responsibility of saving states from `Activity` and `Fragment` components to `ViewModel` components.

We then moved on to `Room` to show how we can persist data with minimal effort and without much `SQLite` boilerplate code. We also explored one-to-many and many-to-many relationships, as well as how to migrate data and break down complex objects into primitives for storage.

The activity we completed in this chapter serves as an example of what direction Android apps are heading in. However, this was not a complete example due to the numerous frameworks and libraries that you will discover, which give developers the flexibility to go in different directions.

The information you've learned in this chapter will serve you well for the next one, which will expand on the concept of repositories. This will allow you to save data that's been obtained from a server into a Room database. The concept of persisting data will also be expanded as you explore other ways to persist data, such as through `SharedPreferences`, `DataStore`, and files. Our focus will be on certain types of files: media files obtained from the camera of a device.

### Unlock this book's exclusive benefits now

Scan this QR code or go to [packtpub.com/unlock](https://packtpub.com/unlock), then search this book by name.

*Note: Keep your purchase invoice ready before you start.*

UNLOCK NOW







# 12

## Persisting Data

In the previous chapter, you learned how to structure your code and save data. In the activity section, *Activity 11.01 – a shopping notes app*, you also had the opportunity to build a repository and use it to access and save data through Room. In this chapter, you will learn about alternative ways to store and access data on an Android device’s internal and external memory.

You’ll develop your understanding of read and write permissions, learn how to create the `FileProvider` class to offer other apps access to your files, and learn how you can save those files without requesting permissions on external drives. You’ll also see how to download files from the internet and save them on a filesystem.

Another concept that will be explored in this chapter is using the **Camera** application to take photos and videos on your application’s behalf and save them to external storage using `FileProvider`.

This chapter goes in depth about data persistence in Android. By the end of the chapter, you will know multiple ways to store (persist) data directly on a device and the frameworks accessible to do this. When dealing with a filesystem, you will know how it’s partitioned and how you can read and write files in different locations and use different frameworks.

We will cover the following topics in the chapter:

- Using `SharedPreferences` and `DataStore`
- Saving data into files
- Understanding scoped storage

### Technical requirements

The complete code for all the exercises and the activity in this chapter is available on GitHub at <https://packt.link/EQRyT>.

## Using SharedPreferences and DataStore

Imagine you are tasked with integrating a third party that uses something such as OAuth to implement logging in with Facebook, Google, and so on. These mechanisms work as follows: they give you a token to store locally, which can be used to send other requests to access user data.

This raises several questions. How can you store that token? Do you use Room just for one token? Do you save the token in a separate file and implement methods for writing the file? What if that file must be accessed in multiple places at the same time? SharedPreferences and DataStore are the answers to these questions. SharedPreferences is a functionality that allows you to save Booleans, integers, floats, longs, strings, and sets of strings into an XML file.

When you want to save new values, you specify what values you want to save for the associated keys, and when you are done, you commit the change, which will trigger the save to the XML file in an asynchronous way. The SharedPreferences mappings are also kept in memory so that when you want to read these values, it's instantaneous, thereby removing the need for an asynchronous call to read the XML file.

We now have two ways to store data in key-value pairs in the form of SharedPreferences and DataStore. We will now look at how each of them works and the benefits each one provides.

### SharedPreferences

The way to access the SharedPreferences object is through the Context object:

```
val prefs = getSharedPreferences("my-prefs-file", Context.MODE_PRIVATE)
```

The first parameter is where you specify the name of your preferences, and the second is how you want to expose a file to other apps. Currently, the best mode is the private one. All the others present potential security risks.

If you want to write data into your SharedPreferences file, you first need to get access to the SharedPreferences editor. The editor will give you access to write the data. You can then write your data in it. Once you finish writing, you will have to apply the changes that will trigger persistence to the XML file and change the in-memory values as well.

You have two choices to apply the changes on your `SharedPreferences` file – `apply` or `commit`. Choosing `apply` will save your changes in memory instantly, but then writing to disk will be asynchronous, which is useful if you want to save data from your app's main thread. Note that `commit` does everything synchronously and gives you a `Boolean` result, informing you whether the operation was successful or not. In practice, `apply` tends to be favored over `commit`:

```
val editor = prefs.edit()
editor.putBoolean("my_key_1", true)
editor.putString("my_key_2", "my string")
editor.putLong("my_key_3", 1L)
editor.apply()
```

Now, you want to clear your entire data. The same principle will apply; you'll need `editor`, `clear`, and `apply`:

```
val editor = prefs.edit()
editor.clear()
editor.apply()
```

If you want to read the values you previously saved, you can use the `SharedPreferences` object to read the stored values. If there is no saved value, you can opt for a default value to be returned instead:

```
prefs.getBoolean("my_key_1", false)
prefs.getString("my_key_2", "")
prefs.getLong("my_key_3", 0L)
```

We should now have an idea about how we can persist data with `SharedPreferences`, and we can apply this in an exercise in the following section.

## Exercise 12.01 – wrapping `SharedPreferences`

We're going to build an application that displays `Text`, `TextField`, and `Button` UI elements. `Text` will display the previously saved value in `SharedPreferences`. The user can type new text, and when the button is clicked, the text will be saved in `SharedPreferences` and `TextView` will display the updated text. We will need to use `ViewModel` and coroutines to make the code more testable.

To complete this exercise, we will need to create a `Wrapper` class, which will be responsible for saving the text. This class will return the value of the text as a `Flow` object. This will be injected into our `ViewModel` object, which will be bound to the activity:

1. Create a new Android Studio project with an empty activity.

2. Let's begin by adding the appropriate libraries to `gradle/libs.versions.toml`:

```
[versions]
...
viewModelCompose = "2.8.7"

[libraries]
...
androidx-viewmodel-compose = { group = "androidx.lifecycle", name =
 "lifecycle-viewmodel-compose", version.ref = "viewModelCompose"
}
```

3. Add the ViewModel dependency to `app/build.gradle.kts`:

```
implementation(libs.androidx.viewmodel.compose)
```

4. Let's make our `PreferenceWrapper` class. Here, we will use `MutableStateFlow`, which is initialized with the last value from `SharedPreferences`. When a new value persists, then it will be saved in `SharedPreferences` and emitted to the consumers of `textFlow`:

```
const val KEY_TEXT = "keyText"

class PreferenceWrapper(private val sharedPreferences:
 SharedPreferences) {

 private val _textFlow =
 MutableStateFlow(sharedPreferences.getString(
 KEY_TEXT, "").orEmpty())
 val textFlow: StateFlow<String> = _textFlow

 suspend fun saveText(text: String) {
 sharedPreferences.edit()
 .putString(KEY_TEXT, text)
 .apply()
 _textFlow.emit(text)
 }
}
```

The usage of `_textFlow` and `textFlow` is to prevent consumers of `Flow` from being modified by external components, which is achieved by preventing `MutableStateFlow` from being made public.

- Next, let's build the `PreferenceViewModel` class:

```
class PreferenceViewModel(private val preferenceWrapper:
 PreferenceWrapper) : ViewModel() {

 private val _state = MutableStateFlow(UiState())
 val state: StateFlow<UiState> = _state

 init {
 viewModelScope.launch {
 preferenceWrapper.textFlow.collect {
 _state.emit(UiState(text = it))
 }
 }
 }

 fun saveText(text: String) {
 viewModelScope.launch {
 preferenceWrapper.saveText(text)
 }
 }

 data class UiState(
 val text: String = ""
)
}
```

In the preceding code block, we are consuming the `textFlow` object from the `PreferenceWrapper` class, converting it to a `UiState` object, and emitting it, which will later be consumed by the UI.

- Add the following string to `res/values/strings.xml`:

```
<string name="click_me">Click Me</string>
```

- Create the `PreferenceScreen @Composable` function, which will contain all the UI elements:

```
@Composable
fun PreferenceScreen(
```

```

 uiState: PreferenceViewModel.UiState,
 onClicked: (String) -> Unit,
 modifier: Modifier = Modifier
) {
 Column(
 modifier = modifier.fillMaxSize(),
 verticalArrangement = Arrangement.Center,
 horizontalAlignment = Alignment.CenterHorizontally
) {
 var textFieldText by remember {
 mutableStateOf("")
 }
 TextField(value = textFieldText, onValueChange = {
 textFieldText = it
 })
 Button(onClick = {
 onClicked(textFieldText)
 }) {
 Text(text = stringResource(id =
 R.string.click_me))
 }
 Text(text = uiState.text)
 }
 }
}

```

8. Create the `Preference@Composable` function, which will connect `PreferenceViewModel` with `PreferenceScreen`:

```

@Composable
fun Preference(
 preferenceViewModel: PreferenceViewModel,
 modifier: Modifier = Modifier
) {
 PreferenceScreen(
 uiState =
 preferenceViewModel.state.collectAsState()
 .value,
)
}

```





```
 modifier = Modifier.padding(innerPadding)
)
}
}
}
}
```

The preceding code will produce the output presented in *Figure 12.1*:

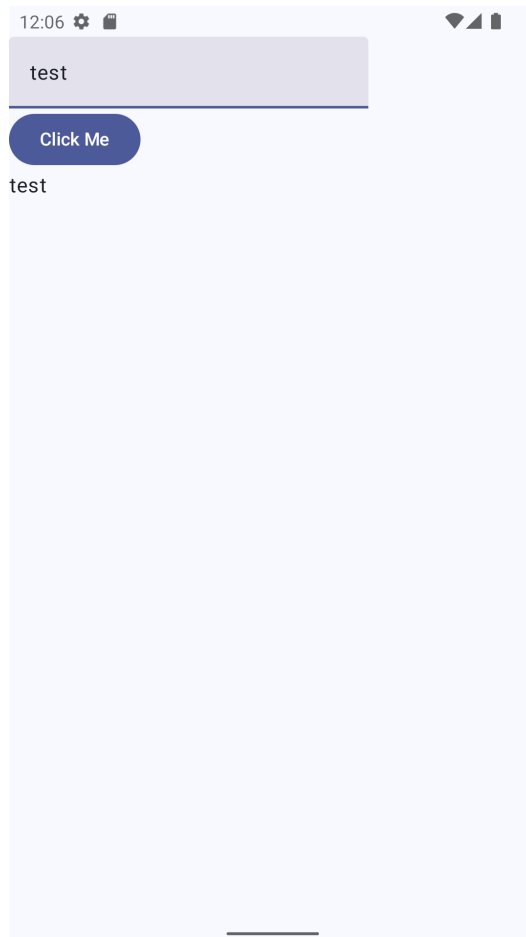


*Figure 12.1 – Output of Exercise 12.01*

Once you insert a value, try closing the application and reopening it. The app will display the last persisted value.

In this exercise, we have saved data on the device in the form of key-value pairs using `SharedPreferences`, which has the advantage of handling the actual write and read operations by the Android framework. It does, however, become more difficult when it comes to integration with `Compose`, which requires the creation of data streams to be consumed by `@Composable` functions. In the following section, we analyze `DataStore` and how it simplifies integration with data streams.

## DataStore

The `DataStore` persistence library represents an alternative to `SharedPreferences` when we want to store data in key-value pairs through `Preferences DataStore`, or if we want to store entire objects through `Proto DataStore`. Both libraries avoid dependencies with the Android framework (unlike `SharedPreferences`, which requires a `Context` object to be initialized) and are built using coroutines and flows, making them the ideal candidate when coroutines and flows are used in your project.

This integration allows `DataStore` to notify subscribers of all changes, which means that developers no longer have to concern themselves with handling the changes:

```
val Context.dataStore: DataStore<Preferences> by
 preferencesDataStore(name = "myDataStore")
val KEY_MY_INT = intPreferencesKey("my_int_key")
val KEY_MY_BOOLEAN =
 booleanPreferencesKey("my_boolean_key")
val KEY_MY_STRING = stringPreferencesKey("my_string_key")
class MyAppSettings(private val context: Context) {
 val myIntValue: Flow<Int> = context.dataStore.data
 .map { preferences ->
 preferences[KEY_MY_INT] ?: 0
 }
 val myBooleanValue: Flow<Boolean> =
 context.dataStore.data
 .map { preferences ->
 preferences[KEY_MY_BOOLEAN] ?: false
 }
 val myStringValue: Flow<String> =
 context.dataStore.data
 .map { preferences ->
 preferences[KEY_MY_STRING] ?: ""
 }
}
```

In the preceding snippet, we initialize `Context.dataStore` in the top-level Kotlin file. We then define three separate keys for the separate types we want to read from. Inside `MyAppSettings`, we map the values from `context.dataStore.data` and extract the values from our keys.

If we want to store data in `DataStore`, then we need to do the following:

```
class MyAppSettings(private val context: Context) {
 ...
 suspend fun saveMyIntValue(intValue: Int) {
 context.dataStore.edit { preferences ->
 preferences[KEY_MY_INT] = intValue
 }
 }

 suspend fun saveMyBooleanValue(booleanValue: Boolean) {
 context.dataStore.edit { preferences ->
 preferences[KEY_MY_BOOLEAN] = booleanValue
 }
 }

 suspend fun saveMyStringValue(stringValue: String) {
 context.dataStore.edit { preferences ->
 preferences[KEY_MY_STRING] = stringValue
 }
 }
}
```

The `suspend` keyword comes from coroutines, and it signals that we need to place the method invocation into an asynchronous call. Note that `context.dataStore.edit` will make the preferences in `DataStore` mutable and allow us to change the values.

## Exercise 12.02 – Preferences DataStore

Modify *Exercise 12.01 – wrapping `SharedPreferences`* so that `DataStore` is used instead of `SharedPreferences`:

1. Let's begin by adding the appropriate libraries to `gradle/libs.versions.toml`:

```
[versions]
...
dataStore = "1.1.5"

[libraries]
```

```
...
androidx-data-store-preferences = { group = "androidx.datastore",
 name = "datastore-preferences", version.ref = "dataStore" }
```

2. Add the DataStore dependency to app/build.gradle.kts:

```
implementation(
 libs.androidx.data.store.preferences)
```

3. Delete the PreferenceWrapper file.
4. Create a new class called PreferenceStore, which will be responsible for managing the DataStore interactions:

```
val Context.dataStore:
 DataStore<Preferences> by preferencesDataStore(
 name = "settingsStore"
)

val KEY_TEXT = stringPreferencesKey("key_text")

class PreferenceStore(private val context: Context) {

 val textFlow: Flow<String> = context.dataStore.data
 .map { preferences ->
 preferences[KEY_TEXT] ?: ""
 }

 suspend fun saveText(text: String) {
 context.dataStore.edit { preferences ->
 preferences[KEY_TEXT] = text
 }
 }
}
```

5. Modify PreferenceViewModel to use PreferenceStore instead of PreferenceWrapper:

```
class PreferenceViewModel(private val preferenceStore:
 PreferenceStore) : ViewModel() {
 ...
 init {
```



```
 })
 ...
 }
}
}
}
```

If we now run the application, we should see the following screen:

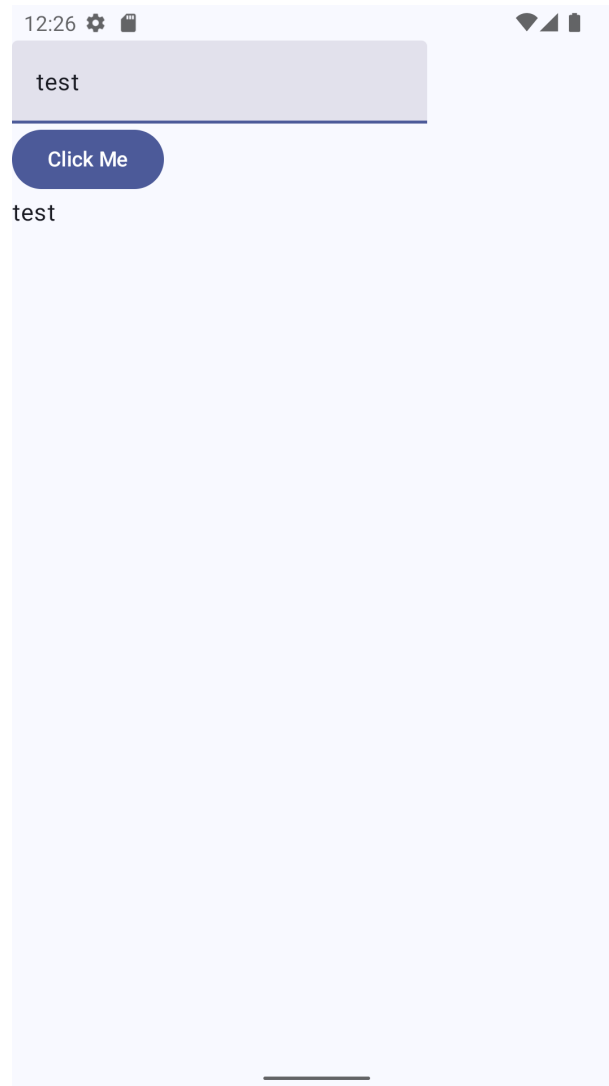


Figure 12.2 – Output of Exercise 12.02

If we enter new text and click the **Click Me** button, we see that the text is instantly updated, unlike in the previous exercise. This is because of the usage of flows and how `DataStore` will emit a new value for each change. We will look at flows and other reactive streams in future chapters.

In this exercise, we have looked at how the `DataStore` library works and its benefits, especially regarding streams of data. In the following sections, we will continue to look at other ways of persisting data, using files.

## Saving data into files

We've discussed `Room`, `SharedPreferences`, and `DataStore` and specified how the data they store is written to files. You may ask yourself where these files are stored. These files are stored in internal storage. Internal storage is a dedicated space for every app that other apps are unable to access (unless a device is rooted). There is no limit to the amount of storage your app can use.

However, users can delete their app's files from the **Settings** menu. Internal storage occupies a smaller part of the total available space, which means that you should be careful when it comes to storing files there. There is also external storage. The files your app stores in external storage are accessible to other apps and the files from other apps are accessible to your app.



In Android Studio, you can use the **Device File Explorer** tool to navigate through the files on a device or emulator. Internal storage is in `/data/data/{packageName}`. If you have access to this folder, this means that the device is rooted. Using this, you can visualize the database files and the `SharedPreferences` files.

An example of how **Device File Explorer** looks can be viewed in the following screenshot:

Pixel 2 API 32 3 Android 12L ("Sv2")

Files Processes

| Name                         | Permissions | Date             | Size |
|------------------------------|-------------|------------------|------|
| > com.google.android.provic  | drwxrwx--x  | 2024-10-13 17:22 | 4 KB |
| > com.google.android.sdkse   | drwxrwx--x  | 2024-10-13 17:22 | 4 KB |
| > com.google.android.settin  | drwxrwx--x  | 2024-10-13 17:22 | 4 KB |
| > com.google.android.setup   | drwxrwx--x  | 2024-10-13 17:22 | 4 KB |
| > com.google.android.sound   | drwxrwx--x  | 2024-10-13 17:22 | 4 KB |
| > com.google.android.synca   | drwxrwx--x  | 2024-10-13 17:22 | 4 KB |
| > com.google.android.syster  | drwxrwx--x  | 2024-10-13 17:22 | 4 KB |
| > com.google.android.tag     | drwxrwx--x  | 2024-10-13 17:22 | 4 KB |
| > com.google.android.tts     | drwxrwx--x  | 2024-10-13 17:22 | 4 KB |
| > com.google.android.webvi   | drwxrwx--x  | 2024-10-13 17:22 | 4 KB |
| > com.google.android.wifi.re | drwxrwx--x  | 2024-10-13 17:22 | 4 KB |
| > com.google.android.youtu   | drwxrwx--x  | 2024-10-13 17:22 | 4 KB |
| ✓ com.packt.android          | drwxrwx--x  | 2024-10-13 17:22 | 4 KB |
| ✓ cache                      | drwxrws--x  | 2025-01-08 20:59 | 4 KB |
| ✓ code_cache                 | drwxrws--x  | 2025-01-08 20:59 | 4 KB |
| ✓ files                      | drwxrwx--x  | 2025-01-08 21:00 | 4 KB |
| > datastore                  | drwx-----   | 2025-01-08 21:00 | 4 KB |
| profileInstalled             | -rw-----    | 2025-01-08 20:59 | 24 B |
| > org.chromium.webview_shi   | drwxrwx--x  | 2024-10-13 17:22 | 4 KB |
| > local                      | drwxrwx--x  | 2024-10-13 17:22 | 4 KB |

Figure 12.3 – Android’s Device File Explorer for an emulated device

**Quick tip:** Need to see a high-resolution version of this image? Open this book in the next-gen Packt Reader or view it in the PDF/ePub copy.

**The next-gen Packt Reader** and a **free PDF/ePub copy** of this book are included with your purchase. Scan the QR code OR visit [packtpub.com/unlock](https://packtpub.com/unlock), then use the search bar to find this book by name. Double-check the edition shown to make sure you get the right one.





We can classify the Android filesystem into two types: **internal** and **external**. These indicate how you want your application's files to be accessed by the user and other apps.

## Internal storage

Internal storage requires no permissions from the user. To access the internal storage directories, you can use one of the following methods from the Context object:

- `getDataDir()`: Returns the root folder of your application sandbox.
- `getFilesDir()`: A dedicated folder for application files – recommended for usage.
- `getCacheDir()`: A dedicated folder where files can be cached. Storing files here does not guarantee that you can retrieve them later because the system may decide to delete this directory to free up memory. This folder is linked to the **Clear Cache** option in **Settings**.
- `getDir(name, mode)`: Returns a folder that will be created if it does not exist, based on the name specified.

When users use the **Clear Data** option in **Settings**, most of these folders will be deleted, bringing the app to a similar state as a fresh installation. When the app is uninstalled, these files will be deleted as well.

A typical example of reading a cache file is as follows:

```
val cacheDir = context.cacheDir
val fileToReadFrom = File(cacheDir, "my-file.txt")
val size = fileToReadFrom.length().toInt()
val bytes = ByteArray(size)
val tmpBuff = ByteArray(size)
val fis = FileInputStream(fileToReadFrom)
try {
 var read = fis.read(bytes, 0, size)
 if (read < size) {
 var remain = size - read
 while (remain > 0) {
 read = fis.read(tmpBuff, 0, remain)
 System.arraycopy(tmpBuff, 0, bytes,
 size - remain, read)
 remain -= read
 }
 }
}
```

```
 } catch (e: IOException) {
 throw e
 } finally {
 fis.close()
 }
}
```

The preceding example will read from `my-file.txt`, located in the `Cache` directory, and will create `FileInputStream` for that file. Then, a buffer will be used that will collect the bytes from the file. The collected bytes will be placed in the `bytes` byte array, which will contain all of the data read from that file. Reading will stop when the entire length of the file has been read.

Writing to the `my-file.txt` file will look something like this:

```
val bytesToWrite = ByteArray(100)
val cacheDir = context.cacheDir
val fileToWriteIn = File(cacheDir, "my-file.txt")
try {
 if (!fileToWriteIn.exists()) {
 fileToWriteIn.createNewFile()
 }
 val fos = FileOutputStream(fileToWriteIn)
 fos.write(bytesToWrite)
 fos.close()
} catch (e: Exception) {
 e.printStackTrace()
}
```

What the preceding example does is take the byte array you want to write, create a new `File` object, create the file if it doesn't exist, and write the bytes into the file through `FileOutputStream`.



There are many alternatives to dealing with files. Readers (`StreamReader`, `StreamWriter`, and so on) are better equipped for character-based data. There are also third-party libraries that help with disk **input/output (I/O)** operations. One of the most common third parties that help with I/O operations is called **Okio**. It started life as part of the `OkHttp` library, which is used in combination with `Retrofit` to make API calls. The methods provided by `Okio` are the same methods it uses to write and read data in HTTP communications.

## External storage

Reading and writing in external storage requires the user to have permission for reading and writing. If the write permission is granted, then your app has the ability to read the external storage. Once these permissions are granted, then your app can do whatever it pleases on the external storage.

That may present a problem because users may not choose to grant these permissions. However, there are specialized methods that offer you the possibility to write to the external storage in folders dedicated to your application.

Some of the most common ways of accessing external storage are from the Context and Environment objects:

- `Context.getExternalFilesDir(mode)`: This method will return the path to the directory on the external storage dedicated to your application. Specifying different modes (pictures, movies, and so on) will create different subfolders, depending on how you want your files saved. This method *does not require permissions*.
- `Context.getExternalCacheDir()`: This will point toward an application's Cache directory on the external storage. The same considerations should be applied to this cache folder as to the internal storage option. This method *does not require permissions*.
- The Environment class has access to paths of some of the most common folders on a device. However, on newer devices, apps may not have access to those files and folders.



Avoid using hardcoded paths to files and folders. The Android operating system may shift the location of folders around, depending on the device or Android version.

## FileProvider

This represents a specialized implementation of ContentProviders that is useful in organizing the file and folder structure of your application. It allows you to specify an XML file, in which you define how your files should be split between internal and external storage if you choose to do so. It also gives you the ability to grant access to other apps to your files by hiding the path and generating a unique URI to identify and query your file.

FileProvider lets you pick between six different folders, where you can set up your folder hierarchies:

- `Context.getFilesDir()` (files-path)
- `Context.getCacheDir()` (cache-path)

- `Environment.getExternalStorageDirectory()` (external-path)
- `Context.getExternalFilesDir(null)` (external-files-path)
- `Context.getExternalCacheDir()` (external-cache-path)
- The first result of `Context.getExternalMediaDirs()` (external-media-path)

The main benefits of `FileProvider` are the abstractions it provides in organizing your files while leaving a developer to define the paths in an XML file, and more importantly, if you choose to use it to store files in external storage, you do not have to ask for permissions from the user.

Another benefit is the fact that it makes the sharing of internal files easier while giving a developer control of what files other apps can access without exposing their real location.

Let us understand better through the following example:

```
<paths
 xmlns:android="http://schemas.android.com/apk/res/android">
 <files-path
 name="my-visible-name"
 path="/my-folder-name" />
</paths>
```

The preceding example will make `FileProvider` use the internal files directory and create a folder named `my-folder-name`. When the path is converted to a URI, then the URI will use `my-visible-name`.

## The Storage Access Framework

The **Storage Access Framework (SAF)** is a file picker introduced in Android KitKat that apps can use for their users to pick files, with a view to them being processed or uploaded. You can use it in your app in the following scenarios:

- Your app requires a user to process a file saved on a device by another app (photos and videos)
- You want to save a file on a device and give a user the choice of where to save the file and the name of the file
- You want to offer the files your application uses to other apps for scenarios similar to the first scenario in this list

This is again useful because your app will avoid read and write permissions and still write and access external storage. The way this works is based on intents. You can register for an activity result for `GetDocument` or `CreateDocument`. Then, in the activity result callback, the system will give you a URI that grants you temporary permissions to that file, allowing you to read and write.

Another benefit of the SAF is the fact that files don't have to be on a device. Apps such as Google Drive expose their content in the SAF, and when a Google Drive file is selected, it will be downloaded to the device and the URI will be sent as a result.

Another important thing to mention is the SAF's support for virtual files, meaning that it will expose Google docs, which have their own format, but when those docs are downloaded through the SAF, their formats will be converted to a common format such as PDF.

## Asset files

**Asset files** are files you can package as part of your APK. If you've used an app that played certain videos or GIFs when the app was launched or as part of a tutorial, odds are that the videos were bundled with the APK. To add files to your assets, you need the `assets` folder inside your project. You can then group your files inside your assets using folders.

You can access these files at runtime through the `AssetManager` class, which itself can be accessed through the `Context` object. `AssetManager` offers you the ability to look up the files and read them, but it does not permit any write operations:

```
val assetManager = context.assets
val root = ""
val files = assetManager.list(root)
files?.forEach {
 val inputStream = assetManager.open(root + it)
}
```

The preceding example lists all files inside the root of the `assets` folder. The `open` function returns `InputStream`, which can be used to read the file information if necessary.

One common usage of the `assets` folder is for custom fonts. If your application uses custom fonts, then you can use the `assets` folder to store font files.



For the following exercise, you will need an emulator. You can do so by selecting **Tools | AVD Manager** in Android Studio. Then, you can create one with the **Create Virtual Device** option, selecting the type of emulator, clicking **Next**, and then selecting an x86 image. Any image larger than Lollipop should be acceptable for this exercise. Next, you can give your image a name and click **Finish**.

## Exercise 12.03 – copying files

Let's create an app that will keep a file named `my-app-file.txt` in the `assets` directory. The app will display two buttons called **FileProvider** and **SAF**. When the **FileProvider** button is clicked, the file will be saved on the external storage inside the app's external storage dedicated area (`Context.getExternalFilesDir(null)`). The **SAF** button will open the SAF and allow a user to indicate where the file should be saved.

The steps for completion are as follows:

1. Create a new Android Studio project with an empty activity.
2. Let's begin by adding the appropriate libraries to `gradle/libs.versions.toml`:

```
[versions]
...
viewModelCompose = "2.8.7"

[libraries]
...
androidx-viewmodel-compose = { group = "androidx.lifecycle", name =
 "lifecycle-viewmodel-compose", version.ref = "viewModelCompose" }
```

3. Add the `ViewModel` dependency to `app/build.gradle.kts`:

```
implementation(libs.androidx.viewmodel.compose)
```

4. Create the `my-app-file.txt` file in the `main/assets` folder. Feel free to fill it up with the text you want to be read. If the `main/assets` folder doesn't exist, then you can create it. To create the `assets` folder, you can right-click on the `main` folder, select **New**, then **Directory**, and name it `assets`.

This folder will now be recognized by the build system, and any file inside it will also be installed on the device along with the app. You may need to switch **Project View** from **Android** to **Project** to be able to view this file structure.

5. We can also define a class that will wrap `AssetManager` in the `main/java` folder in the root package and define a method to access this file:

```
class AssetFileManager(private val assetManager:
 AssetManager) {
 fun getAppFileInputStream() =
 assetManager.open("my-app-file.txt")
}
```

- Now, let's work on the FileProvider aspect. Create the `xml` folder in the `res` folder. Define `file_provider_paths.xml` inside the new folder. We will define `external-files-path`, name it `docs`, and place it in the `docs/` folder:

```
<?xml version="1.0" encoding="utf-8"?>
<paths>
 <external-files-path name="docs" path="docs/" />
</paths>
```

- Next, we need to add FileProvider to the `AndroidManifest.xml` file and link it with the new path we defined inside the `<application>` tag:

```
<provider
 android:name=
 "androidx.core.content.FileProvider"
 android:authorities=
 "com.packt.android.files"
 android:exported="false"
 android:grantUriPermissions="true">
 <meta-data
 android:name="android.support
 .FILE_PROVIDER_PATHS"
 android:resource="@xml/
 file_provider_paths" />
</provider>
```

The name will point to the FileProvider path that's part of the Android Support Library. The `authorities` field represents the domain your application has (usually the package name of the application).

The `exported` field indicates whether we wish to share our provider with other apps, and `grantUriPermissions` indicates whether we wish to grant other applications access to certain files through the URI. The metadata links the XML file we defined in *step 6* with FileProvider.

- Define the `ProviderFileManager` class in the `main/java` folder in the root package, which is responsible for accessing the `docs` folder and writing data into the file:

```
class ProviderFileManager(
 private val context: Context,
 private val fileToUriMapper: FileToUriMapper
```

```
) {

 private fun getDocsFolder(): File {
 val folder = File(
 context.getExternalFilesDir(null), "docs")
 if (!folder.exists()) {
 folder.mkdirs()
 }
 return folder
 }

 suspend fun writeStream(
 name: String, inputStream: InputStream
) {
 withContext(Dispatchers.IO) {
 val fileToSave = File(getDocsFolder(), name)
 val outputStream = context.contentResolver
 .openOutputStream(fileToUriMapper
 .getUriFromFile(
 context,
 fileToSave
), "rw"
)
 outputStream?.let {
 inputStream.copyTo(outputStream)
 }
 }
 }
}
```

Here `getDocsFolder` will return the path to the docs folder we defined in the XML. If the folder does not exist, then it will be created.

The `writeStream` method will extract the URI for the file we wish to save and using the Android `ContentResolver` class will give us access to the `OutputStream` class of the file we will be saving in. Note that `FileToUriMapper` doesn't exist yet. The code is moved into a separate class to make this class testable.



9. Create the `FileToUriMapper` class in the `main/java` folder in the root package:

```
class FileToUriMapper {
 fun getUriFromFile(context: Context, file: File):
 Uri {
 return FileProvider.getUriForFile(context,
 " com.packt.android.files", file)
 }
 }
}
```

The `getUriForFile` method is part of the `FileProvider` class, and its role is to convert the path of a file into a URI that can be used by `ContentProviders` and `ContentResolvers` to access data. Because the method is static, it prevents us from testing properly.

10. Make sure that the following strings are added to `strings.xml`:

```
<string name="file_provider">FileProvider</string>
<string name="saf">SAF</string>
```

11. Create the `FileViewModel` class, which will transfer and invoke the `writeStream` function from `ProviderFileManager` using `getMyAppFileInputStream` from `AssetFileManager` as an input:

```
class FileViewModel(
 private val assetFileManager: AssetFileManager,
 private val providerFileManager: ProviderFileManager
) : ViewModel() {

 fun copyUsingFileProvider() {
 viewModelScope.launch {
 providerFileManager.writeStream(
 "Copied.txt",
 assetFileManager
 .getMyAppFileInputStream()
)
 }
 }
}
```

12. Create the `FileScreen@Composable` function, which will hold the two buttons:

```
@Composable
fun FileScreen(
 onFileProviderClicked: () -> Unit,
 onSafClicked: () -> Unit,
 modifier: Modifier = Modifier
) {
 Column(modifier = modifier) {
 Button(onClick = onFileProviderClicked) {
 Text(text = stringResource(id =
 R.string.file_provider))
 }
 Button(onClick = onSafClicked) {
 Text(text = stringResource(id =
 R.string.saf))
 }
 }
}
```

13. Create the `File@Composable` function, which will link `FileViewModel` with `FileScreen`:

```
@Composable
fun File(
 fileViewModel: FileViewModel,
 modifier: Modifier
) {
 FileScreen(
 onFileProviderClicked = {
 fileViewModel.copyUsingFileProvider() },
 onSafClicked = { },
 modifier
)
}
```

14. Modify `MainActivity` to render the screen defined in the `File` function. `MainActivity` would also be responsible for initializing the previously defined classes:

```
class MainActivity : ComponentActivity() {
 override fun onCreate(savedInstanceState: Bundle?) {
```

```

 super.onCreate(savedInstanceState)
 enableEdgeToEdge()
 val assetFileManager = AssetFileManager/assets)
 val providerFileManager = ProviderFileManager(
 applicationContext, FileToUriMapper())
 setContent {
 Exercise1203Theme {
 Scaffold(
 modifier = Modifier.fillMaxSize()
) { innerPadding ->
 File(
 fileViewModel =
 viewModel<FileViewModel>(
 factory = object :
 ViewModelProvider
 .Factory
 {
 override fun <T : ViewModel>
 create(modelClass:
 Class<T>): T
 {
 return FileViewModel(
 assetFileManager,
 providerFileManager
) as T
 }
 }
),
 modifier = Modifier
 .padding(innerPadding)
)
 }
 }
 }
 }
}

```

For this example, we chose `MainActivity` to create our objects and inject data into the different classes we have. If we execute this code and click the **FileProvider** button, we don't see an output on the UI.

However, if we look at Android's **Device File Explorer**, we can locate where the file was saved. The path may be different on different devices and operating systems. The paths could be as follows:

- `mnt/sdcard/Android/data/<package_name>/files/docs`
- `sdcard/Android/data/<package_name>/files/docs`
- `storage/emulated/0/Android/data/<package_name>/files/docs`

The output will be as follows:

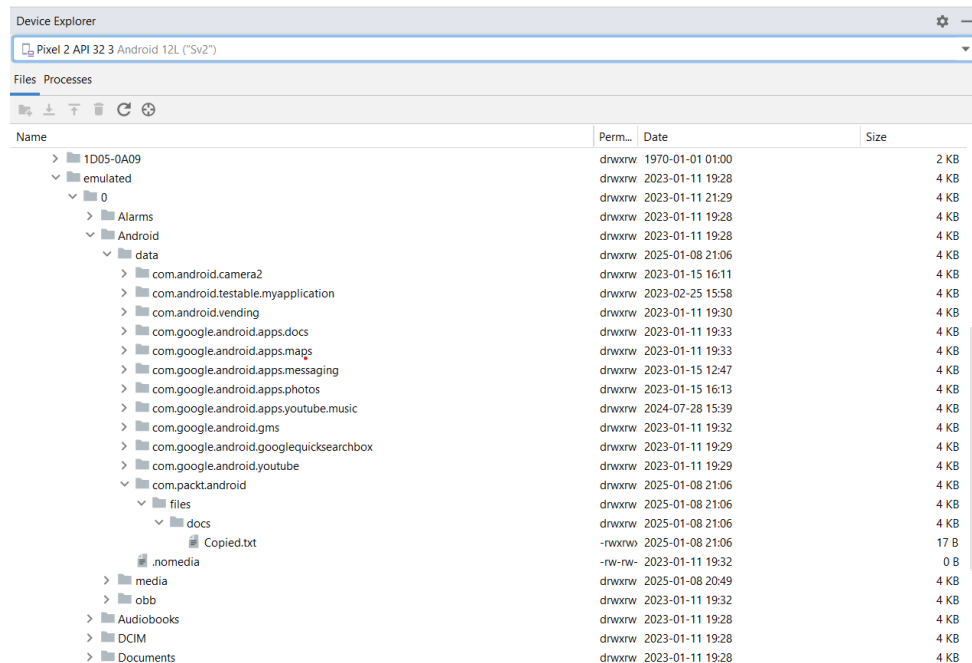


Figure 12.4 – Output of copy through FileProvider

- Let's add the logic for the **SAF** button. We will need to start an activity pointing toward SAF with the `CREATE_DOCUMENT` intent, in which we specify that we want to create a text file. To do that, we will need to modify the `File @Composable` function to start the `CREATE_DOCUMENT` intent:

```
@Composable
fun File(
 fileViewModel: FileViewModel,
```

```

 modifier: Modifier
) {
 val safFileName = "CopiedSAF.txt"
 val launcher =
 rememberLauncherForActivityResult(
 ActivityResultContracts
 .CreateDocument("text/plain")) {
 uri: Uri? ->
 uri?.let {
 fileViewModel.copyUsingSaf(it)
 }
 }
 FileScreen(
 onFileProviderClicked = {
 fileViewModel.copyUsingFileProvider() },
 onSafClicked = { launcher.launch(safFileName) },
 modifier
)
 }
}

```

What the preceding code will do is register for an Activity result when a user creates a new file. When the action completes, we will then invoke `FileViewModel` to copy the file.

16. Modify `FileViewModel` to add the `copyUsingSaf` function:

```

class FileViewModel(
 private val assetFileManager: AssetFileManager,
 private val providerFileManager: ProviderFileManager
) : ViewModel() {
 ...
 fun copyUsingSaf(uri: Uri) {
 viewModelScope.launch {
 providerFileManager.writeStreamFromUri(
 assetFileManager
)
 }
 }
}

```

```

 .getMyAppFileInputStream(),
 uri
)
}
}
}

```

17. Modify `ProviderFileManager` to add the `writeStreamFromUri` function, which will copy from the assets folder into the URI returned by SAF:

```

class ProviderFileManager(
 private val context: Context,
 private val fileToUriMapper: FileToUriMapper
) {
 ...
 suspend fun writeStreamFromUri(
 inputStream: InputStream, uri: Uri
) {
 withContext(Dispatchers.IO) {
 val outputStream = context.contentResolver
 .openOutputStream(uri, "rw")
 outputStream?.let {
 inputStream.copyTo(outputStream)
 }
 }
 }
}

```

If we run the preceding code and click on the **SAF** button, we will see the output presented in *Figure 12.5*:

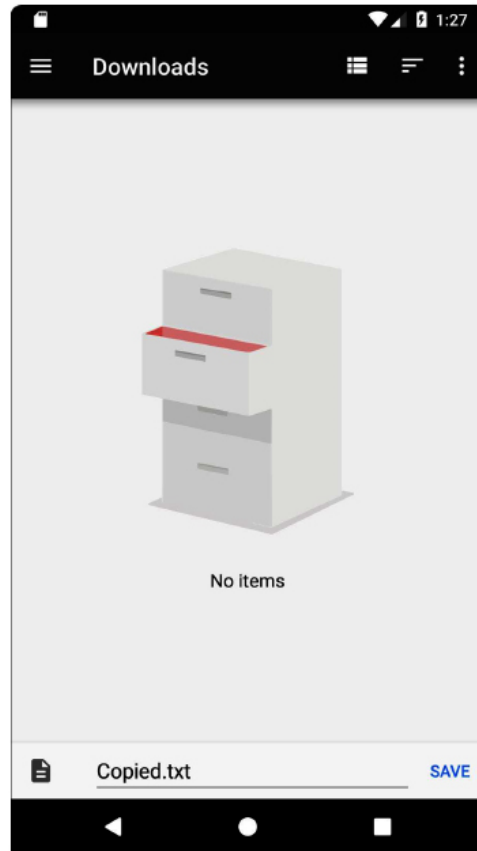


Figure 12.5 – Output of copy through the SAF

If you choose to save the file, the SAF will be closed and the callback from `rememberLauncherForActivityResult` will be invoked, which will trigger the file copy. Afterward, you can navigate the Android Device File Manager tool in Android Studio to see whether the file was saved properly.

In this exercise, we have implemented two approaches for copying a file. The first one was through `FileProvider` without any user prompt, and the second one was through SAF, which prompts the user to choose the location where a file should be saved. In the next section, we will look at scoped storage, which allows an app to prevent access to its files on external storage.

## Understanding scoped storage

Since Android 10 and with further updates in Android 11, the notion of scoped storage was introduced. The main idea behind this is to allow apps to gain more control of their files in external storage and prevent other apps from accessing these files.

The consequences of this mean that `READ_EXTERNAL_STORAGE` and `WRITE_EXTERNAL_STORAGE` will only apply to files a user interacts with (such as media files). This discourages apps from creating their own directories in external storage, instead sticking with the one already provided to them through `Context.getExternalFilesDir`.

`FileProvider` and the SAF are a good way of making your app comply with scoped storage practices, with one allowing the app to use `Context.getExternalFilesDir` and the other using the built-in File Explorer app, which will now avoid files from other applications in the `Android/data` and `Android/obb` folders in external storage. An important use of external storage is to store and access media files such as photos and videos, which is what we will analyze in the following section.

## Camera and media storage

Android offers a variety of ways to interact with media on its devices, from building your own camera application and controlling how users take photos and videos to using an existing camera application and instructing it on how to take photos and videos.

Android also comes with a `MediaStore` content provider, allowing applications to extract information about media files that are stored on a device and shared between applications.

This is useful in situations where you want a custom display for media files that exist on a device (such as a photo or music player application). Another use of `MediaStore` is in situations where you use the `MediaStore.ACTION_PICK` intent to select a photo from the device and want to extract the information about the selected media image (this is usually the case for older applications where the SAF cannot be used).

To use an existing camera application, you will need to use the `MediaStore.ACTION_IMAGE_CAPTURE` intent to start a camera application for a result and pass the URI of the image you wish to save. The user will then go to the camera activity and take the photo, and then you handle the result of the operation:

```
val imageCaptureLauncher = registerForActivityResult(
 ActivityResultContracts.TakePicture()
) {
 }
imageCaptureLauncher.launch(photoUri)
```



The `photoUri` parameter will represent the location where you want your photo to be saved. It should point to an empty file with a JPEG extension. You can build this file in two ways:

- Create a file on the external storage using the `File` object (this requires the `WRITE_EXTERNAL_STORAGE` permission) and then use the `Uri.fromFile()` method to convert it into URI (this is no longer applicable on Android 10 and above)
- Create a file in a `FileProvider` location using the `File` object, and then use the `FileProvider.getUriForFile()` method to obtain the URI and grant it permissions if necessary (the recommended approach for when your app targets Android 10 and Android 11)



The same mechanism can be applied to videos using `MediaStore.ACTION_VIDEO_CAPTURE`.

If your application relies heavily on camera features, then you can exclude the application from users whose devices don't have cameras by adding the `<uses-feature>` tag to the `AndroidManifest.xml` file. You can also specify the camera as non-required and query whether the camera is available using the `Context.hasSystemFeature(PackageManager.FEATURE_CAMERA_ANY)` method.

If you wish to have your file saved in `MediaStore`, there are multiple ways to achieve this:

- Send an `ACTION_MEDIA_SCANNER_SCAN_FILE` broadcast with the URI of your media:

```
val intent = Intent(
 Intent.ACTION_MEDIA_SCANNER_SCAN_FILE)
intent.data = photoUri
sendBroadcast(intent)
```

- Use the media scanner to scan files directly:

```
val paths = arrayOf("path1", "path2")
val mimeTypes = arrayOf("type1", "type2")
MediaScannerConnection.scanFile(
 context, paths, mimeTypes) { path, uri ->
}
```

- Insert the media into `ContentProvider` directly using `ContentResolver`:

```
val contentValues = ContentValues()
contentValues.put(MediaStore.Images
 .ImageColumns.TITLE, "my title")
```

```

 contentValues.put(MediaStore.Images
 .ImageColumns .DATE_ADDED, timeInMillis)
 contentValues.put(MediaStore.Images
 .ImageColumns .MIME_TYPE, "image/*")
 contentValues.put(MediaStore.Images
 .ImageColumns .DATA, "my-path")
 val newUri =
 contentResolver.insert(MediaStore.Video
 .Media.EXTERNAL_CONTENT_URI,
 contentValues)
 newUri?.let {
 val outputStream = contentResolver
 .openOutputStream(newUri)
 // Copy content in outputstream
 }
 }

```



The MediaScanner functionality no longer adds files from Context. getExternalFilesDir in Android 10 and above. Apps should rely on the insert method instead if they choose to share their media files with other apps.

## Exercise 12.04 – taking photos

We're going to build an application that has two buttons; the first button will open a camera app to take a photo, and the second button will open the camera app to record a video. We will use FileProvider to save the photos to external storage (external-path) in two folders, pictures and movies.

The photos will be saved using `img_{timestamp}.jpg`, and the videos will be saved using `video_{timestamp}.mp4`. After a photo and video have been saved, you will copy the file from FileProvider into MediaStore so that they will be visible for other apps:

1. Create a new Android Studio project with an empty activity.
2. Let's begin by adding the appropriate libraries to `gradle/libs.versions.toml`:

```

[versions]
...
viewModelCompose = "2.8.7"

[libraries]

```

```
...
 androidx-viewmodel-compose = { group = "androidx.lifecycle", name =
 "lifecycle-viewmodel-compose", version.ref = "viewModelCompose"
 }
```

3. Add the ViewModel dependency to app/build.gradle.kts:

```
implementation(libs.androidx.viewmodel.compose)
```

4. We will need to request the WRITE\_EXTERNAL\_STORAGE permission for devices that predate Android 10, which means we need the following in AndroidManifest.xml outside of the <application tag:

```
<uses-permission
 android:name=
 "android.permission.WRITE_EXTERNAL_STORAGE"
 android:maxSdkVersion="28" />
```

5. Let's define a FileHelper class, which will contain methods that are harder to test in the test folder in the root package:

```
class FileHelper(private val context: Context) {
 fun getUriFromFile(file: File): Uri {
 return FileProvider.getUriForFile(context,
 "com.android.testable.camera", file)
 }
 fun getPicturesFolder(): String =
 Environment.DIRECTORY_PICTURES
 fun getVideosFolder(): String =
 Environment.DIRECTORY_MOVIES
}
```

6. Let's define our FileProvider paths in res/xml/file\_provider\_paths.xml. Make sure to include the appropriate package name for your application in FileProvider:

```
<?xml version="1.0" encoding="utf-8"?>
<paths>
 <external-path
 name="photos"
 path="Android/data/com.android.testable
```

```
 .myapplication/files/Pictures" />
 <external-path
 name="videos"
 path="Android/data/com.android.testable
 .myapplication/files/Movies" />
</paths>
```

7. Let's add the file provider paths to the `AndroidManifest.xml` file inside the `<application` tag:

```
<provider
 android:name=
 "androidx.core.content.FileProvider"
 android:authorities=
 "com.packt.android.camera"
 android:exported="false"
 android:grantUriPermissions="true">
 <meta-data
 android:name=
 "android.support.FILE_PROVIDER_PATHS"
 android:resource="@xml/file_provider_paths" />
</provider>
```

8. Next, let's define a model that will hold both `Uri` and the associated path for a file in the `main/java` folder in the root package:

```
data class FileInfo(
 val uri: Uri,
 val file: File,
 val name: String,
 val relativePath:String,
 val mimeType:String
)
```

9. Let's create a `ContentHelper` class in the `main/java` folder in the root package, which will provide us with the data required for `ContentResolver`. We will define two methods for accessing the photo and video content URI and two methods that will create `ContentValues`

We do this because of the static methods required to obtain URIs and create ContentValues, which makes this functionality hard to test. The following code is truncated for space:

```
class MediaContentHelper {

 fun getImageContentUri(): Uri =
 if (android.os.Build.VERSION.SDK_INT >=
 android.os.Build.VERSION_CODES.Q) {
 MediaStore.Images.Media.getContentUri(
 MediaStore.VOLUME_EXTERNAL_PRIMARY)
 } else {
 MediaStore.Images.Media.EXTERNAL_CONTENT_URI
 }

 fun generateImageContentValues(fileInfo: FileInfo) =
 ContentValues().apply {
 this.put(MediaStore.Images.Media.DISPLAY_NAME, fileInfo.name)
 if (android.os.Build.VERSION.SDK_INT >=
 android.os.Build.VERSION_CODES.Q)
 {
 this.put(MediaStore.Images.Media.RELATIVE_PATH,
 fileInfo.relativePath)
 }
 this.put(MediaStore.Images.Media.MIME_TYPE,
 fileInfo.mimeType)
 }

 fun getVideoContentUri(): Uri =
 if (android.os.Build.VERSION.SDK_INT >=
 android.os.Build.VERSION_CODES.Q)
 {
 MediaStore.Video.Media.getContentUri(
 MediaStore.VOLUME_EXTERNAL_PRIMARY)
 } else {
 MediaStore.Video.Media
 .EXTERNAL_CONTENT_URI
 }
}
```

```

fun generateVideoContentValues(fileInfo: FileInfo) =
 ContentValues().apply {
 this.put(MediaStore.Video.Media.DISPLAY_NAME,
 fileInfo.name)
 if (android.os.Build.VERSION.SDK_INT >=
 android.os.Build.VERSION_CODES.Q)
 {
 this.put(MediaStore.Video.Media.RELATIVE_PATH,
 fileInfo.relativePath)
 }
 this.put(MediaStore.Video.Media.MIME_TYPE,
 fileInfo.mimeType)
 }
}

```

10. Now, let's create the `ProviderFileManager` class in the `main/java` folder in the root package, where we will define methods to generate files for photos and videos that will then be used by the camera and the methods that will be saved to the media store. We will begin by creating two methods for generating the data for a photo and a video:

```

class ProviderFileManager(
 private val context: Context,
 private val fileHelper: FileHelper,
 private val contentResolver: ContentResolver,
 private val mediaContentHelper: MediaContentHelper
) {

 fun generatePhotoUri(time: Long): FileInfo {
 val name = "img_${time}.jpg"
 val file = File(
 context.getExternalFilesDir(
 fileHelper.getPicturesFolder()),
 name
)
 return FileInfo(
 fileHelper.getUriFromFile(file),
 file,
 name,

```

```

 fileHelper.getPicturesFolder(),
 "image/jpeg"
)
}

fun generateVideoUri(time: Long): FileInfo {
 val name = "video_{$time}.mp4"
 val file = File(
 context.getExternalFilesDir(
 fileHelper.getVideosFolder()
),
 name
)
 return FileInfo(
 fileHelper.getUriFromFile(file),
 file,
 name,
 fileHelper.getVideosFolder(),
 "video/mp4"
)
}
}

```

11. Let's add a private method that will save ContentValues into ContentResolver to the same file:

```

class ProviderFileManager(...) {
 ...
 private suspend fun insertToStore(
 fileInfo: FileInfo,
 contentUri: Uri,
 contentValues: ContentValues
) {
 withContext(Dispatchers.IO) {
 val insertedUri = contentResolver
 .insert(contentUri,

```

```

 contentValues)
 insertedUri?.let {
 val inputStream = contentResolver
 .openInputStream(fileInfo.uri)
 val outputStream = contentResolver
 .openOutputStream(insertedUri)
 outputStream?.let {
 inputStream?.copyTo(outputStream)
 }
 }
}
}
}
}

```

In the same file, add two methods that will save an image into MediaStore and a video into MediaStore:

```

class ProviderFileManager(...) {
 ...
 suspend fun insertImageToStore(fileInfo: FileInfo?)
 {
 fileInfo?.let {
 insertToStore(
 fileInfo,
 mediaContentHelper.getImageContentUri(),
 mediaContentHelper.generateImageContentValues(it)
)
 }
 }

 suspend fun insertVideoToStore(fileInfo: FileInfo?)
 {
 fileInfo?.let {
 insertToStore(
 fileInfo,
 mediaContentHelper.getVideoContentUri(),
 mediaContentHelper.generateVideoContentValues(it)
)
 }
 }
}

```



```

 }
 }
}

```

Note how we defined the root folders as `context.getExternalFilesDir(Environment.DIRECTORY_PICTURES)` and `context.getExternalFilesDir(Environment.DIRECTORY_MOVIES)`. This connects to `file_provider_paths.xml` and it will create a set of folders called `Movies` and `Pictures` in the application's dedicated folder in external storage. The `insertToStore` method is where the files will then be copied to `MediaStore`.

12. First, we will create an entry in that store that will give us a URI for that entry. Next, we copy the contents of the files from the URI generated by `FileProvider` into `OutputStream`, pointing to the `MediaStore` entry.
13. Create `FileViewModel`, which will be responsible for interacting with `ProviderFileManager`:

```

class FileViewModel(private val providerFileManager:
 ProviderFileManager) : ViewModel() {

 private var photoFileInfo: FileInfo? = null
 private var videoFileInfo: FileInfo? = null

 fun generatePhotoUri(time: Long): Uri {
 val info = providerFileManager.generatePhotoUri(time)
 photoFileInfo = info
 return info.uri
 }

 fun generateVideoUri(time: Long): Uri {
 val info = providerFileManager.generateVideoUri(time)
 videoFileInfo = info
 return info.uri
 }

 fun insertImageToStore() {
 viewModelScope.launch {
 providerFileManager.insertImageToStore(photoFileInfo)
 }
 }
}

```

```
fun insertVideoToStore() {
 viewModelScope.launch {
 providerFileManager.insertVideoToStore(videoFileInfo)
 }
}
```

14. Add the following strings to strings.xml:

```
<string name="photo">Photo</string>
<string name="video">Video</string>
```

15. Create the MediaScreen function, which will contain the two buttons for photos and videos:

```
@Composable
fun MediaScreen(
 onPhotoClicked: () -> Unit,
 onVideoClicked: () -> Unit,
 modifier: Modifier = Modifier
) {
 Column(modifier = modifier) {
 Button(onClick = onPhotoClicked) {
 Text(text = stringResource(id = R.string.photo))
 }
 Button(onClick = onVideoClicked) {
 Text(text = stringResource(id = R.string.video))
 }
 }
}
```

16. Create the Media function, which will be responsible for launching the camera app, capturing a photo or video, making sure that all the permissions are set, and then saving the files. We will first need to create two launches for capturing a photo and a video and then define a variable for what we want to capture:

```
@Composable
fun Media(
 fileViewModel: FileViewModel,
 modifier: Modifier = Modifier
```

```

) {
 val imageLauncher =
 rememberLauncherForActivityResult(
 ActivityResultContracts.TakePicture()
) {
 fileViewModel.insertImageToStore()
 }
 val videoLauncher =
 rememberLauncherForActivityResult(
 ActivityResultContracts.CaptureVideo()
) {
 fileViewModel.insertVideoToStore()
 }
 var isPhoto = remember {
 true
 }
}

```

17. In the same function, we will now define two lambdas. One will launch the camera, and the other to request external storage permissions:

```

@Composable
fun Media(
 fileViewModel: FileViewModel,
 modifier: Modifier = Modifier
) {
 ...
 val launchCamera = {
 if (isPhoto) {
 imageLauncher.launch(
 fileViewModel.generatePhotoUri(
 System.currentTimeMillis()
)
)
 } else {
 videoLauncher.launch(
 fileViewModel.generateVideoUri(
 System.currentTimeMillis()
)
)
 }
 }
}

```

```

)
 }
}
val externalStoragePermissionLauncher =
 rememberLauncherForActivityResult(
 ActivityResultContracts.RequestPermission()
)
{ isGranted ->
 if (isGranted) {
 launchCamera()
 }
}
}

```

18. Next, we define a lambda that will check whether the external storage permissions were granted. If not, it will call the permission launcher, and if they are, it will launch the camera app:

```

@Composable
fun Media(
 fileViewModel: FileViewModel,
 modifier: Modifier = Modifier
) {
 ...
 val context = LocalContext.current
 val checkStoragePermissions = {
 if (android.os.Build.VERSION.SDK_INT <
 android.os.Build.VERSION_CODES.Q)
 {
 when (ContextCompat.checkSelfPermission(
 context,
 android.Manifest.permission
 .WRITE_EXTERNAL_STORAGE
)) {
 PackageManager.PERMISSION_GRANTED -> {
 launchCamera()
 }
 }
 }
 }
}

```

```

 else -> {
 externalStoragePermissionLauncher
 .launch(android.Manifest
 .permission
 .WRITE_EXTERNAL_STORAGE)
 }
 }
} else {
 launchCamera()
}
}
}

```

19. In the same function, connect `MediaScreen` and connect the button clicks to the `checkStoragePermissions` lambda defined in the previous step:

```

@Composable
fun Media(
 fileViewModel: FileViewModel,
 modifier: Modifier = Modifier
) {
 ...
 MediaScreen(
 onPhotoClicked = {
 isPhoto = true
 checkStoragePermissions()
 },
 onVideoClicked = {
 isPhoto = false
 checkStoragePermissions()
 },
 modifier = modifier
)
}

```

20. Finally, modify `MainActivity` in order to invoke the `File` function defined previously:

```

class MainActivity : ComponentActivity() {
 override fun onCreate(savedInstanceState: Bundle?) {

```

```

super.onCreate(savedInstanceState)
enableEdgeToEdge()
setContent {
 Exercise1204Theme {
 Scaffold(
 modifier = Modifier.fillMaxSize()
) { innerPadding ->
 Media(fileViewModel =
 viewModel<FileViewModel>(
 factory = object :
 ViewModelProvider
 .Factory
 {
 override fun <T : ViewModel>
 create(modelClass:
 Class<T>): T
 {
 return FileViewModel(
 ProviderFileManager(
 applicationContext,
 FileHelper(
 applicationContext
),
 contentResolver,
 MediaContentHelper()
)
) as T
 }
 },
 modifier = Modifier
 .padding(innerPadding)
)
 }
}
}
}
}
}

```

If we execute the preceding code, we will see the following:

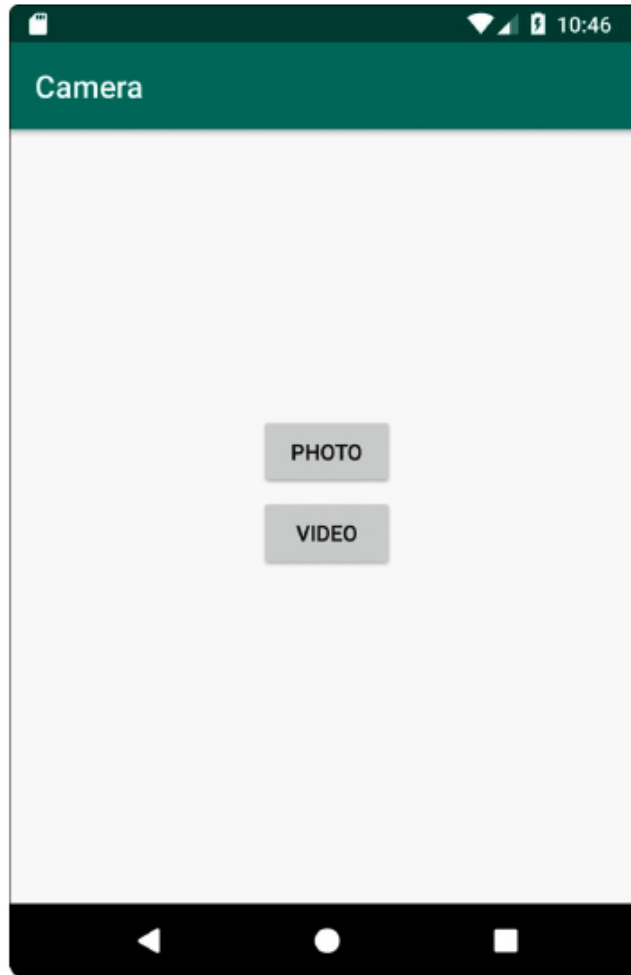
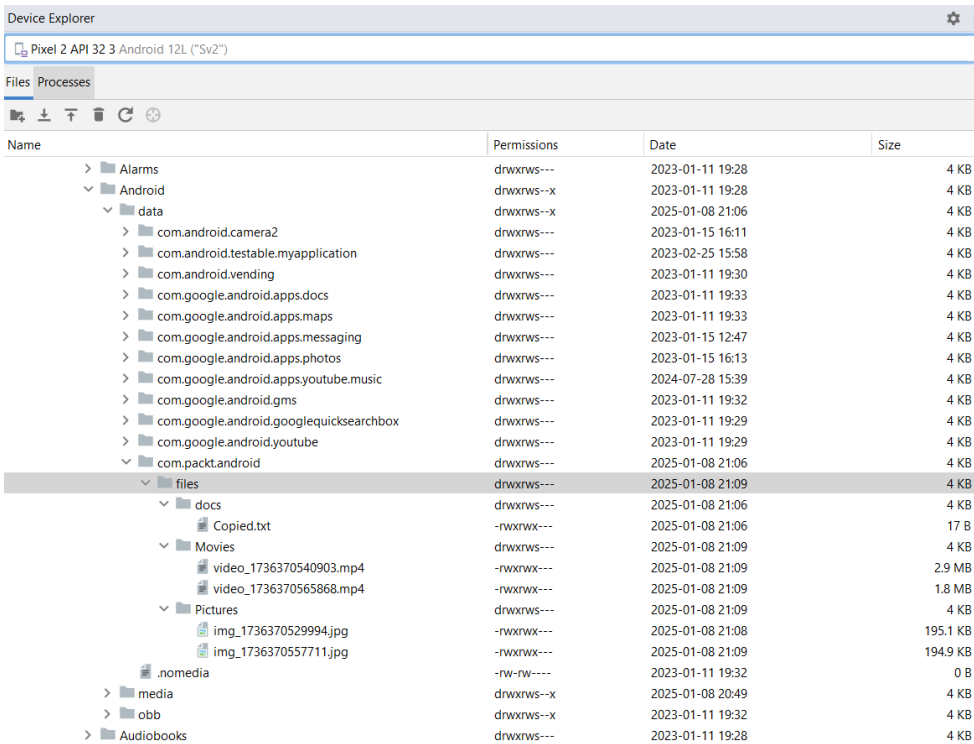


*Figure 12.6 – Output of Exercise 12.04*

21. By clicking on either of the buttons, you will be redirected to the camera application, where you can take a photo or a video if you are running the example on Android 10 and above. If you're running on lower Android versions, then the permissions will be asked for first.

Once you have taken your photo and confirmed it, you will be taken back to the application. The photo will be saved in the location you defined in `FileProvider`:



| Name                                      | Permissions | Date             | Size     |
|-------------------------------------------|-------------|------------------|----------|
| > Alarms                                  | drwxrws---  | 2023-01-11 19:28 | 4 KB     |
| > Android                                 | drwxrws--x  | 2023-01-11 19:28 | 4 KB     |
| > data                                    | drwxrws--x  | 2025-01-08 21:06 | 4 KB     |
| > com.android.camera2                     | drwxrws---  | 2023-01-15 16:11 | 4 KB     |
| > com.android.testable.myapplication      | drwxrws---  | 2023-02-25 15:58 | 4 KB     |
| > com.android.vending                     | drwxrws---  | 2023-01-11 19:30 | 4 KB     |
| > com.google.android.apps.docs            | drwxrws---  | 2023-01-11 19:33 | 4 KB     |
| > com.google.android.apps.maps            | drwxrws---  | 2023-01-11 19:33 | 4 KB     |
| > com.google.android.apps.messaging       | drwxrws---  | 2023-01-15 12:47 | 4 KB     |
| > com.google.android.apps.photos          | drwxrws---  | 2023-01-15 16:13 | 4 KB     |
| > com.google.android.apps.youtube.music   | drwxrws---  | 2024-07-28 15:39 | 4 KB     |
| > com.google.android.gms                  | drwxrws---  | 2023-01-11 19:32 | 4 KB     |
| > com.google.android.googlequicksearchbox | drwxrws---  | 2023-01-11 19:29 | 4 KB     |
| > com.google.android.youtube              | drwxrws---  | 2023-01-11 19:29 | 4 KB     |
| > com.pactk.android                       | drwxrws---  | 2025-01-08 21:06 | 4 KB     |
| > files                                   | drwxrws---  | 2025-01-08 21:09 | 4 KB     |
| > docs                                    | drwxrws---  | 2025-01-08 21:06 | 4 KB     |
| Copied.txt                                | -rwxrwx---  | 2025-01-08 21:06 | 17 B     |
| > Movies                                  | drwxrws---  | 2025-01-08 21:09 | 4 KB     |
| video_1736370540903.mp4                   | -rwxrwx---  | 2025-01-08 21:09 | 2.9 MB   |
| video_1736370565868.mp4                   | -rwxrwx---  | 2025-01-08 21:09 | 1.8 MB   |
| > Pictures                                | drwxrws---  | 2025-01-08 21:09 | 4 KB     |
| img_1736370529994.jpg                     | -rwxrwx---  | 2025-01-08 21:08 | 195.1 KB |
| img_1736370557711.jpg                     | -rwxrwx---  | 2025-01-08 21:09 | 194.9 KB |
| .nomedia                                  | -rw-rw----  | 2023-01-11 19:32 | 0 B      |
| > media                                   | drwxrws--x  | 2025-01-08 20:49 | 4 KB     |
| > obb                                     | drwxrws--x  | 2023-01-11 19:32 | 4 KB     |
| > Audiobooks                              | drwxrws---  | 2023-01-11 19:28 | 4 KB     |

Figure 12.7 – The location of the captured files through the camera app

 **Quick tip:** Need to see a high-resolution version of this image? Open this book in the next-gen Packt Reader or view it in the PDF/ePub copy.

 The next-gen Packt Reader and a free PDF/ePub copy of this book are included with your purchase. Scan the QR code OR visit [packtpub.com/unlock](https://packtpub.com/unlock), then use the search bar to find this book by name. Double-check the edition shown to make sure you get the right one.





In the preceding screenshot, you can see where the files are located with the help of Android Studio’s **Device File Explorer**. If you open any file-exploring app, such as the **Files**, **Gallery**, or **Google Photos** app, you will be able to see the videos and pictures taken.

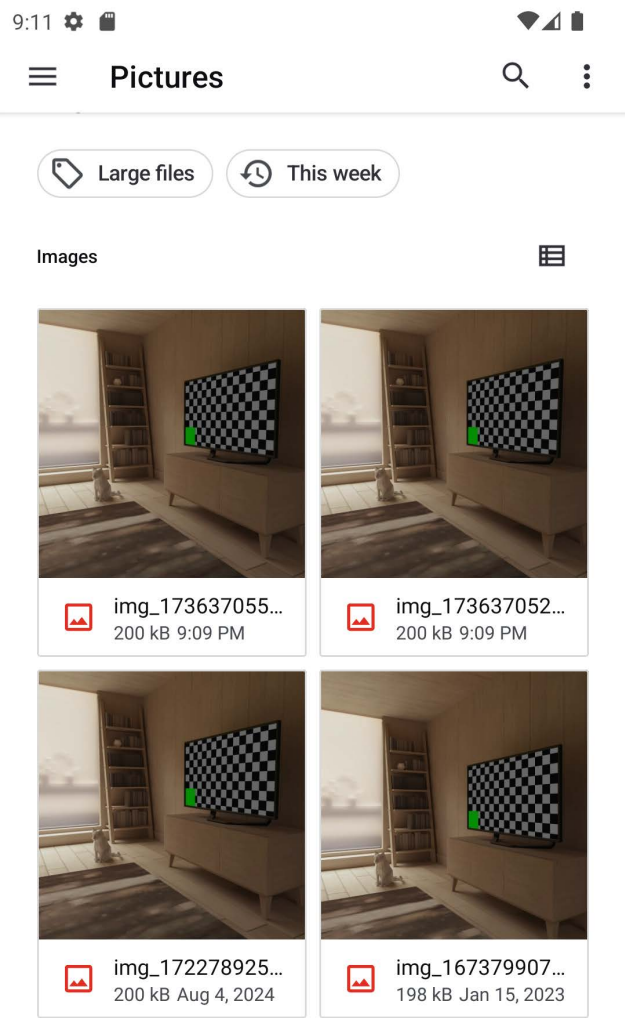


Figure 12.8 – The files from the app present in the File Explorer app

In this exercise, we looked at interactions with **MediaStore** and how we can use the camera app to generate photos and videos for our apps. In the following section, we will implement an application that combines multiple elements that were studied in this chapter.

## Activity 12.01 – managing multiple persistence options

You are tasked with building an application that will target Android versions above API 21 and display a list of URLs for dog photos. The URL you will connect to is `https://dog.ceo/api/breed/hound/images/random/{number}`, where `number` will be controlled through `TextInput` shown above the list where the user can change the number of results when a button is clicked. When a user clicks on a URL, an image will be downloaded locally in the application's external cache path. While the image is being downloaded, the user will see an indeterminate progress bar. The list of URLs will be persisted locally using `Room`.

The technologies that we will use are as follows:

- `Retrofit` to retrieve the list of URLs and download files
- `Room` to persist the list of URLs
- `SharedPreferences` to store the number of URLs to retrieve
- `FileProvider` to store the files in the cache
- `Repository` to combine all the data sources
- `Coroutines` and `ViewModel` to handle the logic from the user

The response JSON will look similar to this:

```
{
 "message": [
 "https://images.dog.ceo/breeds/hound-afghan/
 n02088094_4837.jpg",
 "https://images.dog.ceo/breeds/hound-basset/
 n02088238_13908.jpg",
 "https://images.dog.ceo/breeds/hound-ibizan/
 n02091244_3939.jpg"
],
 "status": "success"
}
```

Perform the following steps to complete this activity:

1. Create an `api` package that will contain the network-related classes.
2. Create a data class that will model the response JSON.
3. Create a `Retrofit Service` class that will contain two methods. The first method will represent the API call to return a list of breeds, and the second method will represent the API call to download the file.

4. Create a storage package, and inside it, create a room package.
5. Create the Dog entity, which will contain an autogenerated ID and a URL.
6. Create the DogDao class, which will contain methods to insert a list of dogs, delete all dogs, and query all dogs. The delete method is required because the API model does not have any unique identifiers.
7. Inside the storage package, create a preference package.
8. Inside the preference package, create a wrapper class around DataStore that will return the number of URLs we need to use and set the number. The default will be 10.
9. In res/xml, define your folder structure for FileProvider. The files should be saved in the root folder of the external-cache-path tag.
10. Inside the storage package, create a filesystem package.
11. Inside the filesystem package, define a class that will be responsible for writing InputStream into a file in FileProvider, using Context.externalCacheDir.
12. Create a repository package.
13. Inside the repository package, create a sealed class that will hold the result of an API call. The subclasses of the sealed class will be Success, Error, and Loading.
14. Define a Repository interface that will contain three methods: one to load the list of URLs, another to download a file, and one to update the list of results.
15. Define a DogUi model class that will be used in the UI layer of your application and that will be created in your repository.
16. Define a mapper class that will convert your API models into entities and entities into UI models.
17. Define an implementation for Repository that will implement the preceding three methods. The repository will be responsible for saving the list of dogs pulled from the internet into Room and then converting it into DogUi, which will then be consumed by ViewModel. It will also be responsible for invoking the download and saving it on the filesystem, as well as the number of results.
18. Define the ViewModel class used by your UI, which will have a reference to Repository, and call it to load the URL list, download the images, and update the results.
19. Define your UI, which will have a state for Loading, one for Success, and one for Error when loading the list of Dog objects.
20. Connect your UI to ViewModel and load the list of results. When the button is clicked, the list of dogs will be reloaded with the new number of results. When a button is clicked, then a download will be triggered and saved in FileProvider.



The solution to this activity can be found at <https://packt.link/5j4Aj>.

## Summary

In this chapter, we analyzed alternatives to Room when it comes to persisting data. We looked first at `SharedPreferences` and how it constitutes a handy solution for data persistence when it's in a key-value format and the amount of data is small. We also looked at `DataStore` and how we can use it like `SharedPreferences` but with built-in observability, which notifies us when values are changed.

Next, we looked over something that is continuously changing when it comes to the Android framework – the evolution of abstractions regarding a filesystem. We started with an overview of the types of storage that Android has and then took a more in-depth look at two of the abstractions – `FileProvider`, which your app can use to store files on a device and share them with others if necessary, and the `SAF`, which can be used to save files on the device in a location selected by a user.

We also used the benefits of `FileProvider` to generate URIs for files to use camera applications to take photos and record videos, saving them in the application's files while also adding them to `MediaStore`.

The activity performed in this chapter combined multiple approaches to persisting and managing data to showcase where each approach is best suited. Note that for the activity and exercises in this chapter and *Chapter 11, Android Architecture Components*, we had to keep using the `MainActivity` class to instantiate the data sources.

In the next chapter, you will learn how to overcome this through dependency injection and see how it can benefit Android applications.

### Unlock this book's exclusive benefits now

Scan this QR code or go to [packtpub.com/unlock](https://packtpub.com/unlock), then search this book by name.

*Note: Keep your purchase invoice ready before you start.*

UNLOCK NOW





# 13

## Dependency Injection with Dagger, Hilt, and Koin

This chapter covers the concept of **Dependency Injection (DI)** and the benefits it provides to an Android application. In *Chapter 12, Persisting Data*, we dealt with various data sources, such as the internet, local databases through Room, files, and DataStore. All these components would be initialized in the `Application` class and then injected into repositories and `ViewModel` objects.

This solution, however, is not a scalable one as our code base will grow. What we want is to separate the creation of an object from its usage. In other words, when we need a dependency in our class, we shouldn't be concerned with the creation of the dependency. This makes our code testable because it allows the tests to insert stable components in places where we would have unstable ones, which would cause testing failure.

In this chapter, we will analyze multiple ways to inject dependencies in Android. **Manual DI** is a technique in which developers handle DI manually by creating container classes. In this chapter, we will examine how we can do this in Android. By studying how we manually manage dependencies, we will get some insight into how other DI frameworks operate and get a basis for how we can integrate these frameworks.

**Dagger** is a DI framework developed for Java. It allows you to group your dependencies into different modules. You can also define components, where the modules are added to create the dependency graph, and which Dagger automatically implements to perform the injection. It relies on annotation processors to generate the necessary code to perform the injection. A specialized implementation of Dagger called **Hilt** is useful for Android applications because it removes a lot of boilerplate code and simplifies the process.

**Koin** is a lightweight DI library developed for Kotlin. It doesn't rely on annotation processors; it relies on Kotlin's mechanisms to perform the injection. Here, we can also split dependencies into modules.

By the end of the chapter, you should be able to use DI in an Android application either manually or with the help of various libraries.

We will cover the following topics in this chapter:

- Handling manual DI
- Using Dagger 2
- Switching to Hilt
- Using Koin

## Technical requirements

The complete code for all the exercises and the activity in this chapter is available on GitHub at <https://packt.link/qvkCC>.

## Handling manual DI

To understand how DI works, we can first analyze how we can manually inject dependencies into different objects across an Android application. When dealing with DI, there are four roles involved:

- **Service:** This is a class that contains useful functionality.
- **Client:** This is any class that depends on a service. In other words, it is any class that uses a service to perform a certain task.

- **Interface:** This represents the service abstraction. Ideally, clients should not know about the service directly; instead, it should be indirectly represented by the interface.
- **Injector:** This connects the clients and services. It can also be called a container, provider, or factory.

In the case of an Android application, the client-service roles shift from class to class, meaning that a class can represent a service for another class and then become a client when it requires dependencies to be injected.

To implement manual DI in an Android project, you can create containers that will take the roles of injectors. You can also create multiple containers representing different scopes required across the application. Here, you can define dependencies that will only be required if a particular screen is displayed, and when the screen is destroyed, the dependencies will also be garbage collected.

A sample of a container that will hold instances as long as an application lives is shown here. All the instances kept inside the container will represent services that will be needed in other classes:

```
class AppContainer(applicationContext:Context) {
 val myRepository: MyRepository
 init {
 val retrofit = Retrofit.Builder().baseUrl(
 "https://google.com/").build()
 val myService= retrofit.create<MyService>(MyService::class.java)
 val database =
 Room.databaseBuilder(applicationContext,
 MyDatabase::class.java, "db").build()
 myRepository = MyRepositoryImpl(myService,
 database.myDao())
 }
}
```



💡 **Quick tip:** Enhance your coding experience with the **AI Code Explainer** and **Quick Copy** features. Open this book in the next-gen Packt Reader. Click the **Copy** button (1) to quickly copy code into your coding environment, or click the **Explain** button (2) to get the AI assistant to explain a block of code to you.

```
function calculate(a, b) {
 return {sum: a + b};
};
```

Copy

1

Explain

2



🔒 The next-gen Packt Reader is included for free with the purchase of this book. Scan the QR code OR go to [packtpub.com/unlock](https://packtpub.com/unlock), then use the search bar to find this book by name. Double-check the edition shown to make sure you get the right one.



An Application class using that container looks something like the following:

```
class MyApplication : Application() {
 lateinit var appContainer: AppContainer
 override fun onCreate() {
 super.onCreate()
 appContainer = AppContainer(this)
 }
}
```

As you can see in the preceding example, the responsibility for creating the dependencies shifted from the Application class to the Container class. Activities across the code base can still access the dependencies using the following approach:

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

}
```

```

 val myRepository = (application as
 MyApplication).appContainer. myRepository
 ...
 }

```

Modules with a limited scope could be used for something such as creating the `ViewModel` factories, which, in turn, are used by the framework to create `ViewModel`:

```

class MyContainer(private val myRepository: MyRepository) {
 fun geMyViewModelFactory(): ViewModelProvider.Factory {
 return object : ViewModelProvider.Factory {
 override fun <T : ViewModel?>
 create(modelClass: Class<T>): T {
 return MyViewModel(myRepository) as T
 }
 }
 }
}

```

In the preceding snippet, we defined the `MyContainer` class, which is an injector and a client at the same time. It's a client because it has a dependency on `MyRepository`, and it is an injector because it's responsible for managing the `ViewModelProvider.Factory` instance. An activity or fragment can use this container to initialize `ViewModel`:

```

class MyActivity : AppCompatActivity() {
 private lateinit var myViewModel: MyViewModel
 private lateinit var myContainer: MyContainer
 override fun onCreate(savedInstanceState: Bundle?) {
 super.onCreate(savedInstanceState)

 val myRepository = (application as
 MyApplication).appContainer. myRepository
 myContainer = MyContainer (myRepository)
 myViewModel = ViewModelProvider(this,
 myContainer.geMyViewModelFactory())
 .get(MyViewModel::class.java)
 }
}

```

Again, we can see here that the responsibility of creating the Factory class was shifted from the Activity class to the Container class. MyContainer could be expanded to provide instances required by MyActivity in situations where the life cycle of those instances should be the same as the activity, or the constructor could be expanded to provide instances with a different life cycle.

Now, let's apply some of these examples to an exercise.

## Exercise 13.01 – manual injection

In this exercise, we will write an Android application that will apply the concept of manual DI. The application will have a repository, which will generate a random number, and a ViewModel object with a Flow object responsible for retrieving the number generated by the repository and publishing it in the Flow object. To do so, we will need to create two containers that will manage the repository and the ViewModel factory. The app will display the randomly generated number each time a button is clicked:

1. Create a new Android Studio project with an empty activity.
2. Let's start by adding the right library in `gradle/libs.versions.toml`:

```
[versions]
...
viewModelCompose = "2.6.1"
[libraries]
...
androidx-viewmodel-compose = { group = "androidx.lifecycle", name =
 "lifecycle-viewmodel-compose", version.ref = "viewModelCompose"
}
```

3. Now, let's add the dependencies to `app/build.gradle.kts`:

```
dependencies {
 ...
 implementation(libs.androidx.viewmodel.compose)
 ...
}
```

4. Next, let's write a `NumberRepository` interface in the `main/java` folder in the root package, which will contain a method to retrieve an integer:

```
interface NumberRepository {
 fun generateNextNumber(): Int
}
```

- Now, we will provide the implementation for this in the `main/java` folder in the root package. We can use the `java.util.Random` class to generate a random number:

```
class NumberRepositoryImpl(
 private val random: Random): NumberRepository {
 override fun generateNextNumber(): Int {
 return random.nextInt()
 }
}
```

- We will now move on to the `MainViewModel` class, which will contain a `StateFlow` object containing each generated number from the repository:

```
class MainViewModel(
 private val numberRepository: NumberRepository
) : ViewModel() {

 private val _state = MutableStateFlow<Int>(0)
 val state: StateFlow<Int> = _state

 fun generateNextNumber() {
 viewModelScope.launch {
 _state.emit(
 numberRepository.generateNextNumber()
)
 }
 }
}
```

- Make sure to add the string for the button to the `res/values/strings.xml` file:

```
<string name="randomize">Randomize</string>
```

- Create the `@Composable` function to render the screen with `Text` and `Button`:

```
@Composable
fun MainScreen(
 number: Int,
 onButtonClick: () -> Unit,
 modifier: Modifier = Modifier
) {
```

```

 Column(modifier = modifier.fillMaxSize(),
 verticalArrangement = Arrangement.Center,
 horizontalAlignment = Alignment.CenterHorizontally
) {
 Text(
 text = "$number"
)
 Button(onClick = onButtonClick) {
 Text(
 text = stringResource(id = R.string.randomize)
)
 }
 }
 }
}

```

9. Create the `@Composable` `Main` function, which will connect `MainViewModel` with `MainScreen`:

```

@Composable
fun Main(
 mainViewModel: MainViewModel,
 modifier: Modifier = Modifier
) {
 val number =
 mainViewModel.state.collectAsState().value
 MainScreen(number = number, onButtonClick = {
 mainViewModel.generateNextNumber()
 }, modifier)
}

```

10. Now, let's create our first container responsible for managing the `NumberRepository` dependency:

```

class ApplicationContainer {
 val numberRepository: NumberRepository =
 NumberRepositoryImpl(Random())
}

```

11. Next, let's create a `MainApplication` class, which will hold a reference to the preceding repository. This will make our repository available across the entire application because the `Application` object will live longer than all the activities:

```
class MainApplication : Application() {

 val applicationContainer = ApplicationContainer()

}
```

12. Make sure to add your `Application` class to the `AndroidManifest.xml` file:

```
<application
 android:name=".MainApplication"
 ...
>
```

13. We will now move on to creating `MainContainer`, which will need a reference to the `NumberRepository` dependency and will provide a dependency to the `ViewModel` factory required to create `MainViewModel`:

```
class MainContainer(
 private val numberRepository: NumberRepository
) {
 fun getMainViewModelFactory():
 ViewModelProvider.Factory
 {
 return object : ViewModelProvider.Factory {
 override fun <T : ViewModel>
 create(modelClass: Class<T>): T {
 return MainViewModel(numberRepository) as T
 }
 }
 }
}
```

14. Modify `MainActivity` to initialize and access our containers:

```
class MainActivity : ComponentActivity() {
 override fun onCreate(savedInstanceState: Bundle?) {
 super.onCreate(savedInstanceState)
 val mainContainer = MainContainer(

```

```

 (application as MainApplication)
 .applicationContainer.numberRepository
)
 ...
}
}

```

In the highlighted code, we can see that we are using the repository defined in `ApplicationContainer` and injecting it into `MainContainer`, which will then inject it into `ViewModel` through `ViewModelProvider.Factory`.

15. Modify `MainActivity` so that you can set the content using the `MainContainer` object:

```

class MainActivity : ComponentActivity() {
 override fun onCreate(savedInstanceState: Bundle?) {
 ...
 setContent {
 Exercise1301Theme {
 Scaffold(
 modifier = Modifier.fillMaxSize()
) { innerPadding ->
 Main(
 mainViewModel = viewModel(
 factory = mainContainer
 .getMainViewModelFactory()
),
 modifier = Modifier.padding(innerPadding)
)
 }
 }
 }
 }
}

```

The preceding example should render the output presented in *Figure 13.6*:

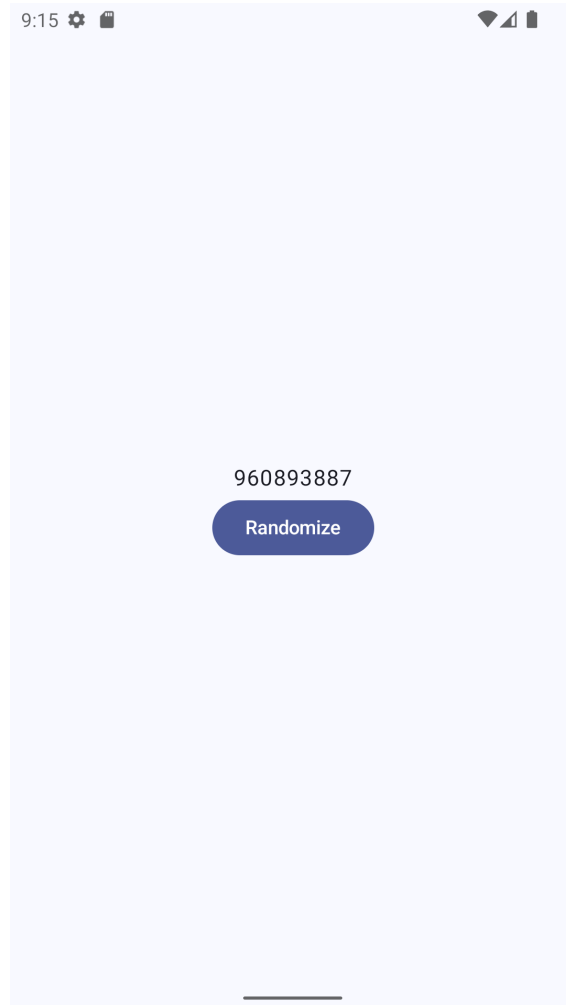


Figure 13.6 – Emulator output of Exercise 13.01 displaying a randomly generated number

Manual DI is an easy way to set up your dependencies in situations where the app is small, but it can become extremely difficult as the app grows. Imagine if, in *Exercise 13.01 – manual injection*, we had two classes that extended from `NumberRepository`. How would we handle such a scenario? How would developers know which one went in what activity? These types of questions have become common in most of the well-known apps on Google Play, which is why manual DI is rarely used. When used, it normally takes the form of a DI framework like the ones we will look over next, which offer a standardized approach to group the app dependencies into different modules and then access them through containers such as components.



## Using Dagger 2

**Dagger 2** offers a comprehensive way to organize your application's dependencies. It has the advantage of being adopted first on Android by the developer community before Kotlin was introduced. This is one of the reasons that many Android applications use Dagger as their DI framework.

Another advantage the framework holds is for Android projects written in Java because the library is developed in the same language. The framework was initially developed by Square (Dagger 1) and later transitioned to Google (Dagger 2). We will cover Dagger 2 in this chapter and describe its benefits.

Some of the key functionality that Dagger 2 provides is listed here:

- Injection
- Dependencies grouped in modules
- Components used to generate dependency graphs
- Qualifiers
- Scopes
- Subcomponents

Annotations are the key elements when dealing with Dagger. They generate the code required to perform the DI through an annotation processor. The main annotations can be grouped as follows:

- **Provider:** Classes that are annotated with `@Module` are responsible for providing an object (dependent object) that can be injected
- **Consumer:** The `@Inject` annotation is used to define a dependency
- **Connector:** A `@Component`-annotated interface defines the connection between the provider and the consumer

You will need to add the following dependencies to the `app/build.gradle.kts` file to add Dagger to your project:

```
implementation("com.google.dagger:dagger:2.56.2")
ksp("com.google.dagger:dagger-compiler:2.56.2")
```

Since we are dealing with annotation and symbol processors, in the same `build.gradle.kts` file, you will need to add the plugin for them:

```
plugins {
 ...
```

```
alias("com.google.devtools.ksp:2.0.21-1.0.25")
}
```

You should now have an idea of how Dagger 2 goes about performing DI. Next, we will look at each group of annotations Dagger 2 offers.

## Consumers

Dagger uses the `javax.inject.Inject` annotation to identify objects that require injection. There are multiple ways to inject dependencies, but the recommended ways are through constructor injection and field injection. Constructor injection looks like the following code:

```
import javax.inject.Inject
class ClassA @Inject constructor()
class ClassB @Inject constructor(private val classA: ClassA)
```

When constructors are annotated with `@Inject`, Dagger will generate the Factory classes responsible for instantiating the objects. In the example of `ClassB`, Dagger will try to find the appropriate dependencies that fit the signature of the constructor, which, in this example, is `ClassA`, which Dagger already created an instance of.

If you do not want Dagger to manage the instantiation of `ClassB` but still have the dependency on `ClassA` injected, you can use field injection, which will look something like this:

```
import javax.inject.Inject
class ClassA @Inject constructor()
class ClassB {
 @Inject
 lateinit var classA: ClassA
}
```

In this case, Dagger will generate the necessary code just to inject the dependency between `ClassB` and `ClassA`.

## Providers

You will find yourself in situations where your application uses external dependencies. That means that you cannot provide instances through constructor injection. Another situation where constructor injection is not possible is when interfaces or abstract classes are used.

In this situation, Dagger can provide the instance using the `@Provides` annotation. You will then need to group the methods where instances are provided into modules annotated with `@Module`:

```
import dagger.Module
import dagger.Provides
class ClassA
class ClassB(private val classA: ClassA)
@Module
object MyModule {
 @Provides
 fun provideClassA(): ClassA = ClassA()
 @Provides
 fun provideClassB(classA: ClassA): ClassB = ClassB(classA)
}
```

As you can see in the preceding example, `ClassA` and `ClassB` don't have any Dagger annotations. A module was created that will provide the instance for `ClassA`, which will then be used to provide the instance for `ClassB`. In this case, Dagger will generate a Factory class for each of the `@Provides`-annotated methods.

## Connectors

Assuming we will have multiple modules, we must combine them in a graph of dependencies that can be used across the application. Dagger offers the `@Component` annotation. This is usually used for an interface or an abstract class that will be implemented by Dagger.

Along with assembling the dependency graph, components also offer the functionality to add methods to inject dependencies into a certain object's members. In components, you can specify provision methods that return dependencies provided in the modules:

```
import dagger.Component
@Component(modules = [MyModule::class])
interface MyComponent {
 fun inject(myApplication: MyApplication)
}
```

For the preceding Component, Dagger will generate a `DaggerMyComponent` class, and we can build it as described in the following code:

```
import android.app.Application
import javax.inject.Inject
```

```
class MyApplication : Application() {
 @Inject
 lateinit var classB: ClassB
 override fun onCreate() {
 super.onCreate()
 val component = DaggerMyComponent.create()
 //needs to build the project once to generate
 //DaggerMyComponent.class
 component.inject(this)
 }
}
```

The Application class will create the Dagger dependency graph and component. The inject method in Component allows us to perform DI on the variables in the Application class annotated with @Inject, giving us access to the ClassB object defined in the module.

## Qualifiers

You can use qualifiers if you want to provide multiple instances of the same class (such as injecting different strings or integers across an application). These are annotations that can help you identify instances. One of the most common ones is the @Named qualifier, as described in the following code:

```
@Module
object MyModule {
 @Named("classA1")
 @Provides
 fun provideClassA1(): ClassA = ClassA()
 @Named("classA2")
 @Provides
 fun provideClassA2(): ClassA = ClassA()
 @Provides
 fun provideClassB(@Named("classA1") classA: ClassA):
 ClassB = ClassB(classA)
}
```

In this example, we create two instances of `ClassA` and give them different names. We then use the first instance whenever possible to create `ClassB`. We can also create custom qualifiers instead of the `@Named` annotation, as shown in the following code:

```
import javax.inject.Qualifier
@Qualifier
@MustBeDocumented
@kotlin.annotation.Retention(AnnotationRetention.RUNTIME)
annotation class ClassA1Qualifier
@Qualifier
@MustBeDocumented
@kotlin.annotation.Retention(AnnotationRetention.RUNTIME)
annotation class ClassA2Qualifier
```

The module can be updated like this:

```
@Module
object MyModule {
 @ClassA1Qualifier
 @Provides
 fun provideClassA1(): ClassA = ClassA()
 @ClassA2Qualifier
 @Provides
 fun provideClassA2(): ClassA = ClassA()
 @Provides
 fun provideClassB(@ClassA1Qualifier classA: ClassA):
 ClassB = ClassB(classA)
}
```

In the preceding snippet, we provide two instances of the `ClassA` type, using the `ClassA1Qualifier` and `ClassA2Qualifier` annotations to distinguish between them. We then inject the first instance into the `ClassB` type. This approach to qualifiers might be preferred so that when a change is made to the naming of your instances, both the provider and consumer are updated. Next, we will look at managing the life cycles of our dependencies.

## Scopes

If you want to keep track of the life cycle of your components and your dependencies, you can use scopes. Dagger offers a `@Singleton` scope. This usually indicates that your component will live as long as your application.

Scopes have no impact on the life cycles of the objects; they are built to help developers identify the life cycles of objects. Giving your components one scope and grouping your code to reflect that scope is recommended.

Some common Dagger scopes on Android are related to the activity or fragment:

```
import javax.inject.Scope
@Scope
@MustBeDocumented
@kotlin.annotation.Retention(AnnotationRetention.RUNTIME)
annotation class ActivityScope
@Scope
@MustBeDocumented
@kotlin.annotation.Retention(AnnotationRetention.RUNTIME)
annotation class FragmentScope
```

The annotation can be used in the module where the dependency is provided:

```
@ActivityScope
@Provides
fun provideClassA(): ClassA = ClassA()
```

The code for Component will be as follows:

```
@ActivityScope
@Component(modules = [MyModule::class])
interface MyComponent {
}
```

The preceding example indicates that Component can only use objects with the same scope. If any of the modules that are part of Component contain dependencies with different scopes, Dagger will throw an error indicating that there is something wrong with the scopes.

## Subcomponents

Something that goes together with scopes is subcomponents. They allow you to organize your dependencies for smaller scopes. One common use case on Android is to create subcomponents for activities and fragments. Subcomponents inherit dependencies from the parent, and they generate a new dependency graph for the scope of the subcomponent.

Let's assume we have a separate module, as shown here:

```
class ClassC
@Module
object MySubcomponentModule {
 @Provides
 fun provideClassC(): ClassC = ClassC()
}
```

A Subcomponent annotation that will generate a dependency graph for that module would look something like the following:

```
import dagger.Subcomponent
@ActivityScope
@Subcomponent(modules = [MySubcomponentModule::class])
interface MySubcomponent {
 fun inject(mainActivity: MainActivity)
}
```

The parent component would need to declare the new component, as shown in the following code snippet:

```
import dagger.Component
@Component(modules = [MyModule::class])
interface MyComponent {
 fun inject(myApplication: MyApplication)
 fun createSubcomponent(mySubcomponentModule:
 MySubcomponentModule): MySubcomponent
}
```

You can inject ClassC into your activity as follows:

```
@Inject
lateinit var classC: ClassC
override fun onCreate(savedInstanceState: Bundle?) {
 super.onCreate(savedInstanceState)
 (application as MyApplication).component
 .createSubcomponent(MySubcomponentModule)
 .inject(this)
}
```

With this knowledge, let's move on to the exercise.

## Exercise 13.02 – Dagger injection

In this exercise, we will write an Android application that will apply the concept of DI with Dagger. The application will have the same `Repository` and `ViewModel` defined in *Exercise 13.01 – manual injection*.

We will need to use Dagger to expose the same two dependencies:

- `Repository`: This will have the `@Singleton` scope and will be provided by `ApplicationModule`. Now, `ApplicationModule` will be exposed as part of `ApplicationComponent`.
- `ViewModelProvider.Factory`: This will have the custom-defined scope named `MainScope` and will be provided by `MainModule`. Now, `MainModule` will be exposed by `MainSubComponent`. Also, `MainSubComponent` will be generated by `ApplicationComponent`.

The app itself will display a randomly generated number each time a button is clicked. To achieve this, execute the following steps:

1. Create a new Android Studio project with an empty activity.
2. Let's start by adding Dagger and the `ViewModel` libraries to the `gradle/libs.versions.toml` file:

```
[versions]
...
kotlin = "2.0.21"
...
viewModelCompose = "2.8.7"
ksp = "2.0.21-1.0.25"
dagger2 = "2.56.2"
[libraries]
...
androidx-viewmodel-compose = { group = "androidx.lifecycle", name =
 "lifecycle-viewmodel-compose", version.ref = "viewModelCompose"
}
dagger2 = { group = "com.google.dagger", name = "dagger",
 version.ref = "dagger2" }
dagger2-compiler = { group = "com.google.dagger",
 name = "dagger-compiler", version.ref = "dagger2" }
[plugins]
...
ksp = { id = "com.google.devtools.ksp", version.ref = "ksp" }
```



3. We also need the ksp plugin in the app/build.gradle.kts module. Attach the plugin as shown in the following code snippet. We also need to make sure that the Compose compiler version is compatible with the Kotlin version:

```
plugins {
 ...
 alias(libs.plugins.ksp)
}
```

4. We now need to add the NumberRepository, NumberRepositoryImpl, MainViewModel, and RandomApplication classes and build our UI with MainActivity. This can be done by following *steps 3–6 from Exercise 13.01 – manual injection*.
5. Now, let's move on to ApplicationModule, which will provide the NumberRepository dependency:

```
@Module
class ApplicationModule {
 @Provides
 fun provideRandom(): Random = Random()
 @Provides
 fun provideNumberRepository(random: Random):
 NumberRepository = NumberRepositoryImpl(random)
}
```

6. Now, let's create MainModule, which will provide the instance of ViewModel.Factory:

```
@Module
class MainModule {
 @Provides
 fun provideMainViewModelFactory(
 numberRepository: NumberRepository
): ViewModelProvider.Factory {
 return object : ViewModelProvider.Factory {
 override fun <T : ViewModel>create(
 modelClass: Class<T>): T {
 return MainViewModel(numberRepository) as T
 }
 }
 }
}
```

7. Now, let's create `MainScope`:

```
@Scope
@MustBeDocumented
@kotlin.annotation.Retention(
 AnnotationRetention.RUNTIME)
annotation class MainScope
```

8. We will need `MainSubcomponent`, which will use the preceding scope:

```
@MainScope
@Subcomponent(modules = [MainModule::class])
interface MainSubcomponent {
 fun inject(mainActivity: MainActivity)
}
```

9. Next, we will require `ApplicationComponent`:

```
@Singleton
@Component(modules = [ApplicationModule::class])
interface ApplicationComponent {
 fun createMainSubcomponent(): MainSubcomponent
}
```

10. Next, we modify the `MainApplication` class to add the code required to initialize the Dagger dependency graph:

```
class MainApplication : Application() {

 lateinit var applicationComponent:
 ApplicationComponent

 override fun onCreate() {
 super.onCreate()
 applicationComponent = DaggerApplicationComponent.create()
 }
}
```

Make sure that `android:name` is set in `AndroidManifest.xml` like in *Exercise 13.01 – manual injection*.

11. We now modify the MainActivity class to inject ViewModelProvider.Factory and initialize ViewModel so that we can display the random number:

```
class MainActivity : AppCompatActivity() {
 @Inject
 lateinit var factory: ViewModelProvider.Factory
 override fun onCreate(savedInstanceState: Bundle?) {
 (application as RandomApplication)
 .applicationComponent
 .createMainSubcomponent()
 .inject(this)
 super.onCreate(savedInstanceState)
 ...
 }
}
```

12. Set the content on MainActivity using the ViewModelProvider.Factory object injected in the previous code snippet:

```
class MainActivity : ComponentActivity() {
 @Inject
 lateinit var factory: ViewModelProvider.Factory

 override fun onCreate(savedInstanceState: Bundle?) {
 ...
 setContent {
 Exercise1302Theme {
 Scaffold(
 modifier = Modifier.fillMaxSize()
) { innerPadding ->
 Main(
 mainViewModel = viewModel(
 factory = factory),
 modifier = Modifier
 .padding(innerPadding)
)
 }
 }
 }
 }
}
```

```
}
}
```

13. We will need to navigate to **Build** and click on **Rebuild project** in Android Studio so that Dagger will generate the code for performing the DI.

If you run the preceding code, it will build an application that will display a different random output when you click the button:

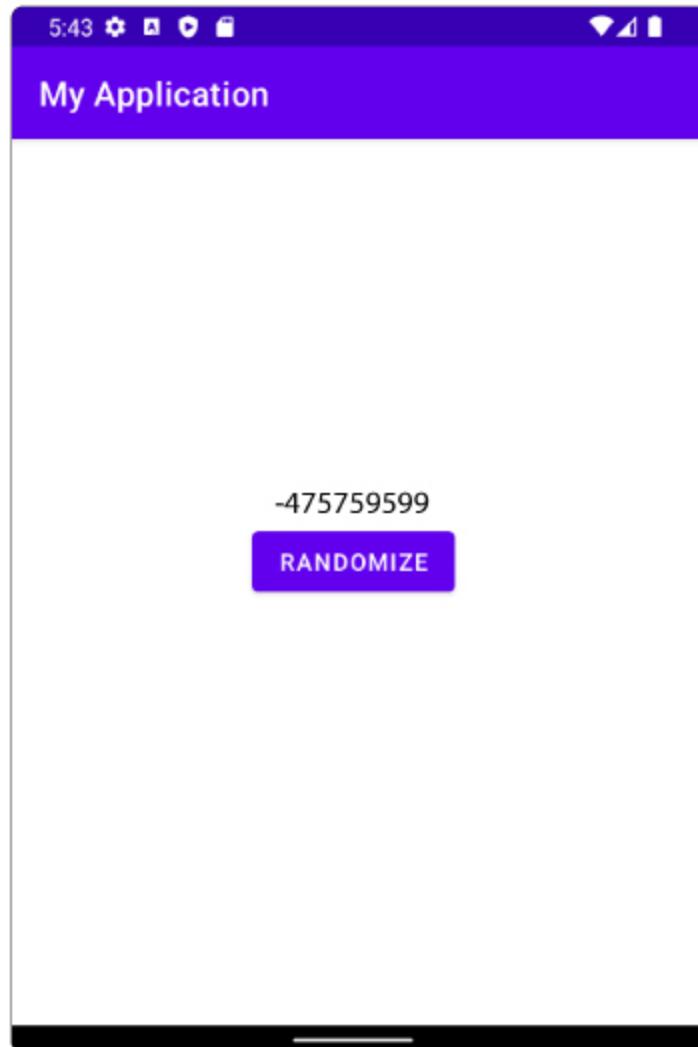


Figure 13.2 – Emulator output of Exercise 13.02 displaying a randomly generated number

Figure 13.3 shows what the application looks like. You can view the generated Dagger code in the app/build folder:

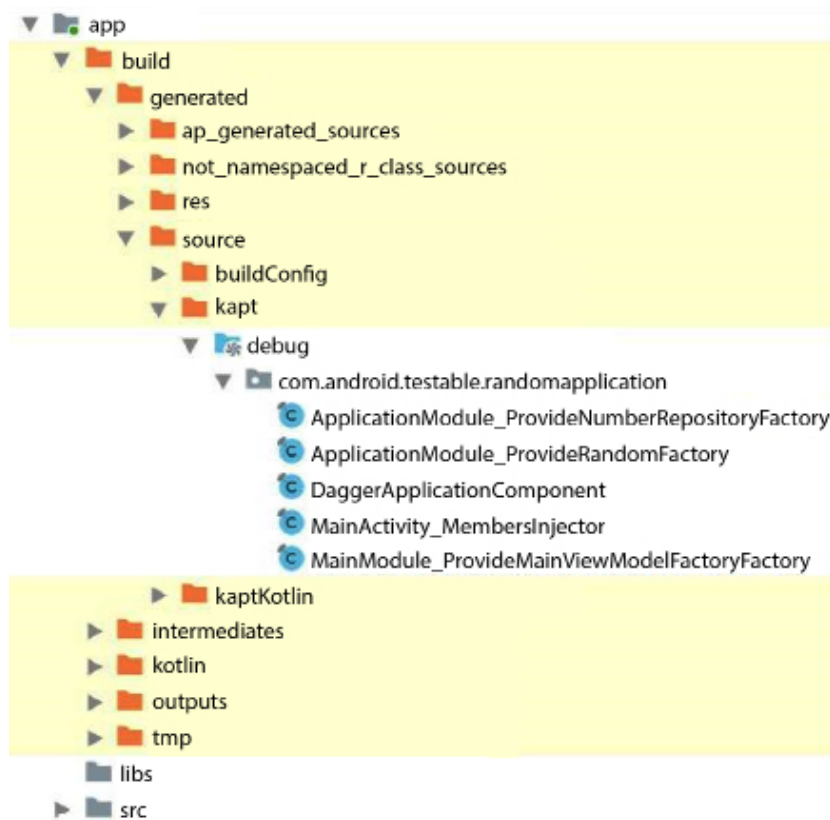


Figure 13.3 – Generated Dagger code for Exercise 13.02

In Figure 13.3, we can see the code that Dagger generated to satisfy the relationship between dependencies. For every dependency that needs to be injected, Dagger will generate an appropriate Factory class (based on the Factory design pattern), which will be responsible for creating the dependency.

Dagger also looks at the places where dependencies will need to be injected and generates an Injector class. This class has the responsibility of assigning the value to the dependency (in this case, it will assign the value to the members annotated with `@Inject` in the `MainActivity` class).

Finally, Dagger creates implementations for the interfaces that have the `@Component` annotation. In the implementation, Dagger will handle how the modules are created and also provide a builder in which developers can specify how modules can be built.

A common setup you will find for Android applications when it comes to organizing their dependencies is as follows:

- **ApplicationModule:** This is where dependencies common to the entire project are defined. Objects such as context, resources, and other Android framework objects can be provided here.
- **NetworkModule:** This is where dependencies related to API calls are stored.
- **StorageModule:** This is where dependencies related to persistence are stored. It can be split into **DatabaseModule**, **FilesModule**, **SharedPreferencesModule**, and so on.
- **ViewModelsModule:** This is where dependencies to **ViewModels** or the **ViewModel** factories are stored.
- **FeatureModule:** This is where dependencies are organized for a particular activity or fragment with their own **ViewModel**. Here, either subcomponents or Android injectors are used for this purpose.

We've raised some questions about how manual DI can go wrong and seen how Dagger can address these issues. Although it does the job, and it does it quickly, it is also a complex framework with a very steep learning curve. Some of these problems are solved by an Android-specific library called **Hilt**, which we will look at in the next section.

## Switching to Hilt

When we use Dagger in an Android application, there is a bit of boilerplate code we are forced to write. Some of it is around dealing with the life cycles of objects linked with activities and fragments, which leads us to create subcomponents; other parts are around the usage of **ViewModel** objects.

An attempt to simplify Dagger for Android was made with the **Dagger-Android** library, but later, a new library was developed on top of Dagger called **Hilt**. This library simplifies much of the Dagger usage through the use of new annotations, which leads to more boilerplate code that can be generated.

To use Hilt in a project, we will need the following plugins:

```
plugins {
 ...
 alias("com.google.devtools.ksp:1.9.20-1.0.14")
 alias("com.google.dagger.hilt.android:2.49")
}
```

To add the Hilt library to your project, you need the following:

```
dependencies {
 implementation("com.google.dagger:hilt-android:2.49")
 ksp("com.google.dagger:hilt-compiler:2.49")
}
```

The first change Hilt makes is in the `Application` class. Instead of needing to invoke a particular Dagger component to be initialized, with Hilt, you can just use the `@HiltAndroidApp` annotation:

```
@HiltAndroidApp
class MyApplication : Application() {
}
```

The preceding snippet will let Hilt know the entry point into your application, and it will start generating the dependency graph.

Another benefit of Hilt comes when interacting with Android components such as activities, fragments, views, services, and broadcast receivers. For these, we can use the `@AndroidEntryPoint` annotation to inject dependencies into each of these classes, which looks like the following:

```
@AndroidEntryPoint
class MyActivity : AppCompatActivity() {
 @Inject
 lateinit var myObject: MyObject
}
```

In the preceding snippet, the usage of `@AndroidEntryPoint` allows Hilt to inject `myObject` into `MyActivity`. A similar approach can be used for injecting dependencies into `ViewModel` objects, through the `@HiltViewModel` annotation:

```
@HiltViewModel
class MyViewModel @Inject constructor(
 private val myObject: MyObject
) : ViewModel()
```

In the preceding snippet, the `@HiltViewModel` annotation allows Hilt to inject `myObject` into `MyViewModel`. We can also observe that the `@Inject` annotation, carried over from Dagger, does not require the use of modules.

When it comes to modules, Hilt follows the same approach as Dagger with one minor addition: the usage of the `@InstallIn` annotation. This associates the annotated module with a particular component. Hilt provides a set of prebuilt components, such as `SingletonComponent`, `ViewModelComponent`, `ActivityComponent`, `FragmentComponent`, `ViewComponent`, and `ServiceComponent`.

Each of these components links the life cycle of the dependencies inside the annotated module to the life cycles of the application, `ViewModel`, activity, fragment, view, and service:

```
@Module
@InstallIn(SingletonComponent::class)
class MyModule {
 @Provides
 fun provideMyObject(): MyObject = MyObject()
}
```

In the preceding snippet, we can see what a `@Module` annotation looks like in Hilt and how we can use the `@InstallIn` annotation to specify that `MyObject` lives as long as our application does.

When it comes to instrumented tests, Hilt provides useful annotations for changing the dependencies for the tests. If we want to take advantage of these features, then we need the following dependencies for tests:

```
androidTestImplementation("com.google.dagger:hilt-android-testing:2.49")
kaptAndroidTest("com.google.dagger:hilt-android-compiler:2.44.2")
```

We can then go to our test and introduce Hilt into it, as follows:

```
@HiltAndroidTest
class MyInstrumentedTest {
 @get:Rule
 var hiltRule = HiltAndroidRule(this)

 @Inject
 lateinit var myObject: MyObject

 @Before
 fun init() {
 hiltRule.inject()
 }
}
```



In the preceding snippet, the `@HiltAndroid` test and `hiltRule` are used to swap the dependencies used in the application with the test dependencies. The call to inject is what allows us to inject the `MyObject` dependency into the test class. To provide the test dependencies, we can write a new module in the `androidTest` folder, as follows:

```
@Module
@TestInstallIn(
 components = [SingletonComponent::class],
 replaces = [MyModule::class]
)
class MyTestModule {
 @Provides
 fun provideMyObject(): MyObject = MyTestObject()
}
```

Here, we are using the `@TestInstallIn` annotation, which will replace the existing `MyModule` from the dependency graph with `MyTestModule`, which can provide a different sub-class of the dependency we want to swap.

For Hilt to be initialized for the instrumented tests, we will need to define a custom test runner to provide a test application from the Hilt library. The runner might look like the following:

```
class HiltTestRunner : AndroidJUnitRunner() {

 override fun newApplication(cl: ClassLoader?, name:
String?, context: Context?): Application {
 return super.newApplication(cl,
 HiltTestApplication::class.java.name, context)
 }
}
```

This runner will need to be registered in `build.gradle` of the module running the test:

```
android {
 ...
 defaultConfig {
 ...
 testInstrumentationRunner "{app_package_name}.HiltTestRunner"
 }
}
```

In this section, we studied the Hilt library and its benefits when it comes to removing the boilerplate code that was required using Dagger. Next, we will move on to an exercise in which we will replace Dagger 2 in an application with Hilt.

## Exercise 13.03 – Hilt injection

Modify *Exercise 13.02 – Dagger injection* such that the `@Component` and `@Subcomponent` classes are removed and Hilt is used instead:

1. Make sure that the following libraries are present in `gradle/libs.versions.toml`:

```
[versions]
...
kotlin = "2.0.21"
...
viewModelCompose = "2.8.7"
ksp = "2.0.21-1.0.25"
hilt = "2.56.2"

[libraries]
...
androidx-viewmodel-compose = { group = "androidx.lifecycle", name =
 "lifecycle-viewmodel-compose", version.ref = "viewModelCompose"
}
hilt-android = { group = "com.google.dagger", name = "hilt-android",
 version.ref = "hilt" }
hilt-android-compiler = { group = "com.google.dagger", name =
 "hilt-android-compiler", version.ref = "hilt" }

[plugins]
...
ksp = { id = "com.google.devtools.ksp", version.ref = "ksp" }
hilt = { id = "com.google.dagger.hilt.android", version.ref = "hilt"
}
```

2. Make sure that the plugins are added to the root `build.gradle.kts` but aren't applied to the entire project:

```
plugins {
 ...
 alias(libs.plugins.ksp) apply false
 alias(libs.plugins.hilt) apply false
}
```

3. Add the Hilt plugin in `app/build.gradle.kts`:

```
plugins {
 ...
 alias(libs.plugins.ksp)
 alias(libs.plugins.hilt)
}
```

4. In the same file, replace the Dagger dependencies with Hilt dependencies and add the fragments extension library used for generating `ViewModel`:

```
dependencies {
 ...
 implementation(libs.androidx.viewmodel.compose)
 implementation(libs.hilt.android)
 ksp(libs.hilt.android.compiler)
}
```

5. Delete the `ApplicationComponent`, `MainModule`, `MainScope`, and `MainSubcomponent` files from the project.
6. Add the `@InstallIn` annotation to `ApplicationModule`:

```
@Module
@InstallIn(SingletonComponent::class)
class ApplicationModule {
}
```

7. Remove all the code from inside `MainApplication` and add the `@HiltAndroidApp` annotation:

```
@HiltAndroidApp
class MainApplication : Application()
```

8. Modify MainViewModel to add the `@HiltViewModel` and `@Inject` annotations:

```
@HiltViewModel
class MainViewModel @Inject constructor(private val
numberRepository: NumberRepository) :
 ViewModel() {
 ...
}
```

9. Modify MainActivity to instead inject MainViewModel, remove all the component dependencies that were deleted previously, and add the `@AndroidEntryPoint` annotation:

```
@AndroidEntryPoint
class MainActivity : ComponentActivity() {
 override fun onCreate(savedInstanceState: Bundle?) {
 super.onCreate(savedInstanceState)
 enableEdgeToEdge()
 setContent {
 Exercise1303Theme {
 Scaffold(
 modifier = Modifier.fillMaxSize()
) { innerPadding ->
 Main(
 mainViewModel = viewModel(),
 modifier = Modifier.padding(innerPadding)
)
 }
 }
 }
 }
}
```

If we run the code, we should see the following output:

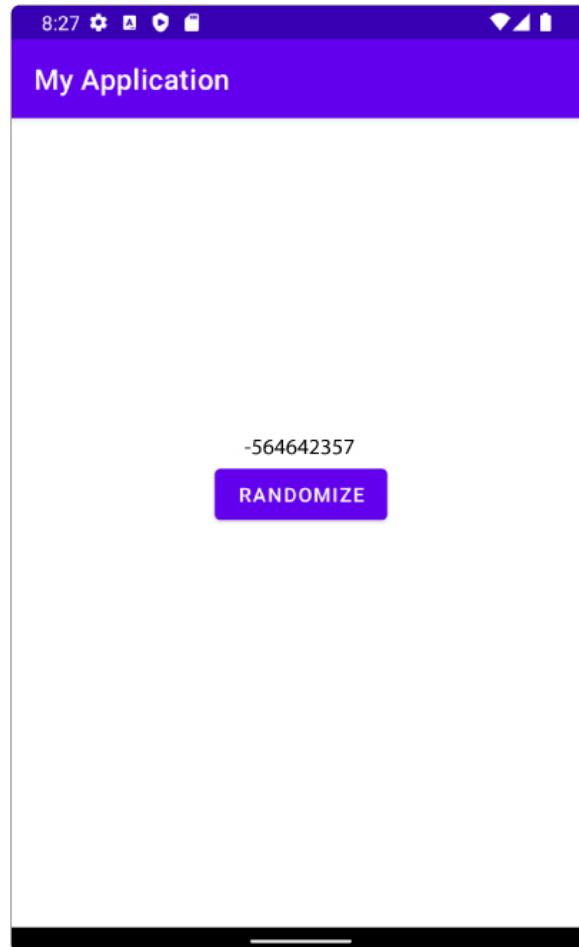


Figure 13.4 – Output of Exercise 13.03

We can see how much we can simplify an application's code by using Hilt instead of Dagger. For example, we no longer need to deal with the `@Component` and `@Subcomponent`-annotated classes and can manage subcomponents in the application component, and we don't need to manually initialize the dependency graph from the `Application` class because Hilt handles this for us. These are some of the main reasons why Hilt has become the most adopted library for DI in Android applications. By default, Hilt is compatible with Jetpack Compose and can use the `viewModel` function. However, if the navigation library is added to your project and you also wish to use the Hilt library, then you will need to add the Hilt navigation-compose library ("androidx.hilt:hilt-navigation-compose:1.2.0"). This will provide the `hiltViewModel` function and will scope the `ViewModel` object that you wish to use to the life cycle of your screen.



To oversimplify Hilt, it's best to view it as Dagger 2. This means that most things that work in Dagger 2 will work in Hilt. Then, you have prebuilt components such as `SingletonComponent` and `ActivityComponent`, along with the `@InstallIn` and `@TestInstallIn` annotations to avoid managing `@Component`-annotated interfaces. Finally, you need to deal with Android-specific components, which means that you need to use the `@AndroidEntryPoint`, `@HiltViewModel`, and `@HiltAndroidApp` annotations and `HiltTestApplication` when you write UI tests. In other words, you can say that Hilt is a Dagger 2 standard applied to Android projects.

Both Dagger 2 and Hilt rely on code generation and having the dependencies ready at compile time. In the next section, we will look at how to manage dependencies at runtime with the use of the Koin library.

## Using Koin

Koin is a lighter framework that is suitable for smaller apps. It requires no code generation and is built based on Kotlin's functional extensions. It is also a **Domain-Specific Language (DSL)**. You may have noticed that when using Dagger, a lot of code must be written to set up the DI. Koin's approach to DI solves most of those issues, allowing faster integration.

Koin can be added to your project by adding the following dependency to your `build.gradle.kts` file:

```
implementation(platform("io.insert-koin:koin-bom:4.0.0-RC1"))
implementation("io.insert-koin:koin-core")
```

The preceding code snippet will provide the bill of materials and then add each dependency you need.

To set up Koin in your application, you need the `startKoin` call with the DSL syntax:

```
class MyApplication : Application() {
 override fun onCreate() {
 super.onCreate()
 startKoin {
 androidLogger(Level.INFO)
 androidContext(this@MyApplication)
 androidFileProperties()
 modules(myModules)
 }
 }
}
```

```
 }
 }
}
```

Here, you can configure what your application context is (in the `androidContext` method). Then, you specify property files to define Koin configurations (in `androidFileProperties`). After that, you state the logger level for Koin, which will output in LogCat the results of Koin operations depending on the level (in the `androidLogger` method). Finally, you list the modules your application uses. A similar syntax is used to create the modules:

```
class ClassA
class ClassB(private val classB: ClassA)
 val moduleForClassA = module {
 single { ClassA() }
 }
 val moduleForClassB = module {
 factory { ClassB(get()) }
 }
 override fun onCreate() {
 super.onCreate()
 startKoin {
 androidLogger(Level.INFO)
 androidContext(this@MyApplication)
 androidFileProperties()
 modules(listOf(moduleForClassA,
 moduleForClassB))
 }
 }
}
```

In the preceding example, the two objects will have two different life cycles. When a dependency is provided using the `single` notation, only one instance will be used across the entire application life cycle. This is useful for repositories, databases, and API components, where multiple instances will be costly for the application.

The `factory` notation will create a new object every time an injection is performed. This may be useful in situations when an object needs to live as long as an activity or fragment.

The dependency can be injected using the `by inject()` method or the `get()` method, as shown in the following code:

```
class MainActivity : AppCompatActivity() {
 val classB: ClassB by inject()
}

override fun onCreate(savedInstanceState: Bundle?) {
 super.onCreate(savedInstanceState)
 val classB: ClassB = get()
}
```

Koin also offers the possibility of using qualifiers with the help of the `named()` method when the module is created. This allows you to provide multiple implementations of the same type (for example, providing two or more list objects with different content):

```
val moduleForClassA = module {
 single(named("name")) { ClassA() }
}
```

One of Koin's main features for Android applications is scopes for activities and fragments, and can be defined as shown in the following code snippet:

```
val moduleForClassB = module {
 scope(named<MainActivity>()) {
 scoped { ClassB(get()) }
 }
}
```

The preceding example connects the life cycle of the `ClassB` dependency to the life cycle of `MainActivity`. In order for you to inject your instance into your activity, you will need to extend the `ScopeActivity` class. This class is responsible for holding a reference as long as the activity lives. Similar classes exist for other Android components, such as fragments (`ScopeFragment`) and services (`ScopeService`):

```
class MainActivity : ScopeActivity() {
 val classB: ClassB by inject()
}
```

You can inject the instance into your activity using the `inject()` method. This is useful when you wish to limit who gets to access the dependency. In the preceding example, if another activity had wanted to access the reference to `ClassB`, then it wouldn't be able to find it in the scope.



Another feature that comes in handy for Android is the `ViewModel` injections. To set this up, you will need to add the library to `build.gradle.kts`:

```
implementation("io.insert-koin:koin-android")
```

If you recall, `ViewModel` requires `ViewModelProvider.Factories` in order to be instantiated. Koin automatically solves this, allowing `ViewModel` to be injected directly and to handle the factory work:

```
val moduleForClassB = module {
 factory {
 ClassB(get())
 }
 viewModel { MyViewModel(get()) }
}
```

To inject the dependency of `ViewModel` into your activity, you can use the `viewModel()` method:

```
class MainActivity : AppCompatActivity() {
 val model: MyViewModel by viewModel()
}
```

Alternatively, you can use the method directly:

```
override fun onCreate(savedInstanceState: Bundle?) {
 super.onCreate(savedInstanceState)
 val model : MyViewModel = getViewModel()
}
```

As we can see in the preceding setup, Koin takes full advantage of Kotlin's language features and reduces the amount of boilerplate required to define your modules and their scopes.

## Exercise 13.04 – Koin injection

In this exercise, we will write an Android application that will perform DI using Koin. The application will be based on *Exercise 13.01 – manual injection*, by keeping `NumberRepository`, `NumberRepositoryImpl`, `MainViewModel`, and `MainActivity`. The following dependencies will be injected:

- `Repository`: As part of a module named `appModule`.
- `MainViewModel`: This will rely on Koin's specialized implementation for `ViewModels`. This will be provided as part of a module named `mainModule` and will have the `MainActivity` scope.

Perform the following steps to complete the exercise:

1. Let's start by adding the right library in `gradle/libs.versions.toml`:

```
[versions]
...
viewModelCompose = "2.8.7"
koin = "4.0.4"

[libraries]
...
androidx-viewmodel-compose = { group = "androidx.lifecycle", name =
 "lifecycle-viewmodel-compose", version.ref = "viewModelCompose"
}
koin-bom = { group = "io.insert-koin", name = "koin-bom",
 version.ref = "koin" }
koin-core = { group = "io.insert-koin", name = "koin-core",
 version.ref = "koin" }
koin-android = { group = "io.insert-koin", name = "koin-android",
 version.ref = "koin" }
koin-compose = { group = "io.insert-koin", name =
 "koin-androidx-compose", version.ref = "koin" }
```

2. Now, let's add the dependencies to `app/build.gradle.kts`:

```
dependencies {
 ...
 implementation(libs.androidx.viewmodel.compose)
 implementation(platform(libs.koin.bom))
 implementation(libs.koin.core)
 implementation(libs.koin.android)
 implementation(libs.koin.compose)
 ...
}
```

3. Next, define the `appModule` variable inside the `MainApplication` class. This will have a similar structure to `AppModule` with the Dagger 2 setup from the *Using Dagger 2* section:

```
class MainApplication : Application() {
 val appModule = module {
 single {
```

```

 Random()
 }
 single<NumberRepository> {
 NumberRepositoryImpl(get())
 }
}
}

```

4. Now, let's add the `viewModel` module variable after `appModule`:

```

val mainModule = module {
 viewModel {
 MainViewModel(get())
 }
}

```

5. Next, let's initialize Koin in the `onCreate()` method of `RandomApplication`:

```

super.onCreate()
startKoin {
 androidLogger()
 androidContext(this@MainApplication)
 modules(listOf(appModule, mainModule))
}

```

6. Finally, let's inject the dependencies into the activity:

```

class MainActivity : ComponentActivity() {
 override fun onCreate(savedInstanceState: Bundle?) {
 super.onCreate(savedInstanceState)
 enableEdgeToEdge()
 setContent {
 Exercise1304Theme {
 Scaffold(
 modifier = Modifier.fillMaxSize()
) { innerPadding ->
 Main(
 mainViewModel = koinViewModel<
 MainViewModel
 >(),
 modifier = Modifier

```

```
 .padding(innerPadding)
)
}
}
}
```

Here, we are using `koinViewModel<MainViewModel>()` to obtain the instance of our `ViewModel`. This function comes from the `implementation(libs.koin.compose)` library, which offers extension functions specifically for Compose.

In *Figure 13.5*, we can see the same output as in previous exercises:

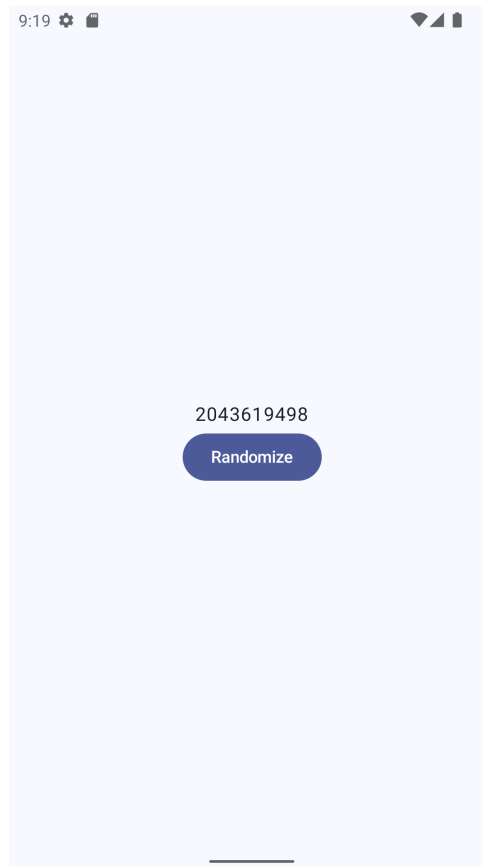


*Figure 13.5 – Emulator output of Exercise 13.04 displaying a randomly generated number*

As we can see from this exercise, Koin is much faster and easier to integrate, especially with its `ViewModel` library. This comes in handy for small projects, but its performance will be impacted once projects grow.

## Activity 13.01 – injected repositories

In this activity, you are going to create an app in Android Studio that connects to a sample API, <https://jsonplaceholder.typicode.com/posts>, using the Retrofit library and retrieves a list of posts from the web page, which will then be displayed on the screen.

You will then need to set up a UI test in which you will check whether the data is asserted correctly on the screen, but instead of connecting to the actual endpoint, you will provide dummy data for the test to display on the screen. You will use the DI concept to swap the dependencies using Hilt when the app is executed, as opposed to when the app is being tested.

To achieve this, you will need to build the following:

- A network component that is responsible for downloading and parsing the JSON file
- A repository that accesses the data from the API layer
- A `ViewModel` instance that accesses the repository
- A `@Composable` function that will render the list of posts
- One UI test that will assert the rows and use a dummy object to generate the API data



Error handling can be avoided for this activity.

Perform the following steps to complete this activity:

1. In Android Studio, create an application with an empty activity (`MainActivity`) and add an `api` package where your API calls are stored.
2. Define a class responsible for the API calls.
3. Create a repository package.
4. Define a repository interface with one method, returning the list of posts.
5. Create the implementation for the repository class.
6. Create a `ViewModel` instance to call the repository class to retrieve the data.
7. Create the activity that will render the UI.
8. Set up a Hilt module that will initialize the network-related dependencies.

9. Create a Hilt module that will be responsible for defining the dependencies required for the activity.
10. Set up the UI tests and a test application and provide a separate `RepositoryModule` class, which will return a dependency holding dummy data.
11. Implement the UI test.



The solution to this activity can be found at <https://packt.link/kqxp>.

## Summary

In this chapter, we analyzed the concept of DI and how it should be applied to separate concerns and prevent objects from having the responsibility of creating other objects. We saw how this is of great benefit for testing. We started the chapter by analyzing the concept of manual DI. This served as a good example of how DI works and how it can be applied to an Android application; it served as the baseline when comparing the DI frameworks.

We also analyzed two of the most popular frameworks that help developers inject dependencies. We started with a powerful and fast framework called Dagger 2, which relies on annotation processors to generate code to perform an injection. We then looked at how Hilt reduces the complexity of Dagger for Android applications. We also investigated Koin, a lightweight framework written in Kotlin with slower performance but simpler integration and a lot of focus on Android components, which makes it ideal for smaller projects but might become harder to maintain as projects grow.

The exercises in this chapter were intended to explore how the same problem can be solved using multiple solutions and compare the degrees of difficulty between the solutions. In the activities for this chapter, we leveraged Dagger's, Hilt's, and Koin's modules to inject certain dependencies when running the app and other dependencies when running the tests on an application that uses `ViewModel`, repositories, and APIs to load data.

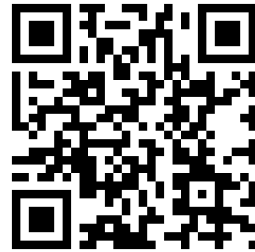
This was designed to show the seamless integration of multiple frameworks that achieve different goals. In the chapter's activity, we looked at how we can use Hilt to swap dependencies for testing purposes, inject dummy data, and then see whether it is displayed on the screen.

In the following chapter, you will have the opportunity to build upon the knowledge acquired thus far by diving into architectural patterns to properly structure your project's code.

**Unlock this book's exclusive benefits now**

Scan this QR code or go to [packtpub.com/unlock](https://packtpub.com/unlock), then search this book by name.

*Note: Keep your purchase invoice ready before you start.*

**UNLOCK NOW**

---

# Part 4

---

## Polishing and Publishing an App

In this part, we will look into different architecture patterns that further help with structuring an application's code. Then, we will look at advanced Jetpack Compose for enhancing applications with effects and animations. Finally, we will learn about the process involved in publishing an application on Google Play.

This part of the book includes the following chapters:

- *Chapter 14, Architecture Patterns*
- *Chapter 15, Advanced Jetpack Compose*
- *Chapter 16, Launching Your App on Google Play*





# 14

## Architecture Patterns

When developing an Android application, you may tend to write most of the code (including business logic) in activities or fragments. This will make your project hard to test and maintain later. As your project grows and becomes more complex, the difficulty also increases. You can improve your projects with architectural patterns.

Architectural patterns are general solutions for designing and developing parts of applications, especially for large apps. There are architectural patterns you can use to structure your project into different layers (the presentation layer, the **user interface (UI)** layer, and the data layer) or functions (observer/observable). With architectural patterns, you can organize your code in a way that makes it scalable and easier for you to develop, test, and maintain.

For Android development, commonly used patterns include **Model-View-Controller (MVC)**, **Model-View-Presenter (MVP)**, and **Model-View-ViewModel (MVVM)**. The recommended architectural pattern is MVVM, which will be discussed in this chapter. You will also learn about the Repository pattern using the Room library.

This chapter will introduce you to architectural patterns you can use for your Android projects. It covers using the MVVM pattern, adding `ViewModels`. You will also learn about using the Repository pattern for caching data.

By the end of the chapter, you will be able to structure your Android project using MVVM. You will also be able to use the Repository pattern with the Room library to save data locally in a database.

We will cover the following topics in the chapter:

- Getting started with MVVM
- Implementing the Repository pattern

## Technical requirements

The complete code for all the exercises and the activity in this chapter is available on GitHub at <https://github.com/PacktPublishing/How-to-Build-Android-Apps-with-Kotlin-Third-Edition/tree/main/Chapter14>.

## Getting started with MVVM

MVVM allows you to separate the UI and business logic. When you need to redesign the UI or update the model/business logic, you only need to touch the relevant component without affecting the other components of your app. This will make it easier for you to add new features and test your existing code. MVVM is also useful in creating huge applications that use a lot of data and views.

With the MVVM architectural pattern, your application will be grouped into three components:

- **Model:** This represents the data layer
- **View:** This is the UI that displays the data
- **ViewModel:** This fetches data from Model and provides it to View

The MVVM architectural pattern can be understood better through the following diagram:

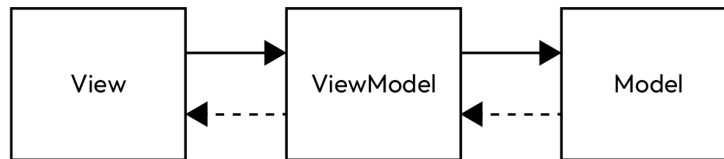


Figure 14.1 – The MVVM architectural pattern

The Model contains the data of the application.

The activities, fragments, and layouts that your users see and interact with are the views in MVVM. Views only deal with how the app looks. They let ViewModel know about user actions (such as opening an activity or clicking on a button).

ViewModel links View and Model. ViewModels handle the business logic by processing the data, where it retrieves it from the data layer, creates the required functions to solve your business needs, and then re-routes the processed information back to the UI layer by binding to it. Views subscribe to the ViewModel and update the UI when a value changes.

You can use Jetpack's ViewModel to create the ViewModel classes for your app. Jetpack's ViewModel is lifecycle-aware and is typically tied to its lifecycle-owner (activity or fragment) so you don't need to handle it yourself.

For example, if you're working on an app that displays movies, you could have `MovieViewModel`. This `ViewModel` will have a function that fetches a list of movies:

```
class MovieViewModel : ViewModel() {
 private val _movies: MutableStateFlow<List<Movie>>
 fun movies: StateFlow<List<Movie>> { ... }
 ...
}
```

Flow was used instead of `LiveData` as it is more flexible and is integrated with coroutines. In your activity, you can create `ViewModel` instance using `ViewModelProvider`:

```
class MainActivity : AppCompatActivity() {
 private val movieViewModel by lazy {
 ViewModelProvider(this).get(
 MovieViewModel::class.java
)
 }
 ...
}
```

Then, you can connect to the `movies` flow from `ViewModel` and automatically update the list on the UI when the list of movies changes:

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

 lifecycleScope.launch {
 repeatOnLifecycle(Lifecycle.State.STARTED) {
 launch {
 movieViewModel.movies.collect {
 movies -> movieAdapter.addMovies(movies)
 }
 }
 }
 }
 ...
}
```

Views are notified when values in `ViewModel` have changed. You can also use data binding to connect `View` with the data from `ViewModel`. You can call the data layer from `ViewModel`. To reduce its complexity, you can use the Repository pattern for loading and caching data. You will learn about this in the next section.

## Implementing the Repository pattern

Instead of `ViewModel` directly calling the services for getting and storing data, it should delegate that task to another component, such as a repository.

With the Repository pattern, you can move the code in the `ViewModel` class that handles the data layer into a separate class. This reduces the complexity of `ViewModel`, making it easier to maintain and test. The repository will manage where the data is fetched and stored, just as if the local database or the network service were used to get or store data:

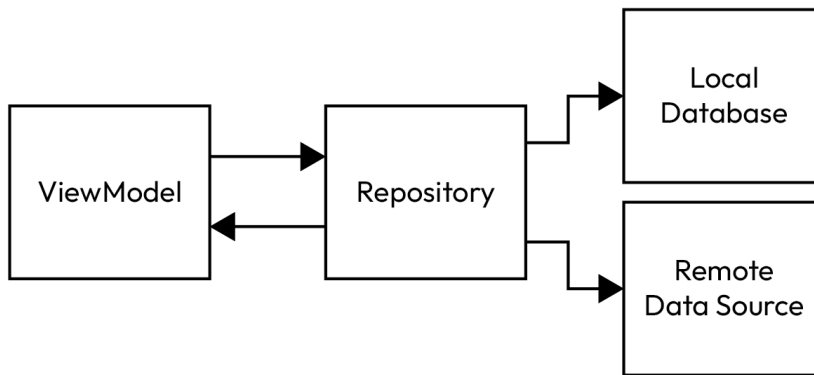


Figure 14.2 – *ViewModel with the Repository pattern*

In `ViewModel`, you can add a property for the repository:

```
class MovieViewModel(val repository: MovieRepository): ViewModel() { ...}
```

`ViewModel` will get the movies from the repository, or it can listen to them. It will not know where you actually got the list from.

You can create a repository interface that connects to a data source, such as in the following example:

```
interface MovieRepository {
 fun getMovies(): List<Movie>
}
```

The `MovieRepository` interface has a `getMovies` function that your repository implementation class will override to fetch movies from the data source. You can also have a single repository class that handles the fetching of data from either the local database or from your remote endpoint.

When using the local database as the data source for your repository, you can use the Room library, which makes it easier for you to work with the SQLite database by writing less code and having compile-time checks on queries.

Let's try adding the Repository pattern with Room to an Android project.

## Exercise 14.1 – using Repository with Room in an Android project

In *Chapter 10, Coroutines and Flow*, you worked on an application that displays popular movies using the Movie Database API. You can use the Android project from that chapter or make a copy of it. In this exercise, you will update the app with the Repository pattern.

When opening the app, it fetches the list of movies from the network. This takes a while. You will cache this data into the local database every time you fetch it. When the user opens the app next time, the app will immediately display the list of movies from the database on the screen. You will be using Room for data caching:

1. Open the Popular Movies project in Android Studio.
2. Update the `MovieRepository` class with the constructor property for `movieDatabase`:

```
class MovieRepository(
 private val movieService: MovieService,
 private val movieDatabase: MovieDatabase,
) {
 ...
}
```

3. Update the `getPopularMovies` function:

```
suspend fun getPopularMovies(): Flow<List<Movie>> {
 val movieDao: MovieDao = movieDatabase.movieDao()

 CoroutineScope(SupervisorJob() + Dispatchers.IO).launch {
 try {
 val movies = movieService.getPopularMovies()
 }
 }
}
```

```
 movieDao.addMovies(movies)
 } catch (exception: Exception) {
 Log.e(
 "MovieRepository",
 exception.toString()
)
 }
}
return movieDao.getMovies()
}
```

It will fetch the movies from the database first. Then, it will fetch the list from the network endpoint and save it.

4. Open `MovieApplication` and in the `onCreate` function, replace the `movieRepository` initialization with the following:

```
val movieDatabase =
 MovieDatabase.getInstance(applicationContext)
movieRepository = MovieRepository(
 movieService = movieService,
 movieDatabase=movieDatabase
)
```

5. Run the application. It will display the list of popular movies, and clicking on one will open the details of the movie selected. If you turn off mobile data or disconnect from the wireless network and then re-open the app, the app will still display the list of movies, which is now cached in the database:



Figure 14.3 – The Popular Movies app using Repository with Room

In this exercise, you have improved the app by moving the loading and storing of data into a repository. You also used Room to cache the data.



The repository fetches the data from the data source. If there's no data stored in the database yet, the app will call the network to request the data. This can take a while. You can improve the user experience by pre-fetching data at a scheduled time so the next time the user opens the app, they will already see the updated contents. You can do this with `WorkManager`

## Activity 14.1 – revisiting the TV Guide app

In *Chapter 10, Coroutines and Flow*, you developed an app that can display a list of TV shows that are on the air. The TV Guide app had two screens: the main screen and the details screen. On the main screen, there's a list of TV shows. When clicking on a TV show, the details screen will be displayed with the details of the selected show.

When running the app, it takes a while to display the list of shows. Update the app to cache the list so it will be immediately displayed when opening the app.

You can use the TV Guide app you worked on in *Chapter 10, Coroutines and Flow*, or download it from the GitHub repository (<https://packt.link/Ep0E5>).

The following steps will help guide you through this activity:

1. Open the TV Guide app in Android Studio.
2. Add an `Entity` annotation in the `TVShow` class.
3. Create a `TVShowDao` data access object for accessing the `tvshows` table.
4. Create a `TVShowDatabase` class.
5. Update `TVShowRepository` with a constructor property for `tvShowDatabase`.
6. Update the `getTVShows` function to get the TV shows from the local database. Afterward, retrieve the list from the endpoint and then save it in the database. Create the `TVShowWorker` class.
7. Open the `TVApplication` file. In `onCreate`, initialize `TVShowRepository` with the service and database.
8. Run your application. The app will display a list of TV shows. Clicking on a TV show will open the details activity, which displays the movie details. The main screen and details screen will be similar to the following:

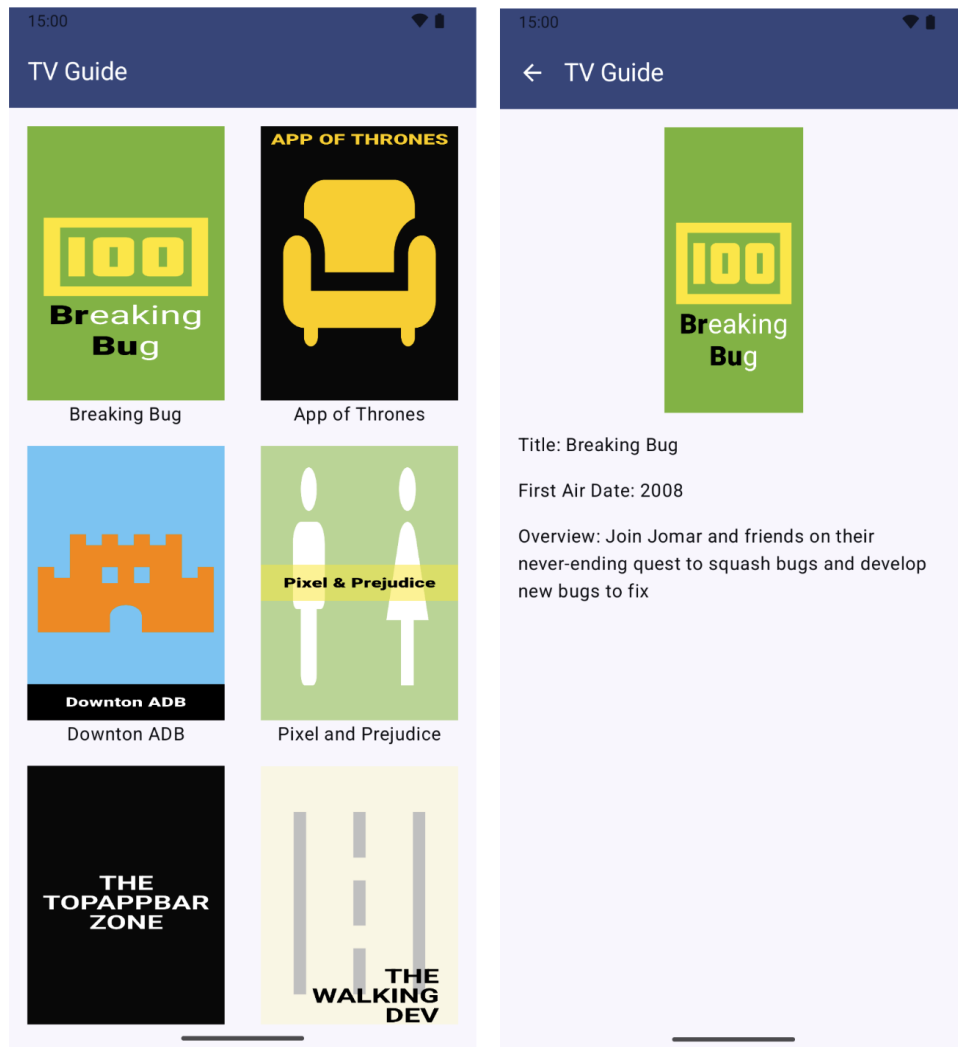


Figure 14.4 – The main screen and details screen of the TV Guide app

In this exercise, you have updated an app to use Repository with Room.



The solution to this activity can be found at <https://packt.link/C1bVh>.

## Summary

This chapter focused on architectural patterns for Android. You started with the MVVM architectural pattern. You learned about its three components: the model, the view, and the ViewModel. Next, you learned about how the Repository pattern can be used to cache data.

In the next chapter, you will learn about advanced Jetpack Compose topics such as effects. You will also learn how to improve the look and design of your apps using Jetpack Compose animations.

### Unlock this book's exclusive benefits now

Scan this QR code or go to [packtpub.com/unlock](https://packtpub.com/unlock), then search this book by name.

*Note: Keep your purchase invoice ready before you start.*

UNLOCK NOW



# 15

## Advanced Jetpack Compose

In the previous chapter, you learned about architecture patterns such as MVVM. You now know how to improve the architecture of an app. Next, we will learn how to improve your applications with `CompositionLocal`, side-effects, and animations to enhance your app's look and feel.

This chapter will introduce you to advanced Jetpack Compose topics. It offers a description of passing data through the **user interface (UI)** tree using `CompositionLocal` and the use of various side-effects in Jetpack Compose. You will also learn about animating Jetpack Compose Composable functions.

By the end of this chapter, you will be able to use `CompositionLocal` and side-effects such as `LaunchedEffect`, `DisposableEffect`, and `SideEffect` in your Jetpack Compose applications, as well as create animations using Jetpack Compose.

We will cover the following topics in this chapter:

- Using `CompositionLocal`
- Using side-effects with Jetpack Compose
- Creating animations using Jetpack Compose

### Technical requirements

The complete code for all the exercises and the activity in this chapter is available on GitHub at <https://packt.link/yXbKI>

## Using CompositionLocal

`CompositionLocal` is a Jetpack Compose API that allows you to pass down data to the UI tree, without having to explicitly call them. Instead of passing that data on each step of the hierarchy tree, you can use default values set with `CompositionLocal`. This will make your code cleaner and more concise. To access the current value of `CompositionLocal`, you can call `CompositionLocal.current` and use it in your composable.

The `MaterialTheme` composable from the Material Compose SDK, for example, uses `CompositionLocal` to provide default color schemes, shapes, and typography values. This makes it easy to access the theme values on any composable in your app. You can simply access a typography text style by calling the following:

```
MaterialTheme.typography.headlineLarge
```

`MaterialTheme.typography` uses a `CompositionLocal` instance to pass the typography set on the theme.

Jetpack Compose has existing `CompositionLocal` instances that are available for you to use in your applications. These `CompositionLocal` instances include the following:

- `LocalConfiguration`: Useful for determining how to organize the UI
- `LocalContext`: Provides a `Context` object that can be used by Android applications
- `LocalLifecycleOwner`: Contains the current `LifecycleOwner` object.
- `LocalSavedStateRegistryOwner`: Contains the current `SavedStateRegistryOwner` object.
- `LocalView`: Contains the current Compose View object.

There are more `CompositionLocal` implementations that are ready to use. Most of them start with `Local` so you can explore them using autocomplete in the Android Studio **integrated development environment (IDE)**.

You can also create your own `CompositionLocal` instance in your project. To do that, you can use either `staticCompositionLocalOf` or `compositionLocalOf`. The difference between the two is that when a value is changed, the `CompositionLocal` instance created with `staticCompositionLocalOf` function will cause recomposition of the whole UI tree. Meanwhile, value changes on `compositionLocalOf` only affect the composables that access that value.

The following example creates a `CompositionLocal` instance of margins for use in the application:

```
data class Margins(
 val horizontal: Dp,
 val vertical: Dp,
)

val margins = Margins(horizontal = 16.dp, vertical = 16.dp)
val LocalMargins = compositionLocalOf { margins }
```

You can access the `LocalMargins` instance in your composables with `LocalMargins.current`, such as by calling `LocalMargins.current.horizontal` to get horizontal margins. To change a value for `CompositionLocal`, wrap the composable with `CompositionLocalProvider` as shown in the following code block:

```
CompositionLocalProvider(LocalMargins provides margins) {
 ...
}
```

The `provides` keyword binds the margins value to the `LocalMargins` `CompositionLocal` instance and when you call the values inside the code block, it will be using the new values set.



It is a good practice to start the name of the custom `CompositionLocal` instance that you create with `Local`. This will make it easier for you and other developers in your team to see and use them.

The `CompositionLocal` API is useful when there are a lot of hierarchies in the tree and some of them are not accessing the data. One good use is when theming an app, as all composables would need to access the same theme values.

Let's try out what you have learned so far by adding activity transitions to an Android application.

## Exercise 15.01 – adding `CompositionLocal` in an app

In *Chapter 14, Architecture Patterns*, under the *Exercise 14.01 – using Repository with Room in an Android project* subsection, you worked on an application that displays popular movies using the `The Movie Database` API. For this chapter, you will be improving the app with advanced Jetpack Compose. You can use the *Popular Movies* project from the previous chapter or make a copy of it.

In this exercise, you will be adding `CompositionLocal` to the *Popular Movies* application:

1. In Android Studio, open the *Popular Movies* project.
2. Create a new file named `CompositionLocal.kt` and add the following contents to the file:

```
data class Margin(val margin: Dp)

val LocalMargin = compositionLocalOf {
 Margin(margin = 8.dp)
}
```

The `Margin` class is for specifying margins and the `LocalMargin` `CompositionLocal` instance is to be used by the composable functions later.

3. Open the `Theme` file and wrap the `MaterialTheme` lambda in a `CompositionLocalProvider` call:

```
CompositionLocalProvider(
 LocalMargin provides Margin(margin = 16.dp)
) {
 ...
}
```

This will create a `CompositionLocal` instance using a margin of `16.dp` as a value. You can now use that margin for specifying padding on any composable function using the theme.

4. Open the `MovieItemView` file and replace the padding in the title text with the margin from the `LocalMargin` variable:

```
modifier = Modifier
 .padding(horizontal = LocalMargin.current.margin),
```

This will use the same value (`16.dp`) that was specified in the `Theme` file in *step 3*.

5. Open the `DetailsActivity` file and on the `DetailsView` composable, replace the padding of the outer `Column` composable with the value from the `LocalMargin` variable:

```
padding(horizontal = LocalMargin.current.margin)
```

This will also use the same value (`16.dp`) for the padding of the `Column` composable as the one set in the theme file.

6. In the `Column` composable for the title and release texts, change the start padding to use the value from the `LocalMargin` instance:

```
Column(modifier = Modifier.padding(
 start = LocalMargin.current.margin)
) {
 ...
}
```

This will also use the same value (16.dp) for the start padding for the title and release column.

7. Wrap the overview text with a new `CompositionLocalProvider` instance to change the margin to use 8.dp as the value:

```
CompositionLocalProvider(
 LocalMargin provides Margin(margin = 8.dp)
) {
 ...
}
```

This will use 8.dp as the value for the overview text instead of the one set by the theme.

8. Run the application. The application should look and work the same as before, displaying a list of movies. Clicking on one should open the screen with the movie details.

You have added `CompositionLocal` to an application. In the next section, you will learn about side-effects in Jetpack Compose.

## Using side-effects with Jetpack Compose

With side-effects in Jetpack Compose, you can run non-UI tasks outside the composable. This separates UI rendering and non-UI operations, making the app more responsive and the code more organized. For example, you can perform a network operation only on the first load of the screen and not on every recomposition.

Jetpack Compose has several Composable functions that allow you to manage side effects, such as `LaunchedEffect`, `DisposableEffect`, and `SideEffect`.



## LaunchedEffect

`LaunchedEffect`, when entering the composition, runs a coroutine with the block passed. This coroutine will be canceled when `LaunchedEffect` leaves the composition. On recomposition, it will only recompute when the key/s change. `LaunchedEffect` is useful when you want to do network, database, or other long-running tasks.

You can pass one or more keys that will determine whether the code inside `LaunchedEffect` will be recomposed. Using `LaunchedEffect(Unit)` will ensure the block is only run once, during the initial composition.

For example, you can use `LaunchedEffect` as follows:

```
LaunchedEffect(Unit) {
 fetchItemsFromNetwork()
}
```

This will call `fetchItemsFromNetwork` only once, when the composable has its first composition. Meanwhile, the following `LaunchedEffect` composable uses an `id` variable for the key:

```
LaunchedEffect(key1 = id) {
 fetchItemById(id)
}
```

In this example, every time the value of the `id` variable is changed, `fetchItemById` will be called with the new `id` value.

## DisposableEffect

`DisposableEffect` is a side-effect used for operations that must be cleaned after the composable leaves the composition or if the keys change. This is useful when you are using resources to clean up or listeners and observers to unregister or remove after running the task. You must add an `onDispose` block at the end of the code that will run the cleanup.

You can use `DisposableEffect` as follows:

```
...

DisposableEffect(lifecycle) {
 ...

 lifecycle.addObserver(observer)
```

```
 onDispose {
 lifecycle.removeObserver(observer)
 }
 }
}
```

In this example, `DisposableEffect` will run every time the `lifecycle` instance, representing the life cycle of the activity or fragment, changes. It will add an observer and then on `onDispose`, it will remove the observer when leaving the composition.

## SideEffect

`SideEffect` is used to perform an effect every time there is a recomposition on the parent composable. That effect is an operation that does not depend on the composable state, such as logging or analytics.

The following example shows how you can use `SideEffect` in your composable functions:

```
@Composable
fun Item() {
 val text = ...
 ...

 SideEffect {
 Log.d(TAG, "value is $text")
 }
 ...
}
```

This will create a side effect that will log the value of `text` whenever the `Item` composable is recomposed.

Let's try out what you have learned so far by adding Jetpack Compose side-effects to an Android application.

## Exercise 15.02 – adding side-effects in an app

In this exercise, we will be adding Jetpack Compose side-effects to the *Popular Movies* application:

1. In Android Studio, open the *Popular Movies* project that you used in the *Exercise 15.01 – adding CompositionLocal in an app* exercise.

2. Open the `MainActivity` file and add the following lines at the top of the block of code of the `MainScreen` composable:

```
SideEffect {
 Log.d("MainScreen", "app loaded")
}
```

This will log the "app loaded" text when the `MainScreen` composable is first loaded and every time it is recomposed.

3. Run the application and you should see the "app loaded" text in the logs in Android Studio.
4. Open the `MovieItemView` file and add the following at the top of the block of the `MovieItemView` composable:

```
LaunchedEffect(movie) {
 Log.d("MovieItemView", "Movie: ${movie.title}")
}
```

This will log the title of the movie every time the `movie` variable changes its value.

5. Run the application. You should see various logs of movie titles in the initial load of the application and as you scroll the list.
6. Open the `DetailsView` file and add the following at the top of the block of the `DetailsView` composable:

```
val lifecycleOwner = LocalLifecycleOwner.current
DisposableEffect(key1 = lifecycleOwner) {
 val observer = LifecycleEventObserver { _, event ->
 if (event == Lifecycle.Event.ON_PAUSE) {
 Log.d("DetailsView", "DetailsView paused")
 } else if (event == Lifecycle.Event.ON_RESUME) {
 Log.d(
 "DetailsView",
 "DetailsView resumed"
)
 }
 }
 lifecycleOwner.lifecycle.addObserver(observer)

 onDispose {
 lifecycleOwner.lifecycle
```

```
 .removeObserver(observer)
 }
}
```

This adds a `DisposableEffect` composable on the `lifecycleOwner` instance and adds an observer to the `lifecycle` events. The `onDispose` function will then remove the observer when the composable leaves the composition.

7. Run the application. Do various `lifecycle` changes such as going to the home screen, switching to the app, and then returning to the application. Check the logs as you do and you should see "DetailsView paused" and "DetailsView resumed" values in the logs.

In this exercise, you added different Jetpack Compose side-effects to your Android application.

You have learned side-effects such as `LaunchedEffect`, `DisposableEffect`, and `SideEffect` in Jetpack Compose. In the next section, you will explore how to create animations with Jetpack Compose.

## Creating animations using Jetpack Compose

Animations can make your apps livelier and improve the user experience. They can entertain users while content is being fetched or processed. It can guide users on what to do next and can guide them through what steps they can take. Beautiful animations when the app encounters an error can help prevent users from getting angry about what has happened.

Jetpack Compose has built-in APIs that you can use to make animations for your applications.

In the following sections, you will explore some of these animations.

### Animating a single value

To animate an individual state change, such as an integer or a color of a composable, you can use various `animate*AsState` APIs. You will just need to pass the `targetValue` instance that, when changed, will run the animation.

For example, to animate color state changes in a composable, you can use something like the following:

```
val background by animateColorAsState(
 if (darkTheme) Color.Black else Color.White
)
```

This will animate the background change from black to white or from white to black when the `darkTheme` value changes.

The following are the available `animate*AsState` functions you can use to animate value changes in your composable functions:

- `animateDpAsState()`
- `animateFloatAsState()`
- `animateIntAsState()`
- `animateIntSizeAsState`
- `animateOffsetAsState()`
- `animateRectAsState()`
- `animateSizeAsState()`

You can also use the `animateValueAsState()` function to add an animation for a custom data type for your composable.

## Animating the appearance or disappearance of elements

To show or hide UI elements with a bit of transition, you can replace the `if` condition that displays the content with `AnimatedVisibility` and pass the parameter that determines whether the content would be displayed or not. For example, to animate the showing and hiding of a `Column` composable when the value of `shouldBeDisplayed` is `true`, you can use the following code:

```
AnimatedVisibility(shouldBeDisplayed) {
 Column {
 ...
 }
}
```

This will make the appearance or disappearance of the content inside the block (`Column` and its children) smoother. You can also customize the entrance or closing transitions, by specifying the `EnterTransition` transition using the `enter` parameter and the `ExitTransition` transition with the `exit` parameter respectively:

```
AnimatedVisibility(
 shouldBeDisplayed,
 enter = fadeIn(),
 exit = fadeOut()
) {
 Column {
 ...
 }
}
```

This will add a `fadeIn` entrance transition as the content becomes visible and a `fadeOut` exit transition as the content becomes hidden.

For `EnterTransition`, you can use the following values:

- `fadeIn`
- `scaleIn`
- `slideIn`
- `slideInHorizontally`
- `slideInVertically`
- `expandIn`
- `expandHorizontally`
- `expandVertically`

`ExitTransition` has the following transitions that you can use:

- `fadeOut`
- `scaleOut`
- `slideOut`
- `slideOutHorizontally`
- `slideOutVertically`
- `shrinkOut`
- `shrinkHorizontally`
- `shrinkVertically`

The enter or exit transitions can also be combined by using the `+` operator to run multiple transitions at the same time, as shown in the following example:

```
AnimatedVisibility(
 shouldBeDisplayed,
 enter = fadeIn() + expandIn(),
 exit = shrinkOut() + fadeOut()
) {
 Column {
 ...
 }
}
```

This will make the appearance animation of the content a combination of fading in while expanding in. Its animation for disappearance, meanwhile, will be a mix of shrinking out and fading out animations.

You can also use your own enter and exit transitions for `AnimatedVisibility`, if you want to customize the transition.

## Animating multiple values

If you want to change several values at the same time during an animation, you can use `TransitionAPI`. You can create a transition by using `updateTransition()` and passing in `targetState`. Then you can use the animations using `animate*`. The following are the transition. `animate*` extension functions that you can use for each animation:

- `animateColor`
- `animateDp`
- `animateFloat`
- `animateInt`
- `animateIntOffset`
- `animateIntSize`
- `animateOffset`
- `animateRect`
- `animateSize`

You can also use the `animateValue` function to add a transition animation for a custom data type.

For example, you can create a `Transition` instance based on the `isLoading` state by using something like the following:

```
val transition = updateTransition(targetState = isLoading)
```

This will create the transition instance. You can then start the different animations on the transition, for example, for color and Dp changes, with the following:

```
val color by transition.animateColor { isLoading ->
 if (isLoading) MaterialTheme.colorScheme.primary else
 MaterialTheme.colorScheme.secondary
```

```
 ...
}

val alpha by transition.animateFloat {
 if (isLoading) 1f else 0f
}
```

This will create an animation with the `MaterialTheme.colorScheme.primary` color and alpha of 1f when `isLoading` is true and `MaterialTheme.colorScheme.secondary` color and 0f alpha when `isLoading` is false.

You can pass an optional label value as well to the `updateTransition` and `animate*` functions. This can help you identify various animations in Android Studio's **Animation Preview**.

## Animating size changes of elements

You can use the `animateContentSize` modifier to animate size changes of composable functions. For example, when showing more or less of a `Text` composable, you can add animation by adding `Modifier.animateContentSize` before any size modifiers, as shown in the following code block:

```
var showMore by remember { mutableStateOf(false) }
...
Text(
 text = overview,
 modifier = Modifier
 .animateContentSize()
 .height(48.dp)
 .clickable {
 showMore = showMore.not()
 },
 maxLines = if (showMore) Int.MAX_VALUE else 1
)
```

This will add an animation when the size of the `Text` composable is changed from one line to multiple lines.



## Animating changes between composables

To animate changes between composables based on the target state, you can use `AnimatedContent`, passing in `targetState`. The following example animates the content when the `question` value changes:

```
AnimatedContent(question) { questionType ->
 when(questionType) {
 MultipleChoiceType -> MultipleChoiceContent()
 CheckBoxType -> CheckBoxContent()
 ...
 }
}
```

You can customize the animation by passing `transitionSpec` with the entrance and exit transitions.

For a simple switch between two composable functions, for example, when switching between screens or themes, you can use `CrossFade()`. The following sample shows how you can use it in your composables:

```
var counter by remember { mutableStateOf(0) }
...
Crossfade(counter) { counter ->
 when {
 counter % 2 == 0 -> EvenView()
 else -> OddView()
 }
}
```

This will fade the animations between `EvenView` and `OddView` depending on the value of the `counter` as it changes. You can add an `animationSpec` value to customize the `CrossFade` animation.

For both `AnimatedContent` and `CrossFade`, adding a `label` property will make it easier to debug the animation using Android Studio.

## Animating values indefinitely

You can create a repeating animation using `rememberInfiniteTransition`. You can then add the transition animation with `InfiniteTransition.animateColor` for color animation,

`InfiniteTransition.animateFloat` for animating values of float type, or `InfiniteTransition.animateValue` for animating custom data types:

```
val infiniteTransition = rememberInfiniteTransition()

val alpha by infiniteTransition.animateFloat(
 initialValue = 0f,
 targetValue = 1f,
 animationSpec = ...
)
```

The previous example will start an animation from the initial alpha value of `0f` and the target value of `1f`, then restart the animation.

## Other compose animations

There are other compose animations that you can use in your applications. Here are some of them.

**Animatable** is a Coroutine-based animation for animating a single value change. It can also be used outside of composable functions. **Animatable** animates its content when the value is changed when calling `animateTo`:

```
val color = remember { Animatable(Color.Black) }
LaunchedEffect(success) {
 color.animateTo(
 if (success) Color.Green else Color.Red
)
}
```

You can then use this in a composable. The initial value will be `Gray` and every time the `success` value changes, it will animate to `Color.Green` or `Color.Red` depending on the value.

**Animation** is the lowest level of animation that is used to manually control the timing of the animation. **Animation** has two subtypes: **TargetBasedAnimation** and **DecayAnimation**. **TargetBasedAnimation** is used to control the animation based on the target value while **DecayAnimation** uses the starting conditions provided with `initialVelocity` and `initialValue` to slow down to the target state.

## Customizing animations

Most of the animation APIs are customizable by using `AnimationSpec`, which specifies how the animation value should transform from the start to the target values. You can use the following `AnimationSpec` types for your animations:

- `tween`
- `keyframes`
- `spring`
- `repeatable`
- `infiniteRepeatable`
- `snap`

## Debugging animations

In Android Studio, you can debug animations from **Animation Preview**. If your preview has inspectable animations, you will be able to click the **Start Animation Preview** icon at the top of the preview, as shown in *Figure 15.1*:

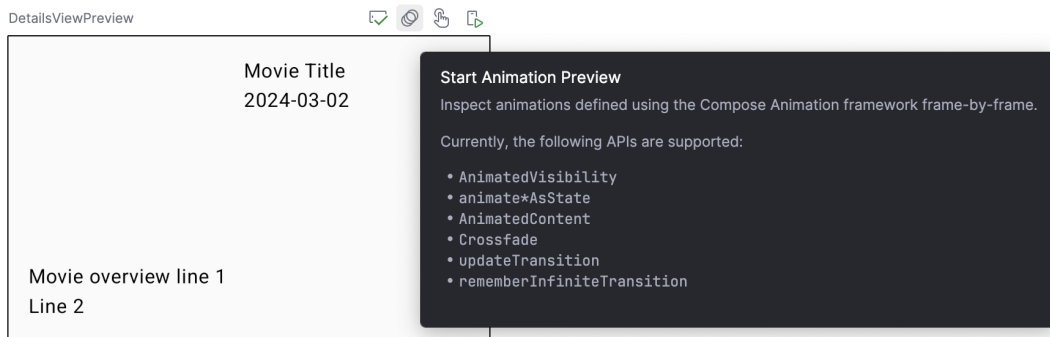


*Figure 15.1 – The Android Studio Preview window where you can click Start Animation Preview*

With **Animation Preview** in Android Studio, you should be able to debug your animations. You can inspect, pause, forward, and slow down the animation. If a composable has multiple animations, you can run freeze an animation (using the first icon at the bottom of the animation as shown in *Figure 15.2*).

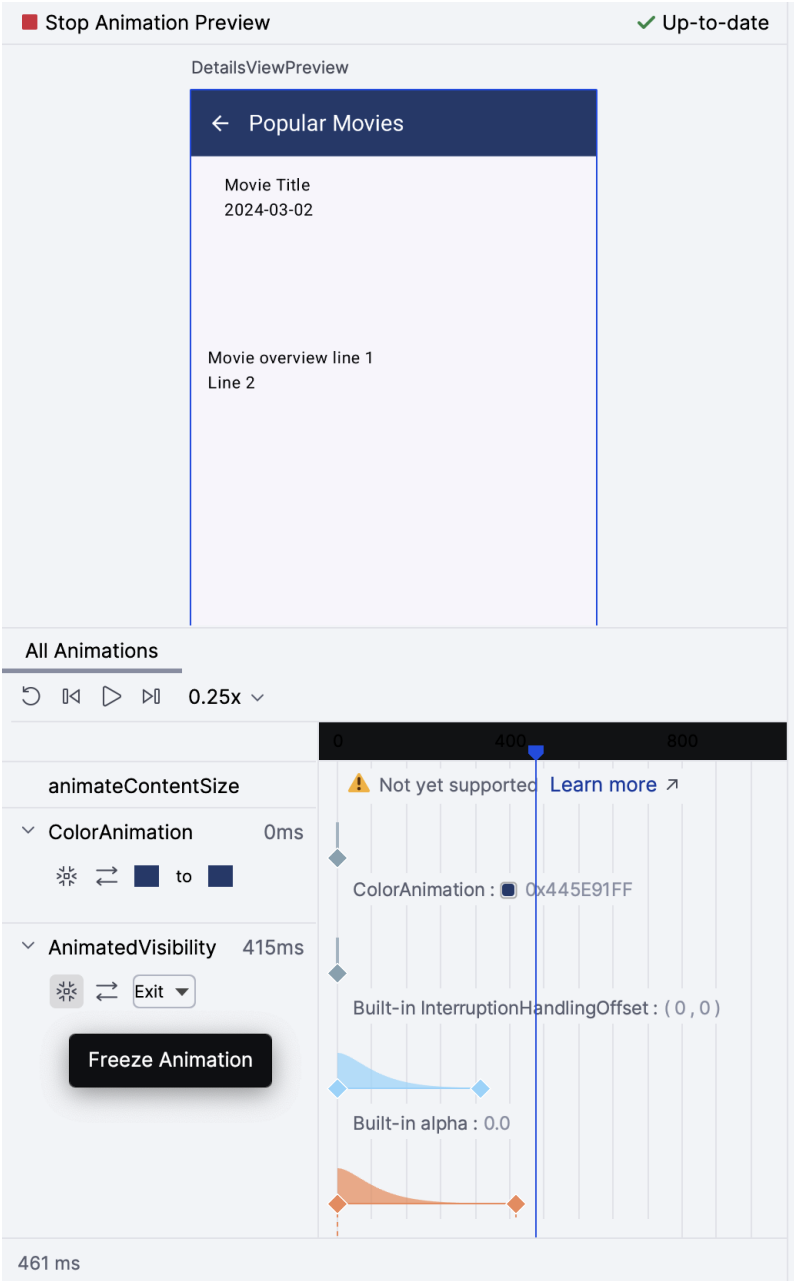


Figure 15.2 – The Android Studio Preview Window in action

Adding the `label` property in your composables would help you identify your animations too in **Animation Preview**.

Let's try adding Jetpack Compose animations by adding some animations to your *Popular Movies* application.

## Exercise 15.03 – adding animations with Jetpack Compose

In this exercise, we will be updating our *Popular Movies* application with a `MotionLayout` animation. In the output screen, the image above the tip text will move down when tapped and will go back to its original position when tapped again:

1. Open the *Popular Movies* project in Android Studio.
2. Open the `DetailsScreen` file. Before the overview text, add the following line of code:

```
var showFullOverview by remember {
 mutableStateOf(false)
}
```

This will be used in the overview text to show two or more lines of text.

3. Add the following code to change the number of lines of the overview, depending on the value of `showFullOverview`, which will be toggled when the text itself is clicked:

```
Text(
 text = overview,
 overflow = TextOverflow.Ellipsis,
 modifier = Modifier
 .padding(top = 8.dp)
 .fillMaxWidth()
 .clickable {
 showFullOverview = showFullOverview.not()
 },
 maxLines = if (showFullOverview)
 Int.MAX_VALUE
 else
 2
)
```

4. Run the application and click on the overview text multiple times to show more lines of text or just two lines of overview. Your details screen should look like the following:

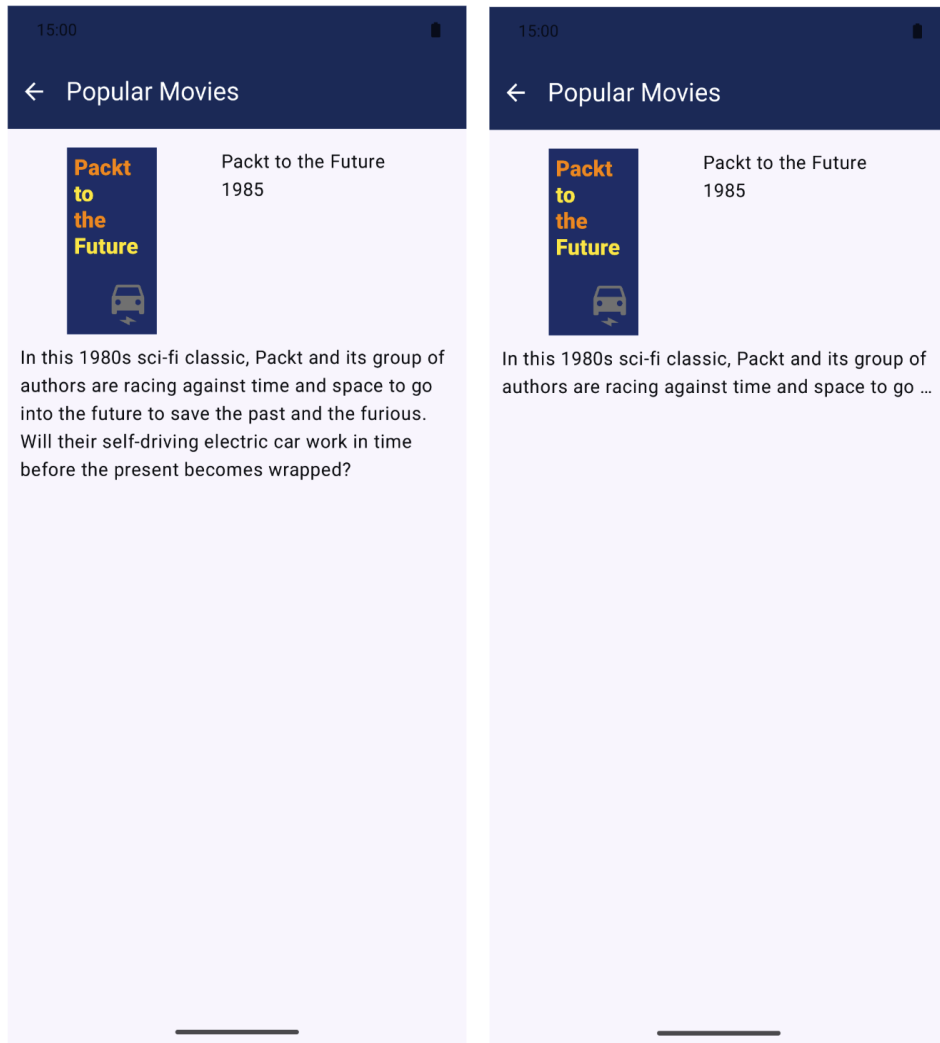


Figure 15.3 – The details screen showing the overview text with 2 lines (right) and multiple lines (left)

5. Add `animateContentSize` on the overview text modifier to add animation when the number of lines changes:

```
Text(
 text = overview,
 overflow = TextOverflow.Ellipsis,
 modifier = Modifier
 .animateContentSize()
```

```

 .padding(top = 8.dp)
 .fillMaxWidth()
 .clickable {
 showFullOverview = showFullOverview.not()
 },
 maxLines = if (showFullOverview)
 Int.MAX_VALUE
 else
 2
)

```

6. Run the application again. You should see an animation as the number of lines of the overview text changes.
7. Your next animation will be to show or hide the poster as you click on the title text. First, add the following at the top of the outer Column composable and before the AsyncImage composable for the poster:

```

var showPoster by remember { mutableStateOf(true) }

```

This will be used to show or hide the poster. It is true by default so the poster should be displayed initially.

8. Add a click event on the title text to toggle the value of showPoster:

```

Text(
 text = title,
 overflow = TextOverflow.Ellipsis,
 modifier = Modifier
 .fillMaxWidth()
 .clickable {
 showPoster = showPoster.not()
 }
)

```

9. Wrap the AsyncImage composable for the image in AnimatedVisibility to add animation when the showPoster value is changed:

```

AnimatedVisibility(visible = showPoster) {
 AsyncImage(
 model = image,
 contentDescription = title,
)
}

```

```
 contentScale = ContentScale.Fit,
 modifier = Modifier.size(160.dp)
)
}
```

10. Run the application. Click on the title text to show or hide the image. Your app should look like the following when the image is hidden:

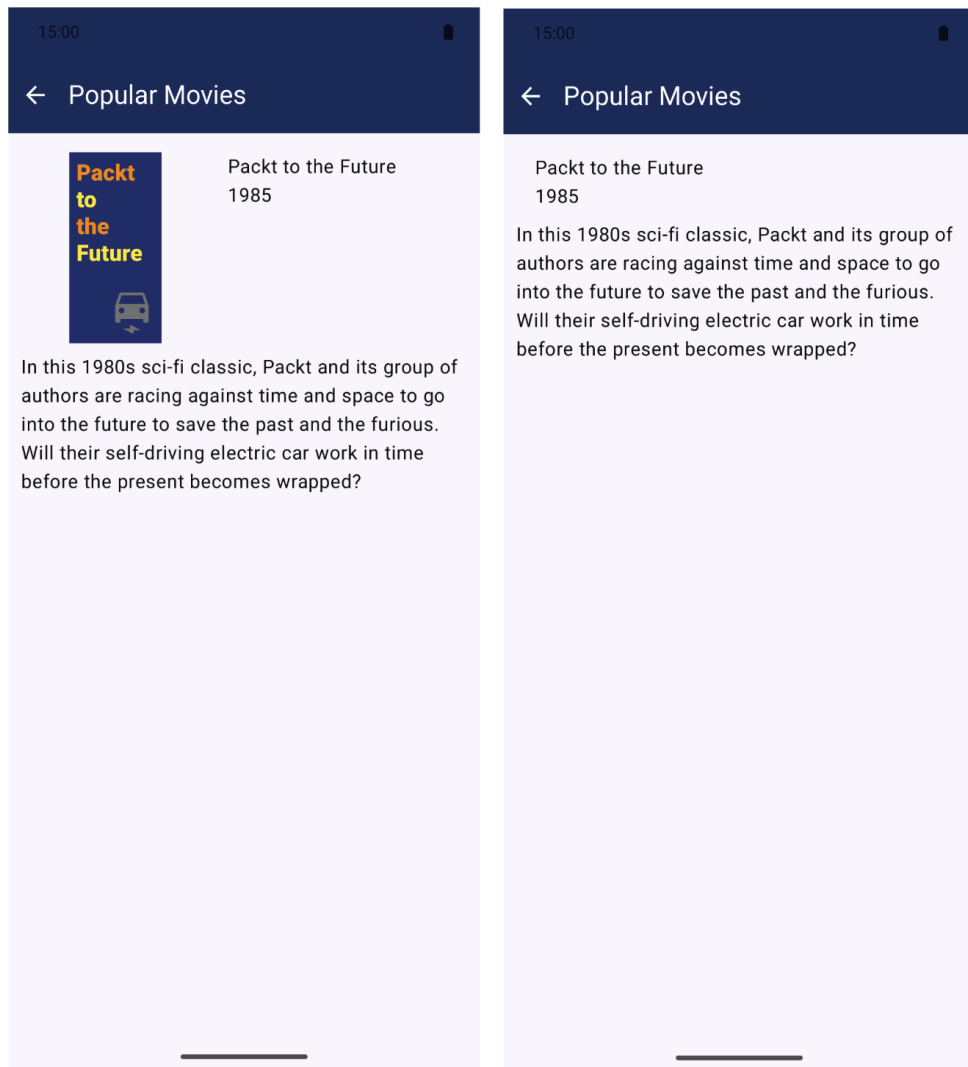


Figure 15.4 – The details screen when the poster image is displayed (left) and when the poster image is not shown (right)



In this exercise, you have added various animations using Jetpack Compose APIs to our Android application.

Let's test everything you've learned in this chapter by doing an activity.

## Activity 15.01 – adding Animations to the TV Guide app

In *Chapter 14, Architecture Patterns*, you have improved the TV Guide app, which is an app that can display a list of TV shows that are on the air. You can use the TV Guide app you have updated in *Activity 14.01 – revisiting the TV Guide app* or download it from the GitHub repository (<https://packt.link/C1bVh>).

Update the app by adding animations in the text and image elements in the Android application.

The steps to complete them are as follows:

1. Open the TV Guide app in Android Studio.
2. Open the `DetailsScreen` file and add one or more animations on the text or image in one or more of the child composable functions.



The solution to this activity can be found at <https://packt.link/ZGSDg>.

## Summary

This chapter covered advanced Jetpack Compose topics including `CompositionLocal` and side-effects such as `LaunchedEffect`, `DisposableEffect`, and `SideEffect`.

You also explored animations on Jetpack Compose. Animations can improve the usability of our application and make it stand out compared to other apps. You learned about the different animation APIs that you can use for your applications, such as animating a single value, the appearance or disappearance of elements, multiple values, size changes of elements, changes between composables, and values indefinitely. You also learned about customizing animations and debugging them on Android Studio.

In the next chapter, you'll learn about launching your apps on the Google Play Store. You'll explore how you can create a Google Play Developer account and prepare your apps for release, as well as how you can publish them later so that users can download and use them.

## Unlock this book's exclusive benefits now

Scan this QR code or go to [packtpub.com/unlock](https://packtpub.com/unlock), then search this book by name.

*Note: Keep your purchase invoice ready before you start.*

UNLOCK NOW





# 16

## Launching Your App on Google Play

After you develop Android apps, they will only be available on your devices and emulators. You must make them available to everyone so they can download them. In turn, you will acquire users, and you can earn from them. The official marketplace for Android apps is Google Play. With Google Play, the apps and games you release will be available to over 2 billion active Android devices globally. There are also other marketplaces where you can publish your apps, but they are beyond the scope of this book.

This chapter will introduce you to the Google Play Console, release channels, and the entire release process. It covers creating a Google Play developer account, setting up the store entry for our developed app, and creating a key store (including coverage of the importance of passwords and where to store files). You'll also learn about app bundles, looking at how to generate the app's AAB file. Later in the chapter, you'll set up release paths, open beta, and closed alpha, and finally, you'll upload our app to the store and download it on a device.

By the end of this chapter, you will be able to create your own Google Play developer account, prepare your signed app bundle for publishing, and publish your first application on Google Play.

We will cover the following topics in this chapter:

- Preparing your apps for release
- Creating a developer account
- Uploading an app to Google Play
- Managing app releases

## Preparing your apps for release

Android Studio normally signs your build using a debug key. This debug build allows you to build and test your app quickly. To publish your app on Google Play, you must create a release build signed with your own key. This release build will not be debuggable and can be optimized for size.

The release build must also have the correct version information. Otherwise, you won't be able to publish a new app or update an already-published app.

Let's start with adding versions to your app.

## Versioning apps

The version of your app is important for the following reasons:

- Users can see the version they have downloaded. They can use this when checking whether there's an update or whether there are known issues when reporting bugs/problems with the app.
- The device and Google Play use the version value to determine whether an app can or should be updated.
- Developers can use this value to add feature support to specific versions. They can also warn or force users to upgrade to the latest version to get important fixes on bugs or security issues.

An Android app has two version values: `versionCode` and `versionName`. Now, `versionCode` is an integer that is used by developers, Google Play, and the Android system, while `versionName` is the string that users see on the Google Play page for your app.

The initial release of an app should have a `versionCode` value of 1, and you should increase it for each new release.

The value of `versionName` can be in `x.y` format (where `x` is the major version and `y` is the minor version). You can also use semantic versioning, as in `x.y.z`, by adding the patch version with `z`.

To learn more about semantic versioning, refer to <https://semver.org>.

In the module's `build.gradle.kts` files, `versionCode` and `versionName` are automatically generated when you create a new project in Android Studio. They are in the `defaultConfig` block under the `android` block. An example `build.gradle.kts` file shows these values:

```
android {
 namespace = "com.example.app"
 compileSdk = 36
```

```
defaultConfig {
 applicationId = "com.example.app"
 minSdk = 24
 targetSdk = 36
 versionCode = 1
 versionName = "1.0"
 ...
}
...
}
```

💡 **Quick tip:** Enhance your coding experience with the **AI Code Explainer** and **Quick Copy** features. Open this book in the next-gen Packt Reader. Click the **Copy** button (1) to quickly copy code into your coding environment, or click the **Explain** button (2) to get the AI assistant to explain a block of code to you.

```
function calculate(a, b) {
 return {sum: a + b};
};
```

Copy

Explain

1

2



🔒 **The next-gen Packt Reader** is included for free with the purchase of this book. Scan the QR code OR go to [packtpub.com/unlock](https://packtpub.com/unlock), then use the search bar to find this book by name. Double-check the edition shown to make sure you get the right one.



When publishing updates, the new package being released must have a higher `versionCode` value because users cannot downgrade their apps and can only download new versions.

After ensuring that the app version is correct, the next step in the release process is to get a keystore to sign the app. This will be discussed in the next section.

## Creating a keystore

Android apps, when run, are automatically signed by a debug key. However, before it can be published on the Google Play Store, an app must be signed with a release key. To do so, you must have a keystore. If you don't have one yet, you can create one in Android Studio.

### Exercise 16.01 – creating a keystore in Android Studio

In this exercise, you'll use Android Studio to make a keystore that can be used to sign Android apps. Follow these steps to complete this exercise:

1. Open a project in Android Studio.
2. Go to the **Build** menu and then click on **Generate Signed App Bundle or APK**:

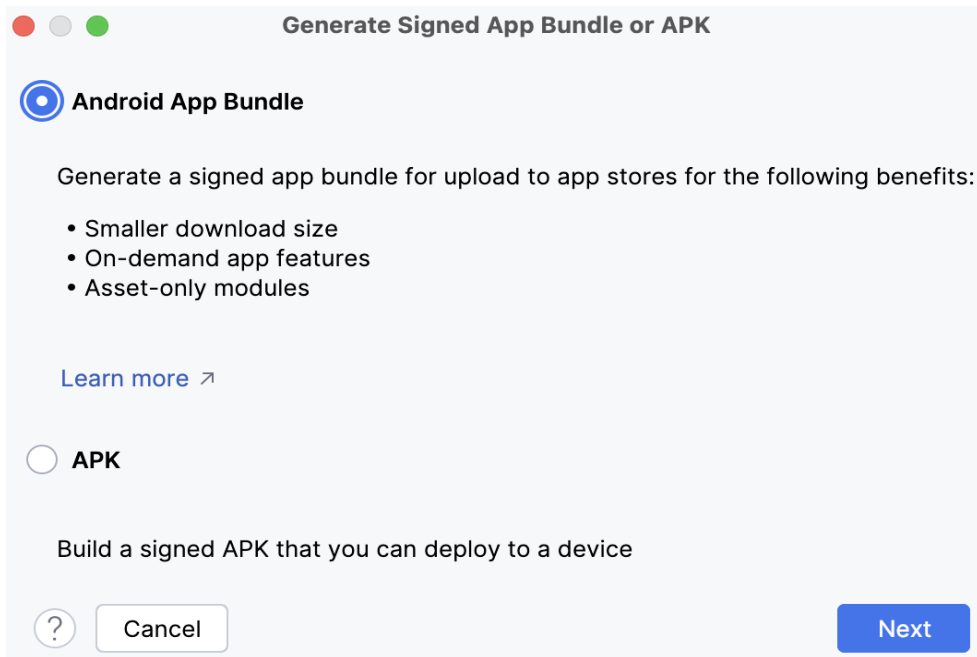
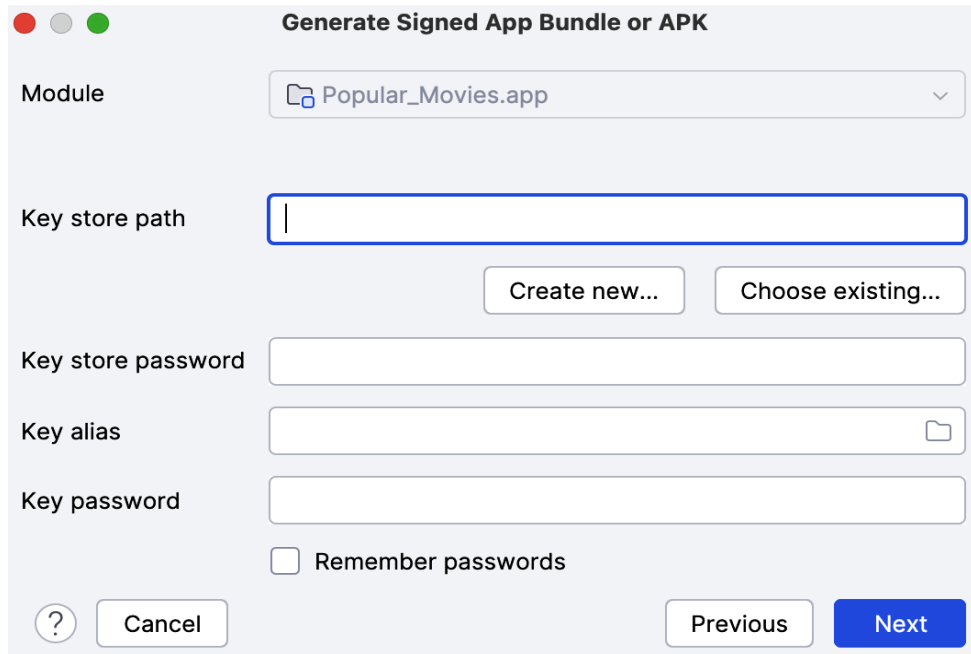


Figure 16.1 – The Generate Signed App Bundle or APK dialog

**APK** is the file format by which users may install your app manually. **Android App Bundle** is a new file publishing format that allows Google Play to distribute specific, smaller APKs to devices, so developers don't need to release and manage multiple APKs to support different devices.

3. Make sure either **APK** or **Android App Bundle** is selected, and then click the **Next** button. Here, you can choose an existing keystore or create a new one:

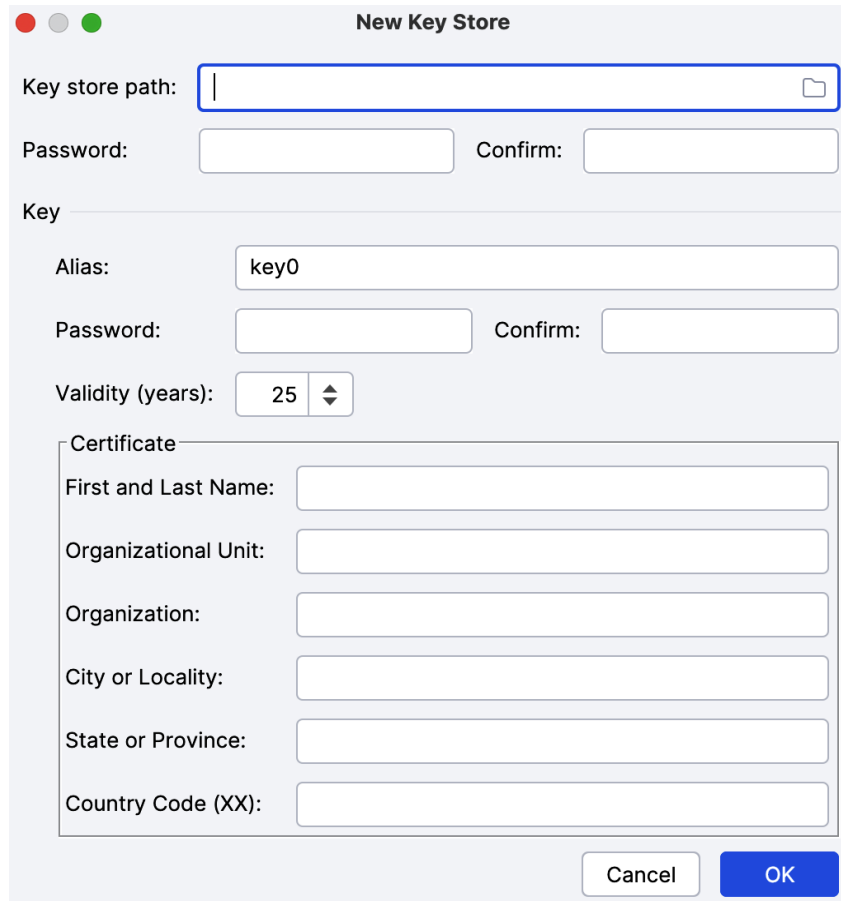


The screenshot shows the "Generate Signed App Bundle or APK" dialog box. At the top, the title is "Generate Signed App Bundle or APK". Below the title, there is a "Module" dropdown menu showing "Popular\_Movies.app". Underneath, the "Key store path" field is empty, with "Create new..." and "Choose existing..." buttons to its right. Below these are three text input fields: "Key store password", "Key alias", and "Key password". A checkbox labeled "Remember passwords" is located below the "Key password" field. At the bottom left, there is a help icon (?) and a "Cancel" button. At the bottom right, there are "Previous" and "Next" buttons. The "Next" button is highlighted in blue.

*Figure 16.2 – The Generate Signed App Bundle or APK dialog after selecting APK and pressing the Next button*

4. Click the **Create new...** button. The **New Key Store** dialog will then appear:





**New Key Store**

Key store path:

Password:  Confirm:

Key

Alias:

Password:  Confirm:

Validity (years):

Certificate

First and Last Name:

Organizational Unit:

Organization:

City or Locality:

State or Province:

Country Code (XX):

Cancel OK

Figure 16.3 – The New Key Store dialog

5. In the **Key store path** field, choose the location where the keystore file will be saved. You can click on the folder icon on the right to select your folder and type the filename. The value will be similar to `/Users/packt/downloads/keystore.keystore`.
6. Provide the password in both the **Password** and **Confirm** fields.
7. In the **Certificate** section under **Key**, input values into the **First and Last Name**, **Organizational Unit**, **Organization**, **City or Locality**, **State or Province**, and **Country Code** fields. Only one of these is required, but it's good to provide all the information.
8. Click the **OK** button. If there is no error, the keystore will be created in the path you provided, and you will be back in the **Generate Signed App Bundle or APK** dialog with the keystore values so you can continue generating the APK or app bundle. You could close the dialog if you only wanted to create a keystore.

In this exercise, you have created your own keystore, which you can use to sign applications to be published to Google Play.

You can also use the command line to generate a keystore if you prefer to use that. The `keytool` command is available in the **Java Development Kit (JDK)**. The command is as follows:

```
keytool -genkey -v -keystore my-key.jks -keyalg RSA -
keysize
2048 -validity 9125 -alias key-alias
```

This command creates a 2,048-bit **Rivest–Shamir–Adleman (RSA)** keystore in the current working directory, which is valid for 9,125 days (25 years), with a `my-key.jks` filename and a `key-alias` alias. You can change the validity, filename, and alias to your preferred values. The command line will prompt you to input the keystore password, then prompt you to enter it again to confirm.

It will then ask you for the first and last name, organizational unit, organization name, city or locality, state or province, and country code, one at a time. Only one of these is required; you can press the *Enter* key if you want to leave something blank. It is good practice, though, to provide all the information.

After the country code prompt, you will be asked to verify the input provided. You can type *yes* to confirm. You will then be asked to provide the password for the key alias. If you want it to be the same as the keystore password, you can press *Enter*. The keystore will then be generated.

Now that you have a keystore for signing your apps, you need to know how you can keep it safe. You'll learn about that in the next section.

## Storing the keystore and passwords

You need to keep the keystore and passwords in a safe and secure place because if you lose the keystore and/or its credentials, you will no longer be able to release updates for your apps. If a hacker gains access to these, they may be able to update your apps without your consent.

You can store the keystore in your **continuous integration (CI)**/build server or on a secure server.

Keeping the credentials secure is a bit tricky, as you will need them later when signing releases for app updates. One way you can do this is by including this information in your project's `app/build.gradle.kts` file.

In the `android` block, you can have `signingConfigs`, which references the keystore file, its password, and the key's alias and password:

```
android {
 ...
 signingConfigs {
 create("release") {
 storeFile = file("keystore-file")
 storePassword = "keystore-password"
 keyAlias = "key-alias"
 keyPassword = "key-password"
 }
 }
 ...
}
```

Under the `buildTypes` `release` block in the project's `build.gradle.kts` file, you can specify the release config in the `signingConfigs` block:

```
buildTypes {
 getByName("release") {
 ...
 signingConfig = signingConfigs.getByName("release")
 }
 ...
}
```

Storing the signing configs in the `build.gradle.kts` file is not that secure, as someone who has access to the project or the repository can compromise the app.

You can store these credentials in environment variables to make them more secure. With this approach, even if malicious people get access to your code, the app updates will still be safe as the signing configurations are not stored in your code but on the system. An environment variable is a key-value pair that is set outside your **integrated development environment (IDE)** or project, such as on your own computer or on a build server.

To use environment variables for keystore configurations in Gradle, you can create them for the store file path, store password, key alias, and key password. For example, you can use the `KEYSTORE_FILE`, `KEYSTORE_PASSWORD`, `KEY_ALIAS`, and `KEY_PASSWORD` environment variables.

On macOS and Linux, you can set an environment variable by using the following command:

```
export KEYSTORE_PASSWORD=securepassword
```

If you're using Windows, it can be done with this command:

```
set KEYSTORE_PASSWORD=securepassword
```

This command will create a `KEYSTORE_PASSWORD` environment variable with `securepassword` as the value. In the `app/build.gradle.kts` file, you can then use the values from the environment variables:

```
storeFile = System.getenv("KEYSTORE_FILE")
storePassword = System.getenv("KEYSTORE_PASSWORD")
keyAlias = System.getenv("KEY_ALIAS")
keyPassword = System.getenv("KEY_PASSWORD")
```

Your keystore will be used to sign your app for release so you can publish it on Google Play. We'll discuss that in the next section.

## Signing your apps for release

When you run an application on an emulator or an actual device, Android Studio automatically signs it with the debug keystore. To publish it on Google Play, you must sign the app bundle with your own key, using a keystore you made in Android Studio or from the command line.

Suppose you have added the signing config for the release build in your `build.gradle.kts` file; you can automatically build a signed APK or app bundle by selecting the release build in the **Build Variants** window. You then need to go to the **Build** menu, click on the **Build Bundle(s)** item, and then select either **Build APK(s)** or **Build Bundle(s)**. The APK or app bundle will be generated in the `app/build/output` directory of your project.

## Android App Bundle

The traditional way of releasing Android apps is through an APK or an application package. This APK file is the one downloaded to users' devices when they install your app. This one big file contains all the strings, images, and other resources for all device configurations.

As you support more device types and more countries, this APK file will grow in size. The APK that users download will contain things that are not really needed for their devices. This will be an issue for you as users with low storage might not have enough space to install your app. Users with expensive data plans or slow internet connections might avoid downloading the app if it's too big. They might also uninstall your app to save storage space.

Some developers have built and published multiple APKs to avoid these issues. However, it's a complicated and inefficient solution, especially when you target different screen densities, **central processing unit (CPU)** architectures, and languages. In addition, that would be too many APK files to maintain per release.

Android App Bundle is a new way of packaging apps for publishing. You just generate a single app bundle file (using Android Studio 3.2 and upward) and upload it to Google Play. Google Play will automatically generate the base APK file and the APK files for each device configuration, CPU architecture, and language. Users who install your app will only download the necessary APKs for their devices. This will be smaller in size compared to a universal APK.

This will work for devices running on Android 5.0 Lollipop and upward; for those below it, the APK files that will be generated are only for device configuration and CPU architecture. All the languages and other resources will be included in each APK file.

## Exercise 16.02 – creating a signed app bundle

In this exercise, you will create a signed app bundle for an Android project using Android Studio:

1. Open a project in Android Studio.
2. Go to the **Build** menu, then click on the **Generate Signed App Bundle or APK** menu item:

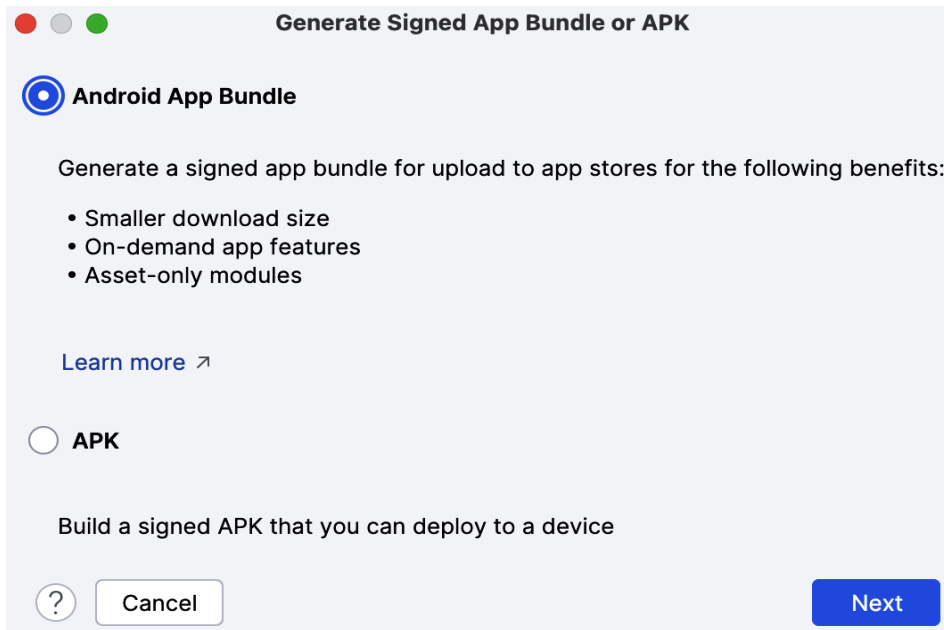
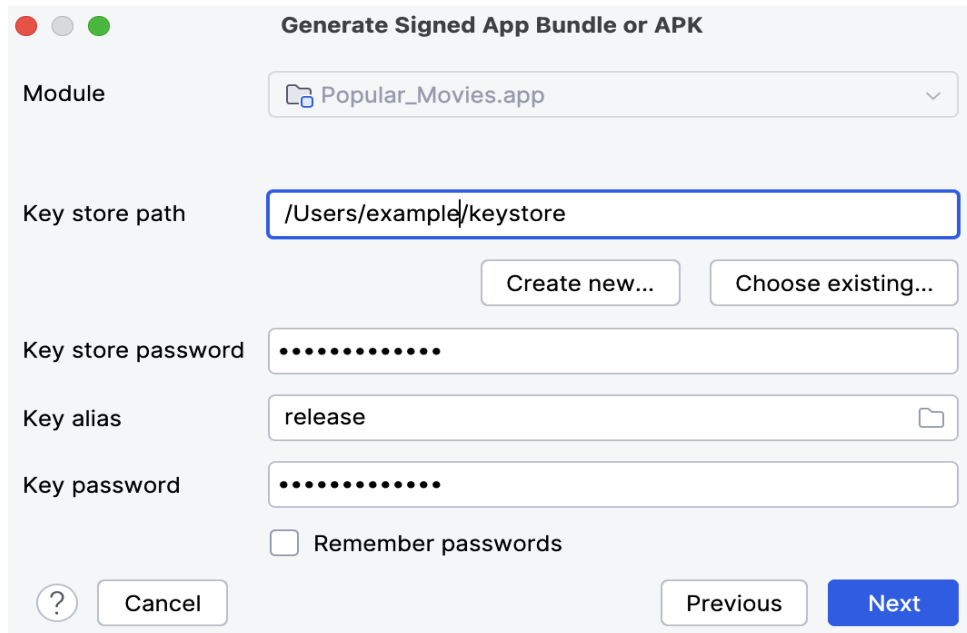


Figure 16.4 – The Generate Signed App Bundle or APK dialog

3. Select **Android App Bundle**, then click the **Next** button:



**Generate Signed App Bundle or APK**

Module: Popular\_Movies.app

Key store path: /Users/example/keystore

Create new... Choose existing...

Key store password: .....

Key alias: release

Key password: .....

☐ Remember passwords

? Cancel Previous Next

Figure 16.5 – The Generate Signed App Bundle or APK dialog after clicking the Next button

4. Choose the keystore you made in *Exercise 16.01 – creating a keystore in Android Studio*.
5. Provide the password that you set for the keystore you created in the **Key store password** field.
6. In the **Key alias** field, click the icon on the right side and select the key alias.
7. Provide the alias password that you set for the keystore you created in the **Key password** field.
8. Click the **Next** button.
9. Choose the destination folder to generate the signed app bundle.
10. In the **Build Variants** field, make sure the **release** variant is selected:

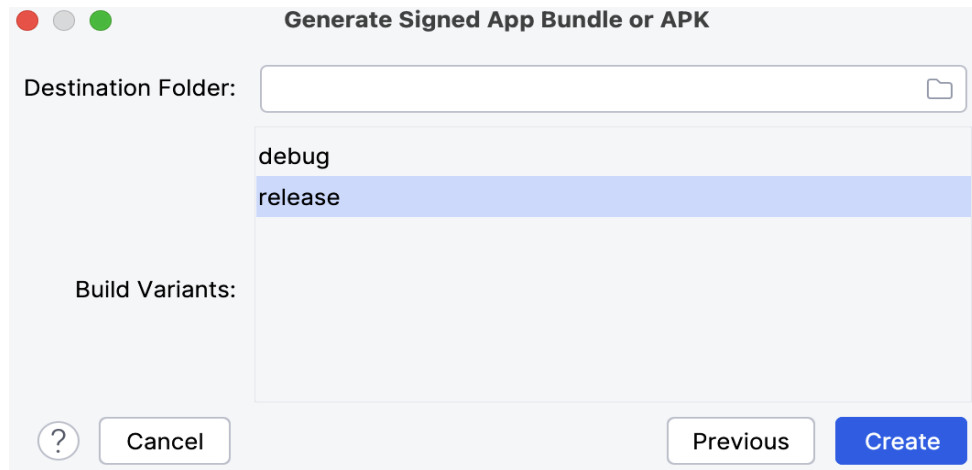


Figure 16.6 – Choosing the release build in the Generate Signed App Bundle or APK dialog

11. Click the **Finish** button. Android Studio will build the signed app bundle. An IDE notification will inform you that the signed app bundle was generated. You can click on **locate** to go to the directory where the signed app bundle file is:

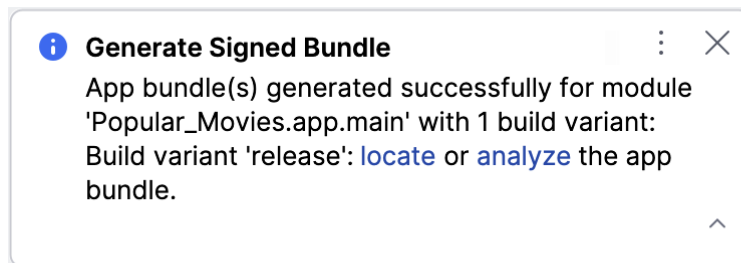


Figure 16.7 – Pop-up notification that the signed app bundle was generated

In this exercise, you have made a signed app bundle that you can now publish on Google Play.

To be able to publish your app to the Google Play Store with the Android App Bundle format, you will first need to opt in to Google Play's app signing feature. We will discuss Google Play app signing in the next section.

## App signing by Google Play

Google Play provides a service called **app signing** that allows Google to manage and protect your app-signing keys and automatically re-sign your app for users.

With the Google Play app-signing service, you can let Google generate the signing key or upload your own. You can also create a different upload key for additional security. You can sign the app with the upload key and publish the app on the Google Play Console.

Google will check the upload key, remove it, and use the app-signing key to re-sign the app for distribution to users. When app signing is enabled for the app, the upload key can be reset. If you lose the upload key or feel that it is already compromised, you can simply contact Google Play developer support, verify your identity, and get a new upload key.

It is easy to opt into app signing when publishing a new app. In the Google Play Console (<https://packt.link/tj3wL>), you can go to **Release Management | App Releases** and select **Continue** in the **Let Google manage and protect your app signing key** section. The key you originally used to sign the app will become the upload key, and Google Play will generate a new app-signing key.

You can also configure existing apps to use app signing. This is available in the **Test and release | Setup | App Signing** section of the app in the Google Play Console. You need to upload your existing app-signing key and generate a new upload key.

Once you enroll in Google Play app signing, you can no longer opt out. Also, if you use third-party services, you must use the app-signing key's certificate. This is available in **Test and release | Setup | App Signing**.

App signing also enables you to upload an app bundle, and Google Play will automatically sign and generate APK files, which users will download when they install your app.

In the next section, you will create a Google Play developer account so you can publish an app's signed APK or app bundle to Google Play.

## Creating a developer account

To publish applications on Google Play, you first need to create a Google Play developer account. Head over to <https://packt.link/bbsqq> and log in with your Google account. If you don't have one, you should create one first.

We recommend using a Google account that you plan to use in the long term instead of a throwaway one. Read the developer distribution agreement and agree to the terms of service.



If your goal is to sell paid apps or add in-app products to your apps/games, you must also create a merchant account. This is not available in all countries, unfortunately. We won't cover this here, but you can read more about it on the registration page or at <https://packt.link/LDncA>.



You must pay a \$25 registration fee to create your Google Play developer account (this is a one-time payment). The fee must be paid using a valid debit/credit card, but some virtual credit cards may work. What you can use varies by location/country.

The final step is to complete the account details, such as the developer's name, email address, website, and phone number. These details, which can be updated later, will form the developer information displayed on your app's store listing.

After completing the registration, you will receive a confirmation email. It may take a few hours (up to 48) for your payment to be processed and your account registered, so be patient. Ideally, you should do this in advance, even if your app is not yet ready, so you can easily publish the app once it's ready for release.

When you have received the confirmation email from Google, you can start publishing apps and games to Google Play.

In the next section, we will discuss uploading apps to Google Play.

## Uploading an app to Google Play

Once you have an app ready for release and a Google Play developer account, you can go to the Google Play Console (<https://packt.link/tj3wL>) to publish the app.



The **user interface (UI)** of the Play Console is subject to change. For updated information, you can check <https://packt.link/tj3wL>.

To upload an app, go to the Google Play Console, click **All Apps**, and then click **Create app**. Provide the name of the application and the default language. In the **App or game** section, set whether it's an app or game. Likewise, in the **Free or paid** section, set whether it's free or paid. Create your store listing, prepare the app release, and roll out the release. We'll have a look at the detailed steps in the following subsections.

## Creating a store listing

The store listing is what users first see when they open your app's page on Google Play. If the app is already published, you can go to **Grow users**, then **Store presence**, and then select **Default store listing**.

## App details

Navigate to the **App details** page. There, you need to fill in the following fields:

- **App name:** Here, you provide your app's name (the maximum number of characters is 50).
- **Short description:** Here, you provide a short text summarizing your app (the maximum number of characters is 80).
- **Full description:** This is the long description for your app. The limit is 4,000 characters, so you can add a lot of relevant information here, such as what its features are and things users need to know.



For the product details, you can add localized versions depending on the languages/countries in which your app will be released.

Your app title and description must not contain copyrighted materials or spam, as this might result in your app being rejected.

## Graphic assets

In this section, provide the following details:

- An icon (a high-resolution icon that is 512 pixels by 512 pixels).
- A feature graphic (1,024 pixels by 500 pixels).
- 2–8 screenshots of the app. If your app supports other form factors (tablet, TV, or Wear OS), you should also add screenshots for each form factor.

You can also add promo graphics and videos if you have any. Your app can be rejected if you use graphics that violate Google Play policy, so ensure that the images you use are your own and don't include copyrighted or inappropriate content.

## Preparing the release

Before preparing your release, ensure that your build is signed with a signature key. If you're publishing an app update, make sure that it has the same package name, is signed with the same key, and has a version code higher than the current one on the Play Store.

You must also make sure you follow the developer policy (to avoid any violations) and make sure your app follows the app quality guidelines. More of these are listed on the launch checklist, which you can see at <https://packt.link/whlcq>.

## App bundle

You can upload an Android App Bundle by going to **Test and release** and then **Latest releases**. This will display a summary of active and draft releases in each track.

There are different tracks where you can release the app:

- **Production**
- **Open testing**
- **Closed testing**
- **Internal testing**

We'll discuss the release tracks in detail in the *Managing app releases* section.

Select the track (**Production** or the specific test track in **Testing**) where you will create the release. To release on a closed testing track, you must also select **Manage track** or create a new track by clicking on **Create track**.

Once done, you can click **Create new release** at the top right of the page. In the **App Bundles** section, you can upload your APK or app bundle.

Make sure that the app bundle or APK file is signed by your release signing key. The Google Play Console will not accept it if it's not properly signed. If you're publishing an update, the version code for the app bundle or APK must be higher than the existing version.

You can also add a release name and release notes. The release name is for the developer's use to track the release and won't be visible to users. By default, the version name of the APK or app bundle uploaded is set as the release name. The release notes form the text that will be shown on the Google Play page and will inform users of what the updates to the app are.

The text for the release notes must be added inside the tags for the language. For example, the opening and closing tags for the default US English language version are `<en-US>` and `</en-US>`. If your app supports multiple languages, each language tag will be displayed in the field for the release notes by default. You can then add the release notes for each language.

If you have already released the app, you can copy the release notes from previous releases and reuse or modify them by clicking the **Copy from a previous release** button and selecting from the list.

When you click the **Save** button, the release will be saved, and you can go back to it later. The **Review release** button will take you to the screen where you can review and roll out the release.

# Rolling out a release

If you’re ready to roll out your release, go to the Google Play Console and select your app. Go to **Test and release** and select your release track. Click the **Releases** tab and then click on the **Edit release** button next to the release:

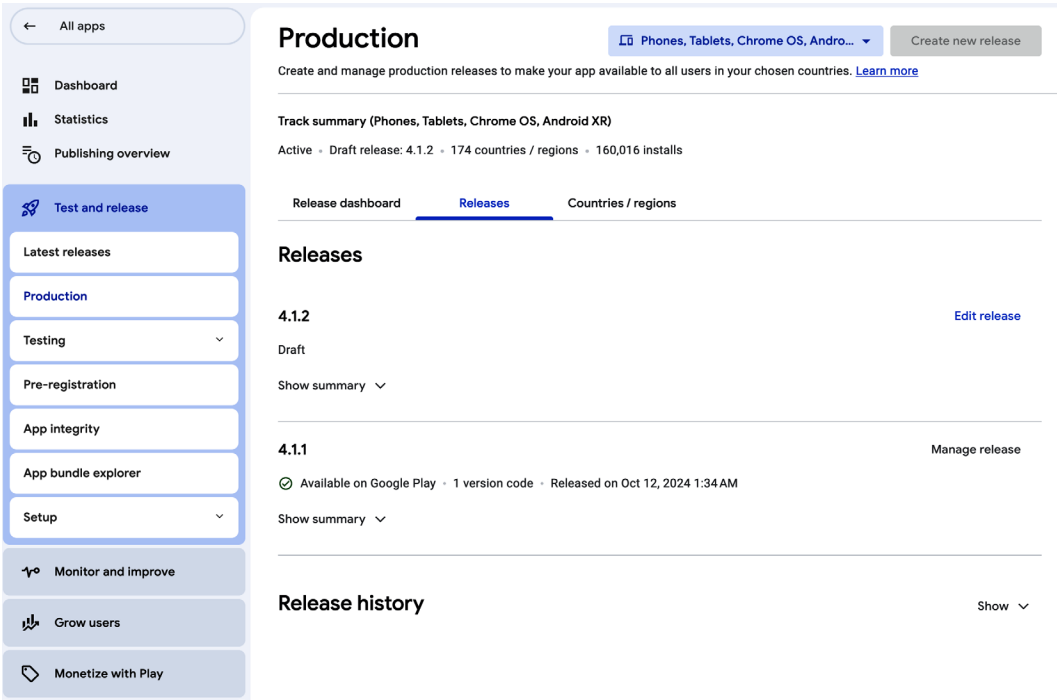


Figure 16.8 – A draft release on the Production track

You can review the APK or app bundle, release names, and release notes. Click the **Review release** button to start the rollout for the release. The Google Play Console will open the **Review and release** screen. Here, you can review the release information and check whether there are warnings or errors.

If you are updating an app, you can also select the rollout percentage when creating another release. Setting it to 100% means it will be available for all your users to download. When you set it to a lower percentage, for example 50%, the release will be available to half of your existing users.

If you’re confident with the release, select the **START ROLLOUT TO PRODUCTION** button (or the similar button for the other tracks) at the bottom of the page. After publishing your app, it will be reviewed, or you might have to manually go to **Publishing Overview** to send the changes for review. Reviews normally take a while (up to seven days, or sometimes longer, especially

for new apps). You can see the status in the top-right corner of the Google Play Console. These statuses include the following:

- **Pending publication:** Your new app is being reviewed
- **Published:** Your app is now available on Google Play
- **Rejected:** Your app wasn't published because of a policy violation
- **Suspended:** Your app violated the Google Play policy and was suspended

If there are issues with your app, you can resolve them and resubmit the app. Your app can be rejected for reasons such as copyright infringement, impersonation, or spam.

Once the app has been published, users can download it. It can take some time before a new app or an app update goes live on Google Play. If you're trying to search for your app on Google Play, it might not be searchable. Make sure you publish it on the production or open track.

In the next section, you will learn about managing app releases on the Google Play Console.

## Managing app releases

You can slowly release your apps on different tracks to test them before publicly rolling them out to users. You can also do managed publishing to make the app available on a certain date instead of automatically publishing it once approved by Google.

The following subsections discuss various ways to manage your app releases on the Google Play Console.

### Release tracks

When creating a release for an app, you can choose between four different tracks:

- **Production** is where everyone can see the app.
- **Open testing** is targeted at wider public testing. The release will be available on Google Play, and anyone can join the beta program and test it.
- **Closed testing** is intended for small groups of users testing pre-release versions.
- **Internal testing** is for the developer/tester builds while the app or feature is being developed or tested.

The internal, closed, and open tracks allow developers to create a special release and allow real users to download it while the rest are on the production version. This allows you to find out whether the release has bugs and quickly fix them before rolling it out to everyone. User feedback on these tracks will also not affect the public reviews/ratings of your app.

The ideal way is to release it first on internal tracks during development and internal testing. When a pre-release version is ready, you can create a closed test for a small group of trusted people/users/testers. Then, you can create an open test to allow other users to try your app before the full launch to production.

For personal developer accounts created after November 13, 2023, it is required to do a closed test of the app before you can get access to the production and open test tracks.

To go to each track and manage releases, you can go to the **Test and release** section of the Google Play Console and select **Production** or **Testing**, and then the **Open testing**, **Closed testing**, or **Internal testing** track.

## The feedback channel and opt-in link

In the internal, closed, and open tracks, there are sections for **Feedback URL or email address** and **How testers join your test**. You can provide an email address or a website under **Feedback URL or email address** to which testers can send their feedback. This is displayed when they opt into your testing program.

In the **How testers join your test** section, you can copy the link to share with your testers. They can then join the testing program using this link.

Figure 16.9 shows the **Testers** tab, where you can see the feedback channel and opt-in link for the internal testing track:

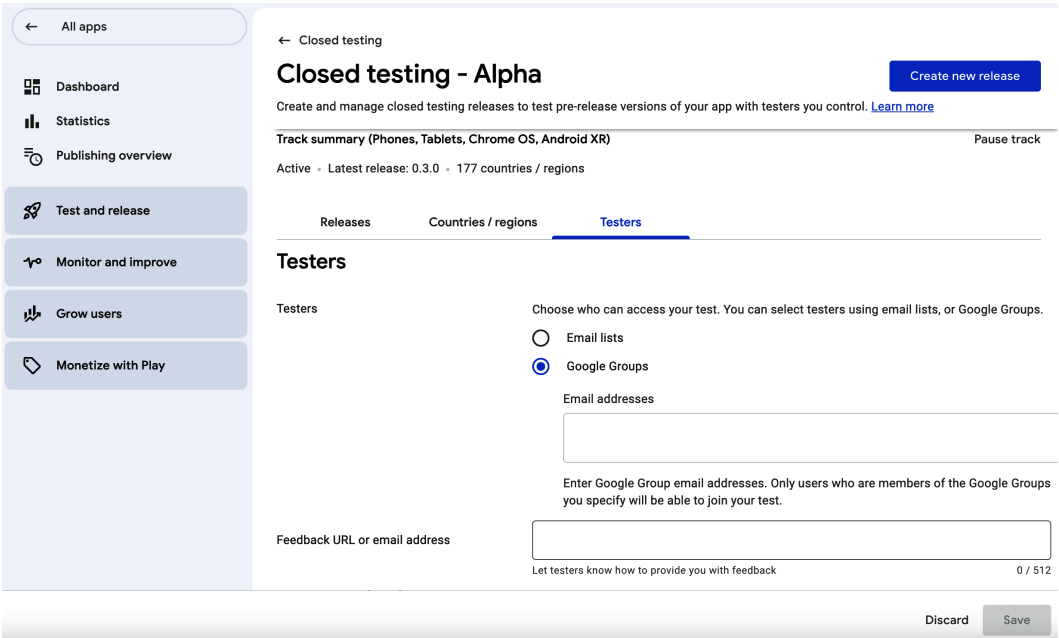


Figure 16.9 – The Testers tab showing the feedback channel and opt-in link

In the next sections, we will be discussing the different test tracks.

## Internal testing

This track is for builds while the developers/testers are developing/testing the app or features. Releases here will be quickly available on Google Play for internal testers. In the **Testers** tab, there's a **Testers** section. You can choose an existing list or create a new one. There is a maximum of 100 testers allowed for an internal test.

## Closed testing

On the **Testers** tab, you can choose **Email lists** or **Google Groups** for the testers. If you choose **Email lists**, choose a list of testers or create a new list. There is a maximum of 2,000 testers allowed for a closed test.

If you select **Google Groups**, you can provide the Google Group email address (for example, the-alpha-group@googlegroups.com), and all the members of that group will become testers.

## Open testing

In the **Testers** tab, you can set **Unlimited** or **Limited number** for the testers. The minimum number of testers for the limited testing that you can set is 1,000.

In the open, closed, and internal tracks, you can add users to be your testers for your applications. You will learn how to add testers in the next section.

## Staged rollouts

When rolling out app updates, you can release them to a small group of users first. Then, when the release has issues, you can stop the rollout or publish another update to fix the issues. If there are none, you can slowly increase the rollout percentage. This is called a **staged rollout**.

If you have published an update to less than 100% of your users, you can go to the Google Play Console, select **Test and release**, click on the track, and then select the **Releases** tab. Below the release you want to update, you can see the **Manage rollout** drop-down menu. It will have options to update or halt the rollout.

You can select **Manage rollout**, then **Update rollout**, to increase the percentage of the rollout of the release. A dialog will appear where you can input the rollout percentage. You can also click the **Update** button to update the percentage.

A 100% rollout will make the release available to all of your users. Any percentage below that means the release will only be available to that percentage of your users.

If a major bug or crash is found during a staged rollout, you can go to the Google Play Console, select **Test and release**, click on the track, and then select the **Releases** tab. Under the release you want to update, select **Manage rollout**, then **Halt rollout**. A dialog will appear with additional information. Add an optional note, then click the **Halt rollout** button to confirm:

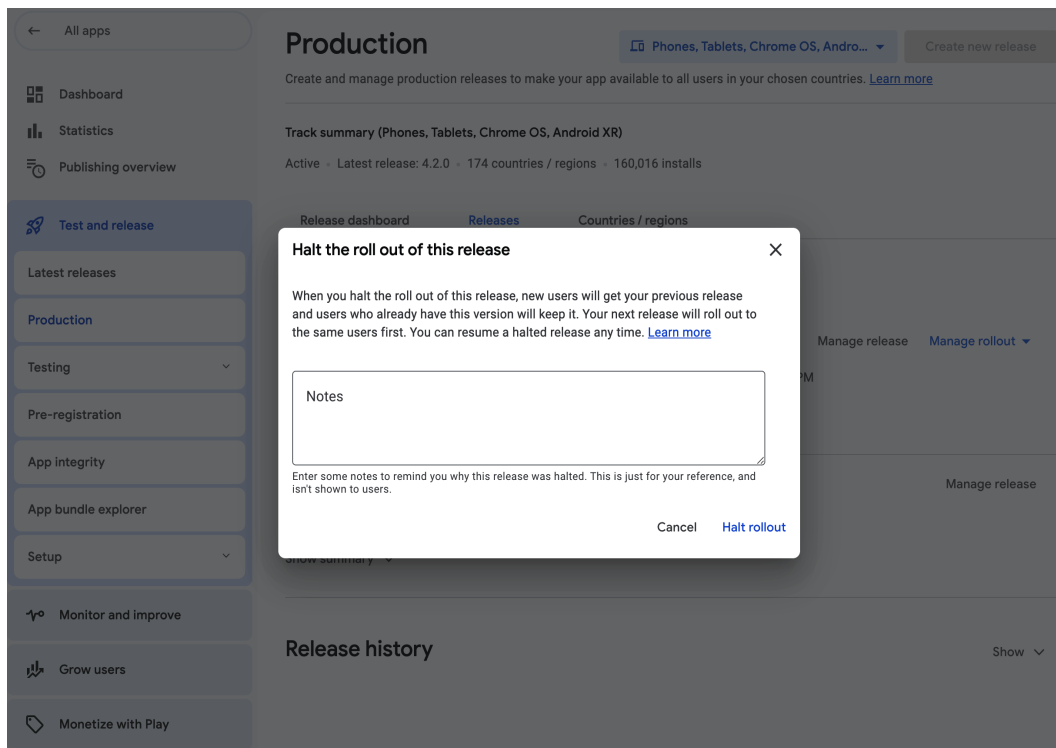


Figure 16.10 – The dialog for halting a staged rollout





 **Quick tip:** Need to see a high-resolution version of this image? Open this book in the next-gen Packt Reader or view it in the PDF/ePub copy.

 **The next-gen Packt Reader** and a **free PDF/ePub copy** of this book are included with your purchase. Scan the QR code OR visit [packtpub.com/unlock](https://packtpub.com/unlock), then use the search bar to find this book by name. Double-check the edition shown to make sure you get the right one.



When a staged rollout is halted, the release page in your track page will be updated with **Rollout halted** and a **Resume rollout** button:

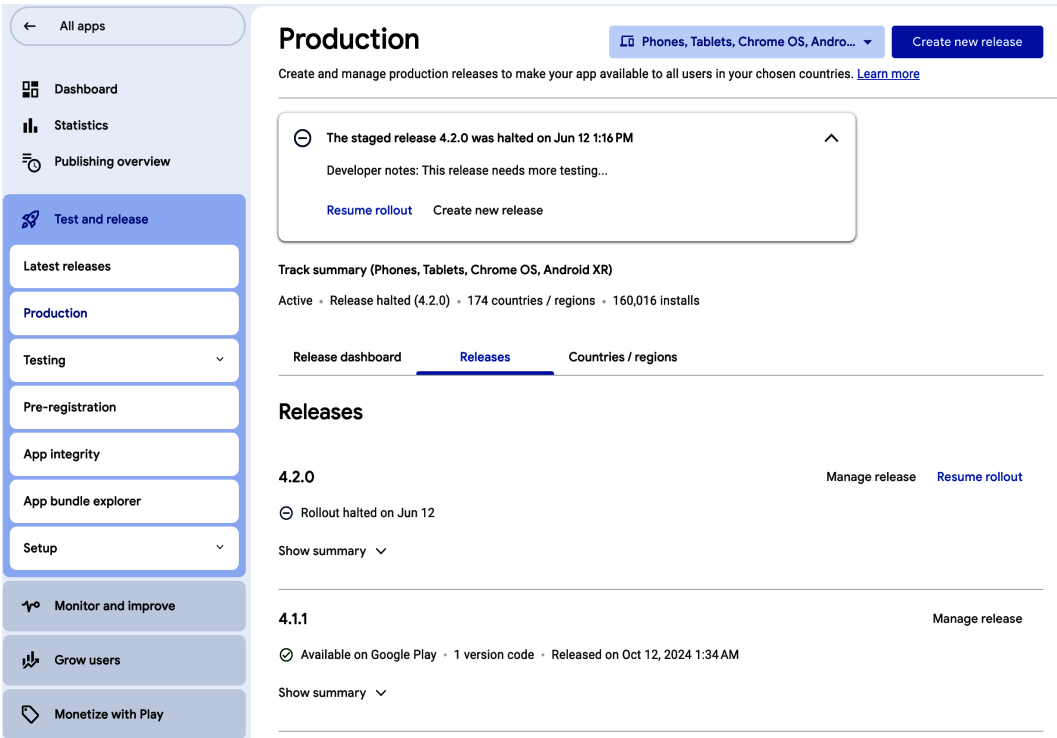


Figure 16.11 – The release page for a halted staged rollout

If you have fixed the issue, for example in the backend, and there’s no need to release a new update, you can resume your staged rollout. To do that, go to the Google Play Console, select **Test and release**, click on the track, and then select the **Releases** tab. Choose the release and click the **Resume rollout** button. In the **Resume staged rollout** dialog, you can update the percentage and click **Resume rollout** to continue the rollout.

In the next section, you will learn about managed publishing on the Google Play Console.

## Managed publishing

When you roll out a new release on Google Play, it will be published automatically. You can change it to be published at a later time. This is useful when targeting a specific day, such as the same day as an iOS or web release, or after a launch date.

Managed publishing must be set up before creating and releasing the update for which you want to control the publishing. When you select your app on the Google Play Console, you can select **Publishing overview** on the left side. In the **Managed publishing** section, click on the **Turn on managed publishing** button:

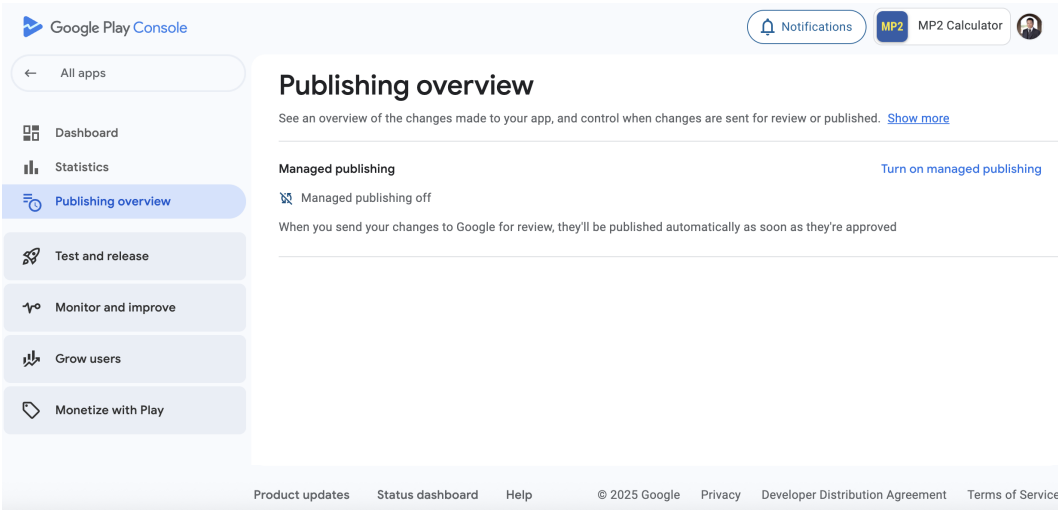


Figure 16.12 – Managed publishing in Publishing overview

The **Managed publishing** dialog will be displayed. Here, you can turn managed publishing on or off, then click the **Save** button.

When you turn on **Managed publishing**, you can continue adding and submitting updates to the app. You can see these changes in **Publishing overview** under the **Changes in review** section.

Once the changes have been approved, **Changes in review** will be empty and the contents will move to the **Changes ready to publish** section. There, you can click the **Publish changes** button. In the dialog that appears, you can click the **Publish changes** button to confirm. Your update will then be published instantly:

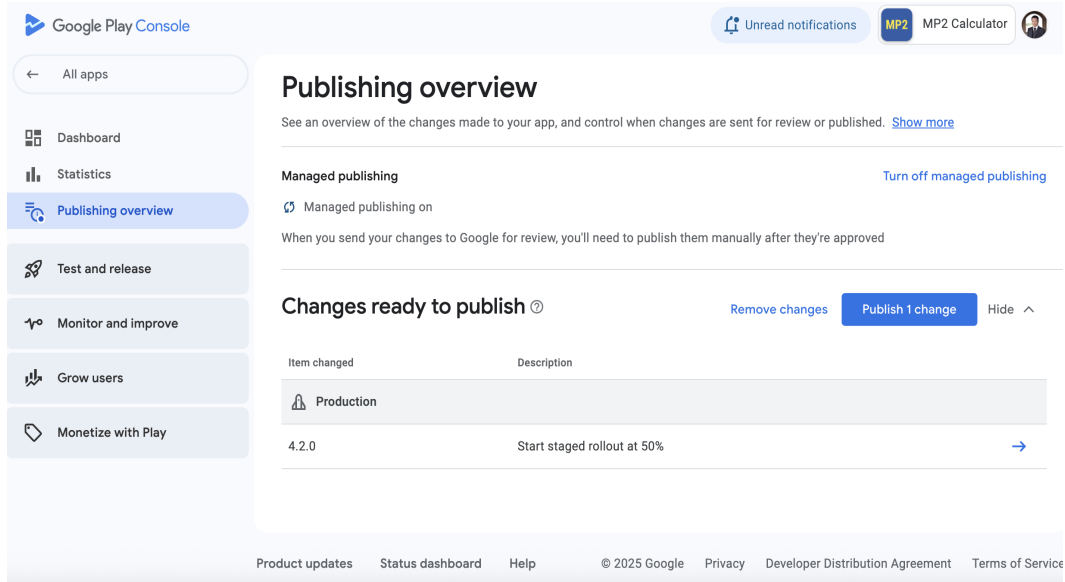


Figure 16.13 – Publishing overview showing Changes ready to publish

In this section, you learned about the release tracks where you can test your releases, perform staged rollouts for your releases, and manage your publishing time.

Let's test everything you've learned by doing an activity.

## Activity 16.01 – publishing an app

For the final activity of this book, you are tasked with creating a Google Play developer account and publishing a newly developed Android app that you have built. You could publish one of the apps you've built as part of this book or another project that you've developed. You can use the following steps as guidelines:

1. Go to the Google Play Console (<https://packt.link/tj3wL>) and create an account.
2. Create a keystore that you can use for signing the release build.
3. Generate an Android App Bundle for release.
4. Publish the app on an open beta track before releasing it to the production track.



The detailed steps for publishing an app have been explained throughout this chapter, so no separate solution is available for this activity. You can follow the exercises in this chapter to successfully complete the preceding steps. The exact steps required will be unique to your app and will depend on the settings you want to use.

## Summary

This chapter covered the Google Play Store: from preparing a release to creating a Google Play developer account and, finally, publishing your app. We started with versioning your apps, generating a keystore, creating an Android App Bundle and signing it with a release keystore, and storing the keystore and its credentials. We then moved on to registering an account on the Google Play Console, uploading your app bundle, and managing releases.

This is the culmination of the work done throughout this book. Publishing your app and sharing it with the world is a great achievement and shows your progress throughout this course.

Throughout this book, you have gained many skills, from the basics of Android app development and building up to implementing features such as fetching data from web services, notifications, and testing. You have seen how to improve your apps with best practices, architecture patterns, and animations, and finally, you have learned how to publish them to Google Play.

Congratulations on finishing the book! This is still just the start of your journey as an Android developer. There are many more advanced skills for you to develop as you continue to build more complex apps of your own and expand upon what you have learned here.

Remember that Android is continuously evolving, so keep yourself up to date with the latest Android releases. You can go to <https://packt.link/i68eu> to find the latest resources and further immerse yourself in the Android world.

### Unlock this book's exclusive benefits now

Scan this QR code or go to [packtpub.com/unlock](https://packtpub.com/unlock), then search this book by name.

*Note: Keep your purchase invoice ready before you start.*

UNLOCK NOW





# 17

## Unlock Your Book's Exclusive Benefits

Your copy of this book comes with the following exclusive benefits:

- ☁ Next-gen Packt Reader
- ✦ AI assistant (beta)
- 📄 DRM-free PDF/ePub downloads

Use the following guide to unlock them if you haven't already. The process takes just a few minutes and needs to be done only once.

### How to unlock these benefits in three easy steps

#### Step 1

Have your purchase invoice for this book ready, as you'll need it in *Step 3*. If you received a physical invoice, scan it on your phone and have it ready as either a PDF, JPG, or PNG.

For more help on finding your invoice, visit <https://www.packtpub.com/unlock-benefits/help>.



#### Note

Did you buy this book directly from Packt? You don't need an invoice. After completing Step 2, you can jump straight to your exclusive content.

## Step 2

Scan this QR code or go to `packtpub.com/unlock`.



On the page that opens (which will look similar to Figure X.1 if you're on desktop), search for this book by name. Make sure you select the correct edition.



Subscription

<sup>0</sup>



Explore Products

Best Sellers

New Releases

Books

Videos

Audiobooks

Learning Hub

Newsletter Hub

Free Learning

### Discover and unlock your book's exclusive benefits

Bought a Packt book? Your purchase may come with free bonus benefits designed to maximise your learning. Discover and unlock them here

 Discover Benefits

 Sign Up/In

 Upload Invoice

[Need Help?](#)

 1. Discover your book's exclusive benefits

CONTINUE TO STEP 2

 2. Login or sign up for free



 3. Upload your invoice and unlock



Figure 17.1: Packt unlock landing page on desktop

## Step 3

Once you've selected your book, sign in to your Packt account or create a new one for free. Once you're logged in, upload your invoice. It can be in PDF, PNG, or JPG format and must be no larger than 10 MB. Follow the rest of the instructions on the screen to complete the process.

## Need help?

If you get stuck and need help, visit <https://www.packtpub.com/unlock-benefits/help> for a detailed FAQ on how to find your invoices and more. The following QR code will take you to the help page directly:



**Note:** If you are still facing issues, reach out to [customer care@packt.com](mailto:customer care@packt.com).







packtpub.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

## Why subscribe?

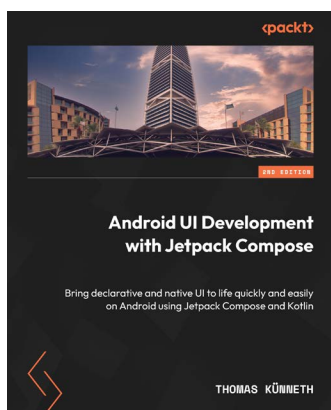
- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

At [www.packt.com](http://www.packt.com), you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.



# Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



## Android UI Development with Jetpack Compose, Second Edition

Thomas Künneth

ISBN: 978-1-83763-425-5

- Recognize the motivation behind Jetpack Compose
- Gain an understanding of the core concepts of Jetpack Compose
- Build a complete app using Jetpack Compose
- Utilize Jetpack Compose inside existing Android applications
- Test and debug apps that use Jetpack Compose
- Understand Material Design and how it is implemented using Jetpack Compose
- Write apps for different form factors
- Bring your Compose UI to other platforms



## **.NET MAUI Cookbook**

Alexander Russkov

ISBN: 978-1-83546-112-9

- Discover effective techniques for creating robust, adaptive layouts
- Leverage MVVM, DI, cached repository, and unit of work patterns
- Integrate authentication with a self-hosted service and Google OAuth
- Incorporate session management and role-based data access
- Tackle real-time updates, chunked file uploads, and offline data mode
- Explore AI integration strategies, from local device to cloud models
- Master techniques to fortify your app with platform-specific APIs
- Identify and eliminate performance and memory issues



## Thriving in Android Development Using Kotlin

Gema Socorro Rodríguez

ISBN: 978-1-83763-129-2

- Create complex UIs with Jetpack Compose
- Structure and modularize apps with a focus on further scaling
- Connect your app to synchronous and asynchronous remote sources
- Store and cache information and manage the lifecycle of this data
- Execute periodic tasks using WorkManager
- Capture and edit photos and videos using CameraX
- Authenticate your users securely
- Play videos in the foreground and background and cast them to other devices

## Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit [authors.packtpub.com](https://authors.packtpub.com) and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

## Share Your Thoughts

Now you've finished *How to Build Android Applications with Kotlin, Third Edition*, we'd love to hear your thoughts! If you purchased the book from Amazon, please [click here](#) to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

# Index

## Symbols

`_state` value 428

## A

**active agents** 233

**activity** 8

**activity callbacks**

logging 59-66

**activity interaction**

result, retrieving 82-94

with intent 73

**activity lifecycle** 40, 56, 58

principal callbacks 56, 57

**activity state**

restoring 66-70

restoring, in Jetpack Compose 71, 72

saving 66-70

saving, in Jetpack Compose 71, 72

**AI Code Explainer** 508

**AlertDialog** 117, 118

**alpha, red, green, blue (ARGB)** 33, 282

**anchoredDraggable**

reference link 237

**Android**

data binding 552

**Android app**

CompositionLocal, adding 561-563

creating, with Jetpack Compose 106-109

running 10, 12

**Android App Bundle** 591, 592

**Android application structure** 37-43

Jetpack Compose UI, building to display  
user greeting 44-51

RGB colors, creating 51, 52

**Android components** 416

**Android manifest file** 17-19

reference link 24

**Android manifest internet permission**

configuring 20-25

**Android Studio** 6

Gradle commands 340

side-effects, adding 565-567

using, to run tests 336-339

**Android Studio project**

creating 6-11, 129, 281, 467, 499, 604

**Android Virtual Device (AVD)** 10

**Animatable** 573

**Animation** 573

**AnimationSpec** 237, 574

types 574



**annotations 516**

- connector 516
- connectors 518, 519
- consumer 516
- consumers 517
- provider 516
- providers 517
- qualifiers 519, 520
- scopes 520, 521
- subcomponents 521, 522

**API response**

- image URL, extracting 202-204

**app dependencies**

- building, with Gradle 25
- configuring, with Gradle 25
- managing, with Gradle 25

**app development**

- for finding parked car location 287, 288

**app-level build.gradle.kts file 26-32****application programming interface (API) 190****app signing service 595****apps release**

- Android App Bundle 591, 592
- Android App Bundle, uploading 598
- app signing service 594
- keystore, creating 586
- keystore, creating in
  - Android Studio 586-589
- keystore, storing 589-591
- managed publishing 605, 606
- managing 600
- passwords, storing 589-591
- preparing 584, 597
- publishing 606
- rolling out 599, 600
- signed app bundle, creating 592-594
- signing 591

- staged rollout 602-605
- store listing, creating 596
- tracking 600
- uploading, to Google Play 596
- versioning 584, 585

**apps release, store listing**

- app details 597
- graphic assets 597

**apps release, tracking**

- closed testing 602
- feedback channel 601, 602
- internal testing 602
- open testing 602
- opt-in link 601, 602

**ARGB\_8888 282****arrange-act-assert (AAA) 329****asset files 472****B****background task**

- starting, with WorkManager 293-297

**bottom navigation 171**

- adding 171
- adding, to app 171-176

**bottom-up approach 107****Box layout group 128, 129, 131****business metrics dashboard**

- creating 144, 145

**C****callback 56****Calls to Action (CTAs) 142****Camera application 453****camera storage 483, 484****cancellation token 267**

- Card layout group** 132, 133
- CatAgents composable** 225
- central processing unit (CPU)** 592
- channels** 305
- checkbox** 114
- Coil** 189
- Column layout group** 133-135
- components**
  - creating, with Jetpack Compose 109
- composable functions** 109
  - button 111
  - icon 112
  - image 112, 113
  - settings screen, creating 118-128
  - text 109-111
  - text input fields 113, 114
- composables** 101
- CompositionLocal**
  - adding, in Android application 561-563
  - using 560, 561
- connectors** 518
- consumers** 517
- ContentNegotiation plugin**
  - tasks 200
- continuous integration (CI)** 339, 589
- coroutine builders** 394
- coroutine context** 397
  - elements 397
- coroutine dispatcher** 396
- coroutine job elements** 398
- coroutines** 193, 392, 426, 427
  - coroutine builders 394
  - coroutine context 397
  - coroutine dispatcher 396
  - coroutine job elements 398

- coroutine scope 394, 395
  - creating 396
  - using, on Android app 392-401
- coroutine scope** 394, 395
- create, read, update, and delete (CRUD)** 432
- current weather**
  - displaying 209-211
- CustomLiveData** 425
- custom markers**
  - adding, to user's map click 283-287
  - using 279-283

## D

- Dagger** 506
- Dagger 2**
  - key functionalities 516
  - using 516
- Dagger injection** 523-529
- data**
  - fetching, from network endpoint 192
  - reading, from API 195-198
- data access object (DAO)** 436-438
- data binding**
  - on Android 552
- data files**
  - asset files 472
  - copying 473-482
  - external storage 470
  - FileProvider 470, 471
  - internal storage 468, 469
  - saving 466, 468
  - Storage Access Framework (SAF) 471, 472
- data models**
  - defining, for JSON mapping 200, 201

**DataStore 461, 462**

- preferences 462-465
- using 454

**data streams 424**

- coroutines and flows 426, 427
- example 427-430
- LiveData 424-426
- using, in ViewModel component 424

**dependency injection (DI) 439, 505**

- client 506
- injector 507
- interface 507
- service 506

**Device File Explorer 466****DisposableEffect 564, 565****domain-specific language (DSL) 9, 214, 537****dots per inch (DPI) 41****double integration 363-373****E****end-to-end tests 327****EnterTransition 569****entities 433-436****ExitTransition 569****explicit intent 73****extended data (extra) 78****Extensible Markup Language (XML) 191****external storage 470**

- methods 470

**F****feature modules 32****FileProvider 470, 471**

- folder hierarchies 470

**flow 426**

- collecting, on Android app 404, 405
- creating, with flow builders 406, 407
- operators, using 407, 408
- TV guide app, creating 411
- using, on Android app 403, 408-410

**Flow builders**

- asFlow() 406
- flow() 406
- flowOf() 406

**foreground worker**

- used, for tracking SCA 307-317
- using 303-307

**Fresco**

- reference link 205

**G****getDatabase method 439****getUriForFile method 476****given-when-then format 352****Google Maps**

- custom markers 279-283
- map clicks 279-283
- permission, requesting from user 254-260
- user's location, displaying 269
- user's location, displaying on map 266-272

**Google Maps API 254****Google Play Console**

- reference link 596

**Google Play developer account**

- creating 595, 596
- reference link 595

**Gradle 25**

- app-level build.gradle.kts file 26-32
- Material Design, exploring in Jetpack Compose 32-37

- project-level build.gradle.kts file 25
- using, to build app dependencies 25
- using, to configure app dependencies 25
- using, to manage app dependencies 25

**Gradle commands**

- in Android Studio 340

**Gson 192****H****hamburger menu 162****Hilt 506, 529-533****Hilt injection 533-536****HTTP-based RESTful API 190****Hypertext Transfer Protocol (HTTP) 190****I****images**

- loading, from obtained URL 207, 208
- loading, from remote URL 205, 206

**image URL**

- extracting, from API response 202-204

**implicit intent 73****inflation 102****injected repositories 544, 545****input/output (I/O) operations 469****integrated development environment (IDE) 6, 560, 590****integration tests 327**

- writing 352

**integration tests, libraries**

- double integration 363-373
- Jetpack ComposeTestRule library 358-363
- Robolectric 353-357

**IntelliJ IDEA IDE 6****intent 73-81**

- using, in activity interaction 73

**intent mode 94****internal storage 468, 469**

- methods 468

**item types 231, 232**

- titles, adding to lazy list 233-235

**J****Jackson 192****Java Development Kit (JDK) 589****JavaScript Object Notation (JSON) 189, 191****Java virtual machine (JVM) 25****Jetpack Compose 326**

- counter app, creating with legacy views 102-105

- DisposableEffect 564, 565

- LaunchedEffect 564

- SideEffect 565

- side-effects, using 563

- used, for creating Android app 106-109

- used, for transitioning XML layouts 102

**Jetpack Compose animations**

- adding 576-580

- adding, to TV Guide app 580

- APIs, customizing 574

- appearance of elements, animating 568-570

- composables changes, animating 572

- creating 567

- debugging 574

- disappearance of elements, animating 568-570

- multiple values, animating 570, 571

- other compose animations 573

- single value, animating 567, 568

size changes of elements, animating 571  
values indefinitely, animating 572, 573

### **Jetpack Compose, layout groups 128**

box layout group 128-131  
business metrics  
    dashboard, creating 144, 145  
card layout group 132, 133  
column layout group 133-135  
profile page, creating 138-144  
row layout group 135, 138  
surface layout group 131, 132

### **Jetpack ComposeTestRule 352**

### **Jetpack ComposeTestRule library 358-363**

### **JSON mapping**

data models, defining 200, 201

### **JSON payload**

processing 192

### **JSON response**

data models, defining for JSON  
    mapping 200, 201  
image URL, extracting from API  
    response 202-204  
Kotlin Serialization, configuring  
    with Ktor 199  
parsing 199

### **JUnit test**

edge cases, handling 331-336  
features 328  
test reliability improvement 331-336  
unit tests, creating 329, 330  
unit tests, running 329, 330  
writing 328, 336

## **K**

### **Koin 506**

using 537-540

### **Koin injection 540-544**

### **Kotlin 6**

### **Kotlin Serialization 189, 192**

configuring, with Ktor 199

### **Kotlin Symbol Processing (KSP) 449**

### **Ktor 189, 190**

API requests, making 193, 194  
data, displaying 193, 194  
internet access, setting up 192, 193  
reference link 192  
used, for configuring Kotlin  
    Serialization 199

## **L**

### **large tests 327**

### **latitude 267**

### **LaunchedEffect 564**

### **launch mode 94**

reference link 94  
setting 95-98  
singleInstance 94  
singleInstancePerTask 94  
singleTask 94

### **LazyColumn 213**

### **LazyColumn composable**

click interaction, implementing 227-231  
populating 217-227

### **lazy list 213**

Add Cat button, implementing 246, 247  
adding, to layout 214, 215  
delete approach 235, 236  
empty LazyColumn, adding 215-217  
items, adding interactively 244, 245  
items, managing 249  
managing, of items 247-249

- swipe gesture detection, adding to delete functionality 240-244
- swipe gesture detection, defining 236-240

**LazyRow** 213**legacy views**

- used, for creating counter app 102-105

**libs.plugins.android.application** plugin

- configuration, settings 27, 28

**LiveData** 424-426**location permission**

- requesting, from user 260-266

**login form**

- creating 99

**longitude** 267**M****manual DI** 506

- applying 510-515
- handling 506-510

**map clicks**

- using 279-283

**Material Design** 32

- exploring, in Jetpack Compose 32-37

**MediaScanner** functionality 485**media storage** 483, 484**medium tests** 327**Mockito** 326, 341-346**MockK** 326**Mockk library**

- using 346, 348

**Model-View-Controller (MVC)** 549**Model-View-Presenter (MVP)** 549**Model-View-ViewModel**

(MVVM) 194, 222, 549, 550, 552

- data binding, on Android 551

Model 550

View 550

ViewModel 550

**Moshi** 192**N****navigation drawer** 161

- features, creating 164-170
- implementing 161, 162

**navigation graph**

- building 151-161

**network endpoint**

- data, fetching 192

**O****object mocking** 341-346

- Mockk library, using 346, 348
- sum of numbers, testing 348-351

**Okio** 469**one-to-many relationship** 443**ongoing work**

- reminder, to drink water 320, 321
- SCA deployment, aborting by canceling 318, 319
- updating 318

**P****permission**

- requesting, from user 257

**postValue()** method 424**Profile page**

- creating 138-144

**progress indicator** 116, 117  
**project-level build.gradle.kts file** 25  
**Project Object Model (POM)** 31  
**providers** 517, 518  
**provides keyword** 561

## Q

**qualifiers** 519, 520  
**Quick Copy feature** 508

## R

**RadioButton** 116  
**recomposition** 44  
**RecyclerView component** 213  
**remote URL**  
    images, loading 205, 206  
**Repository pattern**  
    implementing 552, 553  
    Repository with Room, using in Android project 553-556  
    TV Guide app, implementing 556, 557  
**Repository with Room**  
    using, in Android project 553-556  
**Representational State Transfer (REST) architecture** 190  
**Retrofit** 190  
**Rivest-Shamir-Adleman (RSA)** 589  
**Robolectric** 326, 352-357  
    features 353  
**Robot** 376  
**Room persistence library** 432, 433  
    application, building 443-449  
    data access object (DAO) 432, 436-438  
    database 432  
    database, setting up 438-441

    entities 432-436  
    shopping notes app, creating 449  
    third-party frameworks 442, 443

**row layout group** 136-138

**RxJava** 442

## S

**SCA deployment**  
    aborting, by canceling worker 318, 319  
**Scaffold**  
    used, for creating screen structure 148-150  
**scoped storage** 482  
    camera storage 483, 484  
    media storage 483, 484  
    multiple persistence options, managing 501, 502  
    photo shoot 485-500  
**scopes** 520, 521  
**screen structure**  
    creating, with Scaffold 148-151  
    creating, with slots 148-150  
**Secret Cat Agents (SCAs)** 292  
    tracking, with foreground worker 307-317  
**semantic versioning**  
    reference link 584  
**separation of concerns (SoC)** 88  
**services** 291  
**settings screen**  
    creating 118-128  
**setValue() method** 424  
**SharedPreferences** 454, 455  
    using 454  
    wrapping 455-461  
**Short Message Service (SMS)** 20

**SideEffect 565**

- adding, in Android Studio 565-567
- with Jetpack Compose 563

**singleTop activity 94****sleeper agents 233****slider 115****slots**

- used, for creating screen structure 148-150

**software development kit (SDK) 10****staged rollout 602****state hoisting 88****state saving 431****state value 428****Storage Access Framework (SAF) 471, 472****subcomponents 521, 522****Surface layout group 131, 132****suspend keyword 462****swipe-to-delete functionality**

- adding 240-244

**switch 115****T****tabbed navigation 177**

- primary app navigation, building 184, 185
- secondary app navigation, building 184, 185
- tabs, using for app navigation 177-183

**tasks mode 94****test-driven development (TDD) 325****test-driven development (TDD) approach**

- applying 384, 385
- used, for developing application 388, 389
- using, to calculate sum of numbers 385-388

**testing pyramid 327****test types 326, 327**

- integration tests 327
- UI tests 327
- unit tests 326

**text input fields 113, 114**

- AlertDialog 117, 118
- checkbox 114
- progress indicator 116, 117
- RadioButton 116
- slider 115
- switch 115

**The Cat API**

- reference link 192, 201

**Tom's Obvious, Minimal Language (TOML) file 29****TransformLiveData 425****TV Guide app 556****type-safe navigation 153****U****UI tests 327**

- dealing, with random events 377-384
- running 373-377

**Uniform Resource Locators (URLs) 192****unit tests 326****user interface (UI) 3, 147, 325, 391, 415, 549, 559, 596****user's current location**

- displaying, on map 266-272
- obtaining 272-279

**V****ViewModel component 417, 420**

- example 420-423
- lifecycle 418, 419
- with data streams 424



**virtual device**

app, running 12-16

setting up 10-16

**W**

**workers** 291

**WorkManager class**

used, for executing background  
work 297-303

used, for starting with background  
task 293-297

**writeStream method** 475

**X****XML layouts**

transitioning, to Jetpack Compose 102

