

Second Edition – Kotlin 1.9.10

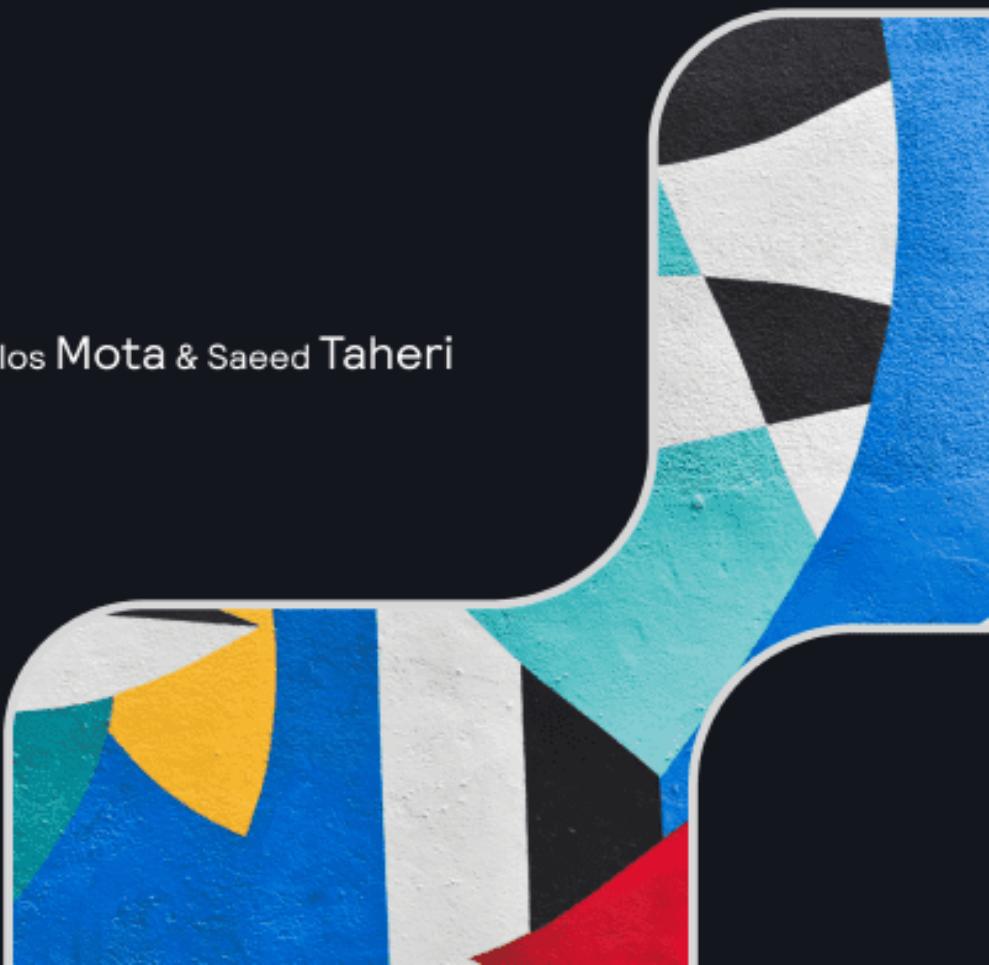
Kotlin Multiplatform by Tutorials

Build Native Apps Faster by Sharing
Code Across Platforms

By the Kodeco Team

Kevin D. Moore, Carlos Mota & Saeed Taheri

Kodeco



Kotlin Multiplatform by Tutorials

By Kevin D. Moore, Carlos Mota & Saeed Taheri

Copyright ©2023 Kodeco Inc.

Notice of Rights

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

Notice of Liability

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

Trademarks

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

Table of Contents: Overview

Book License	11
Before You Begin	12
What You Need.....	13
Book Source Code & Forums	14
Introduction	18
Section I: Beginning Kotlin Multiplatform	21
Chapter 1: Introduction	22
Chapter 2: Getting Started	44
Chapter 3: Developing UI: Android Jetpack Compose	71
Chapter 4: Developing UI: iOS SwiftUI	112
Chapter 5: Developing UI: Compose Multiplatform..	136
Section II: Kotlin Multiplatform: Intermediate.....	164
Chapter 6: Connecting to Platform-Specific API	165
Chapter 7: App Architecture.....	192
Chapter 8: Testing.....	218
Chapter 9: Dependency Injection	240
Chapter 10: Data Persistence.....	259
Section III: Kotlin Multiplatform: Advanced	285
Chapter 11: Serialization	286
Chapter 12: Networking	312
Chapter 13: Concurrency	345

Chapter 14: Creating Your KMP Library	381
Conclusion	416
Section IV: Appendices	417
Appendix A: Kotlin: A Primer for Swift Developers ...	418
Appendix B: Debugging Your Shared Code From Xcode	440
Appendix C: Sharing Your Compose UI Across Multiple Platforms.....	449

Table of Contents: Extended

Book License	11
Before You Begin.....	12
What You Need	13
Book Source Code & Forums	14
About the Authors	16
About the Editors	17
Introduction	18
How to Read This Book	19
Section I: Beginning Kotlin Multiplatform	21
Chapter 1: Introduction.....	22
What Is Kotlin Multiplatform?.....	23
Setting Up Your Environment.....	30
Creating Your First Project	34
Key Points.....	43
Where to Go From Here?.....	43
Chapter 2: Getting Started	44
Getting to Know Gradle	45
Version Catalog.....	45
Build Files.....	50
Find Time	56
Business Logic	57
Challenge	69
Key Points.....	70
Where to Go From Here?.....	70
Chapter 3: Developing UI: Android Jetpack Compose	71
UI Frameworks.....	72

Jetpack Compose	72
Time Finder	75
Time Zone Screen.....	92
Find Meeting Time Screen	102
Key Points	111
Where to Go From Here?.....	111
Chapter 4: Developing UI: iOS SwiftUI.....	112
Getting to Know SwiftUI	115
Key Points	135
Where to Go From Here?.....	135
Chapter 5: Developing UI: Compose Multiplatform	136
Getting to Know Compose Multiplatform	137
Creating a Desktop App.....	137
Shared UI.....	142
Key Points	163
Where to Go From Here?.....	163
Section II: Kotlin Multiplatform: Intermediate ...	164
Chapter 6: Connecting to Platform-Specific API	165
Reusing Code Between Platforms.....	166
Say Hello to Organize	166
Updating the Platform Class	168
Updating the UI	179
Challenge.....	191
Key Points	191
Where to Go From Here?.....	191
Chapter 7: App Architecture	192
Design Patterns	193
Sharing Business Logic	197
Creating Reminders Section	204

Sharing Tests and UI.....	215
Challenge.....	216
Key Points	217
Chapter 8: Testing	218
Setting Up the Dependencies	219
Writing Tests for RemindersViewModel	221
Writing Tests for Platform.....	224
UI Tests	227
Challenge.....	238
Key Points	238
Where to Go From Here?.....	239
Chapter 9: Dependency Injection	240
Advantages of Dependency Injection	241
Automated DI vs. Manual DI	241
Setting Up Koin.....	243
Using Koin on Each Platform	246
Updating AboutViewModel	252
Testing.....	254
Key Points	257
Where to Go From Here?.....	258
Chapter 10: Data Persistence	259
Key-Value Storage.....	260
Database	270
Challenge.....	283
Key Points	284
Where to Go From Here?.....	284
Section III: Kotlin Multiplatform: Advanced.....	285
Chapter 11: Serialization.....	286
The Need for Serialization.....	287

Project Overview.....	287
Application Features	291
Adding Serialization to Your Gradle Configuration	292
Different Serialization Formats	294
Creating a Custom Serializer.....	295
Serializing/Deserializing New Data	297
Serializable vs. Parcelable	300
Implementing Parcelize in KMP	300
Sharing Your Data Classes With the Server.....	304
Testing.....	307
Challenges.....	310
Key Points	311
Where to Go From Here?.....	311
Chapter 12: Networking	312
The Need for a Common Networking Library.....	313
Adding Ktor.....	313
Connecting to the API With Ktor.....	314
Plugins.....	317
Retrieving Content.....	322
Adding Headers to Your Request	333
Uploading Files	337
Testing.....	340
Challenge.....	343
Challenge: Send Your Package Name in a Request Header	343
Key Points	344
Where to Go From Here?.....	344
Chapter 13: Concurrency	345
Concurrency and the Need for Structured Concurrency	346
Understanding kotlinx.coroutines.....	347
Structured Concurrency in iOS	354
Using kotlinx.coroutines	355

Working With <code>kotlinx.coroutines</code>	356
Improving Coroutines Usage for Native Targets	368
Challenge.....	379
Key Points	380
Where to Go From Here?.....	380
Chapter 14: Creating Your KMP Library	381
Migrating an Existing Feature to Multiplatform	382
Publishing Your KMP Library.....	400
Handling Multiple Frameworks	412
Challenges.....	413
Key Points	414
Where to Go From Here?.....	415
Conclusion.....	416
Section IV: Appendices	417
Appendix A: Kotlin: A Primer for Swift Developers	418
Kotlin and Swift: Comparing Both Languages.....	418
Kotlin and Swift Syntax Table	439
Where to Go From Here?.....	439
Appendix B: Debugging Your Shared Code From Xcode ...	440
Debugging the Shared Module.....	441
Where to Go From Here?.....	448
Appendix C: Sharing Your Compose UI Across Multiple Platforms	449
Setting Up an iOS App to Use Compose Multiplatform	449
Updating Your Project Structure.....	453
Sharing Your UI Code.....	456
Migrating Your Android UI Code to Multiplatform	456
Compose Multiplatform.....	460
Using Third-Party Libraries	463

Handling Resources	468
What's Missing?	481
Where to Go From Here?.....	488



Book License

By purchasing *Kotlin Multiplatform by Tutorials*, you have the following license:

- You are allowed to use and/or modify the source code in *Kotlin Multiplatform by Tutorials* in as many apps as you want, with no attribution required.
- You are allowed to use and/or modify all art, images and designs that are included in *Kotlin Multiplatform by Tutorials* in as many apps as you want, but must include this attribution line somewhere inside your app: “Artwork/images/designs: from *Kotlin Multiplatform by Tutorials*, available at www.kodeco.com”.
- The source code included in *Kotlin Multiplatform by Tutorials* is for your personal use only. You are NOT allowed to distribute or sell the source code in *Kotlin Multiplatform by Tutorials* without prior authorization.
- This book is for your personal use only. You are NOT allowed to reproduce or transmit any part of this book by any means, electronic or mechanical, including photocopying, recording, etc. without previous authorization. You may not sell digital versions of this book or distribute them to friends, coworkers or students without prior authorization. They need to purchase their own copies.

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

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

Before You Begin

This section tells you a few things you need to know before you get started, such as what you'll need for hardware and software, where to find the project files for this book, and more.



What You Need

To follow along with this book, you'll need the following:

- **Kotlin 1.9.10:** This book uses Kotlin 1.9.10 throughout to maintain compatibility with versions of several other libraries.
- **Android Studio Hedgehog | 2023.1.x:** Available at <https://developer.android.com/studio/>. You'll use Android Studio to develop the Android and desktop apps in this book.
- **Xcode 15.0:** Available at <https://developer.apple.com/xcode/>. You'll need Xcode to compile and run the iOS apps in this book.
- **macOS Ventura:** You'll need macOS to compile the iOS-specific code and run the tests targeting iOS.



Book Source Code & Forums

Where to Download the Materials for This Book

The materials for this book can be cloned or downloaded from the GitHub book materials repository:

- <https://github.com/kodecocodes/kmpf-materials/tree/editions/2.0>

Forums

We've also set up an official forum for the book at <https://forums.kodeco.com/c/books/kotlin-multiplatform-by-tutorials>. This is a great place to ask questions about the book or to submit any errors you may find.

“To my wife and family for letting me create and learn new things.”

— *Kevin Moore*

“To Beatriz, José, and Petá for being my guiding light. To Carlos and Maria for always giving everything and more for my happiness and future. My most sincere and forever thank you.

To Daniela, for all the love, companionship and support. For always finding the brightness of every scenario.

To all my friends who are always a phone call away. A special mention to Ricardo for all those architecture discussions, to Andreia and Curto for the lovely dinners, and to Fred and Jorge, the iOS developers who embraced Kotlin.

To you, amazing reader, welcome aboard!”

— *Carlos Mota*

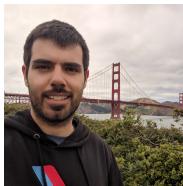
“To my wife Leila and my dad, who will never read this book! Their unconditional love, patience and support made me courageous enough to write a book.”

— *Saeed Taheri*

About the Authors



Kevin David Moore is an author of this book. He is a Google Developer Expert in Flutter and has been developing Android apps for over 13 years and Flutter apps for over 3 years. He's written several articles, books and videos at <https://www.kodeco.com>. He is a cat lover with three cats and he is working towards his black belt in Taikawando.



Carlos Mota is an author of this book. Carlos is an Android GDE and can easily be spotted either working on Android applications written in Kotlin or developing them along with Kotlin Multiplatform. He's enthusiastic about new technology and constantly trying to reach that last 20% of all of his side projects that seem to be really far away. He loves to share his knowledge with others by giving talks, teaching, writing or along with a cold beer in the nearest pub. A GDG Coimbra organizer and Kotlin evangelist, he also has a giant passion for travel, photography, space and the occasional run.



Saeed Taheri is an author of this book. He has been creating iOS applications for about 12 years and Android applications for about 6 years. He has written articles at <https://www.kodeco.com>. When not at his computer, he enjoys spending time with his family, playing football or watching it on TV.

About the Editors



Godfred Afful is a tech editor of this book. He is a software engineer specializing in backend and mobile applications development. He loves art, sports, nature and monkeys.



Yogesh Choudhary is a tech editor of this book. He is a software engineer with 4+ years of experience who likes to create high-performing applications with organized architecture. His go-to framework for developing applications is mostly Flutter nowadays; after that he transitions to native frameworks. He is also skilled in end-to-end ML model development and product design. He embodies a passionate and always curious type of personality for anything tech- and science-related, and he loves learning, sharing, discussing, and exploring the same.



Subhrajyoti Sen is the final pass editor for this book. He is a Google Developer Expert for Android and an Android Engineer at Motive, where he develops apps to improve the trucking industry. Before this, he also worked on apps to improve the experience of Indian investors. He believes in the power of Open Source and communities, and he actively tries to give back. When not writing code, you can find him binge-watching anime, reading up on public policy, or playing Rocket League.



Introduction

If your goal is to leverage Kotlin to share code among your native apps, this is the book for you.

Maintaining multiple native apps with duplicated code can be a time-consuming process. This duplication also increases the testing effort, eventually slowing down the project and increasing costs.

You can use Kotlin Multiplatform to share code between your Android, iOS and desktop apps but there are multiple considerations. You should be able to develop the UI natively using the framework of your choice. Using the right frameworks can drastically reduce the UI development time and provide you with flexible APIs.

At the same time, you need to figure out how Kotlin Multiplatform fits in with your current architecture and how you can access platform-specific APIs. Choosing the right architecture can make your app testable, maintainable and easy to work with.

Then you need to figure out which layers of your app you can migrate to a shared module and how you can use different libraries to assist this migration. Finally, you should be able to publish and share your shared module so that you can use it across apps on multiple platforms.

How to Read This Book

In this book, every chapter contains theory about the specific topic and a simple practical task to learn implementation faster. There are three awesome applications you'll develop — one per section. They each focus on important aspects of Kotlin Multiplatform.

To learn and notice every little detail, read the chapters in order. However, this book is for advanced users, and you might want to skip some chapters. In that case, be sure to continue from the starter project of the chapter you're moving to. The starter projects contain all steps implemented in the previous chapters of a certain section.

While going through the chapter, you can type the code in Android Studio immediately. Feel free to play with the code and investigate the references provided in the chapter. Some of the chapters contain fun challenges for you to expand upon the topics you learned.

This book is split into three main sections:

Section I: Beginning Kotlin Multiplatform

One of the core benefits of Kotlin Multiplatform is that you can share code across native apps. You can continue to develop the UI layer using native UI toolkits like Jetpack Compose for Android and SwiftUI for iOS.

In this section, you'll learn how to add a new Gradle module to write your business logic only once. You'll also learn how to create the native UI for Android, iOS and desktop apps, all while sharing the common module.

Section II: Kotlin Multiplatform: Intermediate

To effectively share code across apps, there are multiple things to keep in mind: access to platform-specific APIs, support for existing software engineering practices and persistence.

In this section, you'll learn how to use Kotlin features to access platform-specific APIs in your shared module and how Kotlin Multiplatform fits in with your current architecture. You'll also learn about dependency injection and how you can use it to test features present in your shared modules. Finally, you'll learn how to use a common codebase to handle persistence on different platforms.

Section III: Kotlin Multiplatform: Advanced

Networking is crucial to most modern apps, and it usually involves implementing similar logic using different frameworks and languages. Under the hood, it also involves concepts like serialization and concurrency. Fortunately, Kotlin Multiplatform has dedicated libraries for each of these.

In this section, you'll learn how to use serialization to decode JSON data to Kotlin objects. You'll then learn how to use a common networking library that leverages this common serialization to fetch data from the internet. To make the networking performant, you'll also learn about concurrency in Kotlin using coroutines and the considerations for different platforms. Finally, you'll learn how to extract an existing feature to a Kotlin Multiplatform library and also different ways of publishing this library.

Section I: Beginning Kotlin Multiplatform

One of the core benefits of Kotlin Multiplatform is that you can share code across native apps. You can continue to develop the UI layer using native UI toolkits like Jetpack Compose for Android and SwiftUI for iOS.

In this section, you'll learn how to add a new Gradle module to write your business logic only once. You'll also learn how to create the native UI for Android, iOS and desktop apps, all while sharing the common module.

1 Chapter 1: Introduction

By Kevin Moore

Congratulations! By reading this book, you're taking the first step toward learning how to write less code. In three sections, you'll learn how to use Kotlin Multiplatform (KMP) to set up and write iOS, Android and desktop apps using the latest user interface (UI) technologies.

In this book, you'll develop several different apps — a time zone meeting helper, an app to track your list of things to do and an app that displays a list of all [kodeco.com](#)'s books, articles and videos.

You'll learn how to leverage KMP by sharing business logic across platforms and creating customized native UIs in each platform. And, you'll learn how to write tests for all your business logic, use the popular JetBrains library Ktor to handle network calls and, of course, use Kotlin Coroutines to handle concurrency.

This book requires some knowledge in mobile development but will walk you through the setup for both iOS and Android, as well as for desktop apps. While most of the book uses Kotlin, iOS developers familiar with Swift will be able to pick up Kotlin easily.

In this chapter, you'll learn about Kotlin Multiplatform and the history of cross-platform frameworks. At the end of the chapter, you'll set up your environment, create a new project and run your project on Android and iOS.

What Is Kotlin Multiplatform?

Kotlin is a modern and type-safe programming language. It incorporates null safety, preventing many of the dreaded null pointer exceptions that have plagued programming for years. Kotlin also has many innovative features — like data classes and sealed classes, extension functions that let you extend classes with functions outside of the class, lazy loading of variables and many more.

As the name implies, Kotlin Multiplatform uses the Kotlin programming language and works on multiple platforms. Kotlin already works on platforms that support the Java Virtual Machine (JVM), and it uses Kotlin Native for platforms that don't support the JVM. Kotlin Native compiles Kotlin to native bytecode that runs natively on Apple's operating systems, Windows and Linux. On the web, KMP compiles Kotlin to JavaScript and HTML.

KMP supports the following platforms:

- Android
- iOS
- macOS
- watchOS
- tvOS
- Windows
- Linux
- Web

That's a lot of platforms. Some, like the web, are not stable at the moment.

KMM

You may have heard the term: Kotlin Multiplatform Mobile (KMM). JetBrains is deprecating this term in favor of Kotlin Multiplatform (KMP) as the technology supports multiple platforms other than mobile alone.

History of Cross-Platform

For as long as there have been both iOS and Android devices, developers have considered the holy grail of app development to be one codebase that could run on both. Many frameworks have tried to achieve multiplatform development, including:

- **PhoneGap**: One of the earliest, PhoneGap enabled you to write mobile apps using HTML5, CSS3 and JavaScript. It was discontinued in 2020.
- **Apache Cordova**: Open source fork of PhoneGap.
- **Ionic**: Uses Angular, React and Vue UI frameworks.
- **Appcelerator Titanium**: JavaScript-based SDK that supported iOS, Android, Windows and Blackberry. It was discontinued in 2022 and later open-sourced.

The frameworks above worked by using web technologies to display either native controls or controls designed to look native. However, they suffered from slow JavaScript-to-native communication and had to be updated every time the native platform changed.

- **Xamarin**: Microsoft-owned C#-based development framework that includes the .NET runtime. The framework is compiled for iOS — so it's faster on iOS than on Android, which uses a just-in-time compilation.
- **React Native**: Facebook's mobile-based framework based on the popular React web framework. It's web- and JavaScript-based. It too has a slow bridge between native and web.
- **Flutter**: This is the new kid on the block, and it works on all platforms. One of the main benefits of this framework is that you can write almost all your UI once. Some UI will need to be different based on the platform. For example, desktop and web don't need a toolbar. One of the disadvantages is that it's written using the Dart language, which many developers don't know. Dart has only recently gotten null safety, and a lot of packages use code generation, which has to be done manually.

A lot of the web-based frameworks are falling out of favor. Flutter is going strong, but many question the use of Dart.

History of Kotlin

Kotlin has been around officially since July 2011. JetBrains released version 1.0 on February 15, 2016, and it was announced at Google I/O 2017 as a first-class language for Android development. JetBrains developed Kotlin because most languages didn't have the features they were looking for. JetBrains now uses Kotlin as its preferred development language for all current work — slowly replacing Java.

Why Kotlin?

Why should you use Kotlin? Because it's one of the only languages that you can compile for both JVM and native and use on iOS as well as the desktop and web.

Kotlin is ideal for server work as well. With the Ktor library, networking is an easy task. Writing common business logic ensures that all platforms behave the same way and you only need to test once. It uses the same code for all platforms, reducing the possibility of errors and speeding up development. Each team can use as much shared code as they want. Start slowly with existing projects, or start writing all your business logic with new projects.

iOS developers are familiar with Swift, and Kotlin is very similar — so the learning curve should be minimal. Developers still use Swift on the UI side, but they can also work and help out with business logic in Kotlin. Since there will still be a lot of iOS development work needed, iOS developers will be included in all parts of development.

How much code to share is up to the team. If you have an existing app, you can slowly move over your business logic so you have a shared set of code that you can test once.

KMP adds minimal extra size to an app. The standard library is small and you only need to include the parts you use. Lots of apps in the app stores already use it. Many companies find that writing their business logic once — instead of on both iOS and Android — saves the team a lot of time. The UIs are native, making the mobile developers happy, and the users are happy they have a fast experience.

What KMP Is Not

While KMP provides Kotlin as the programming language, it doesn't provide a UI. If you want to create a UI for Android, you can write it in native code or use the newer Jetpack Compose UI framework. For iOS, you can use UIKit, the newer SwiftUI framework, or the Alpha version of Compose Multiplatform. For the desktop, you can use Desktop Compose or Java Swing. In other words, you have a choice for how you write your UI. Many see this as an advantage — the UI will always be native, so it won't suffer from the slow bridge communication that web-based frameworks have.

When to Use KMP

One of the nice features of KMP is that you can use as much or as little as you want. If you have an existing app, you can use it for new features or start replacing a feature with KMP. If you start by using KMP for some of the lower layers of your app, you can reuse it for all your platforms. For instance, you can use SQLDelight to replace all your database code with just one set of code. Or, you can write your business logic just once and reuse it on all your platforms. If you need to create code to access networked APIs, you can write it once to work on all platforms.

Layers

Most apps consist of different layers. There's typically a network layer, a database layer (if needed), a repository layer that interacts with the database, a business logic layer (not always) and a UI layer. KMP doesn't provide a UI layer; you'll use the native UI instead.

Business Logic

Most companies now have teams of iOS and Android developers. Each team takes a set of specifications and writes different code to implement those specifications. When testing, each team needs to make sure the logic they've implemented works the same as on the other platform. But with two different sets of code, how do they know that all the corner cases work the same way? With one codebase for business logic, both teams can review the code to make sure the logic matches the specifications and know it will work the same for both platforms. With one business logic code base, you can have either both teams work on it together or have one team specialize in writing business logic.

Database

You can write the database layer using SQLite on mobile and desktop using the SQLDelight library. This library is a multiplatform package designed to run on all these platforms. Imagine having to write this set of code only once. Not only will you write only one set of lower-level SQL database insertions, deletions and updates, but your repository layer only needs to be written once. SQLDelight uses SQL statements to generate code for you. You only need to test once.

UI

Since KMP doesn't provide a UI layer, you can use whichever UI system you want. For iOS, developers are turning to **SwiftUI**: a nice, declarative UI toolkit that makes it easy to create beautiful UIs. The Alpha version of Compose Multiplatform is available to use on iOS so that you can leverage your knowledge of Compose. Now that **Jetpack Compose** has been released as stable, Android developers can use it. Cross-platform desktop UIs have been neglected for quite a while. Swing has been a standard for some time, but it's old and unmaintained. JetBrains hopes to replace it with **Compose for Desktop**. It uses a lot of Android's Jetpack Compose underneath, with a layer of desktop code.

Is It Native?

One of the questions most often asked is: Does it use native controls? The answer is yes. Since KMP doesn't provide any UI layer, all UI is drawn natively. On Android, that can be the built-in View system or the new Jetpack Compose library. On iOS, you can use the built-in native UI, the newer SwiftUI or Compose Multiplatform. On the desktop, you can use the older Java Swing or the newer Desktop Compose. For the Mac desktop, you can also use SwiftUI, AppKit or Compose Multiplatform. For Android, code is generated as Java class files, while iOS uses LLVM to produce native code and create an Xcode framework library.

Current State Of KMP

At the time of writing, KMP is currently in beta, but production apps on both the Google and Apple stores already use it. Since there are multiple layers, here's the current state of the platform as of the writing of this book:

Component	Status
Kotlin/JVM	Stable
Kotlin K2 (JVM)	Alpha
kotlin-stdlib (JVM)	Stable
Coroutines	Stable
kotlin-reflect (JVM)	Beta
Kotlin/JS (Classic back-end)	Stable
Kotlin/JVM (IR-based)	Stable
Kotlin/JS (IR-based)	Stable
Kotlin/Native Runtime	Beta
Kotlin/Native memory manager	Beta
klib binaries	Beta
Kotlin Multiplatform	Beta
Kotlin/Native interop with C and Objective C	Beta

Fig. 1.1 – KMP Component Status

And:

CocoaPods integration	Beta
Kotlin Multiplatform Mobile plugin for Android Studio	Beta
expect/actual language feature	Beta
KDoc syntax	Stable
Dokka	Beta
Scripting syntax and semantics	Alpha
Scripting embedding and extension API	Beta
Scripting IDE support	Beta
CLI scripting	Alpha
Compiler Plugin API	Experimental
Serialization Compiler Plugin	Stable
Serialization Core Library	Stable
Inline classes	Stable
Unsigned arithmetic	Stable

Fig. 1.2 – KMP Component Status

Since Android apps are built with Kotlin and have been for years, there are no compatibility issues. iOS and macOS apps interact with frameworks built with the KMP system. This will continue to evolve and improve but the feature is still in beta as of this writing. Desktop apps can use the same shared code as the other platforms but use their own UI.

Setting Up Your Environment

You can use either IntelliJ or Android Studio to do KMP work. In this book, you'll use Android Studio because it seems to work better with mobile platforms at the time of writing.

Downloading Android Studio

Go to <https://developer.android.com/studio> and download Android Studio Hedgehog edition or later. Once installed, go to Android Studio's preferences and then to plugins. Search for **multiplatform** and install the **Kotlin Multiplatform Mobile** plugin as well as the **Compose Multiplatform IDE Support** plugin:

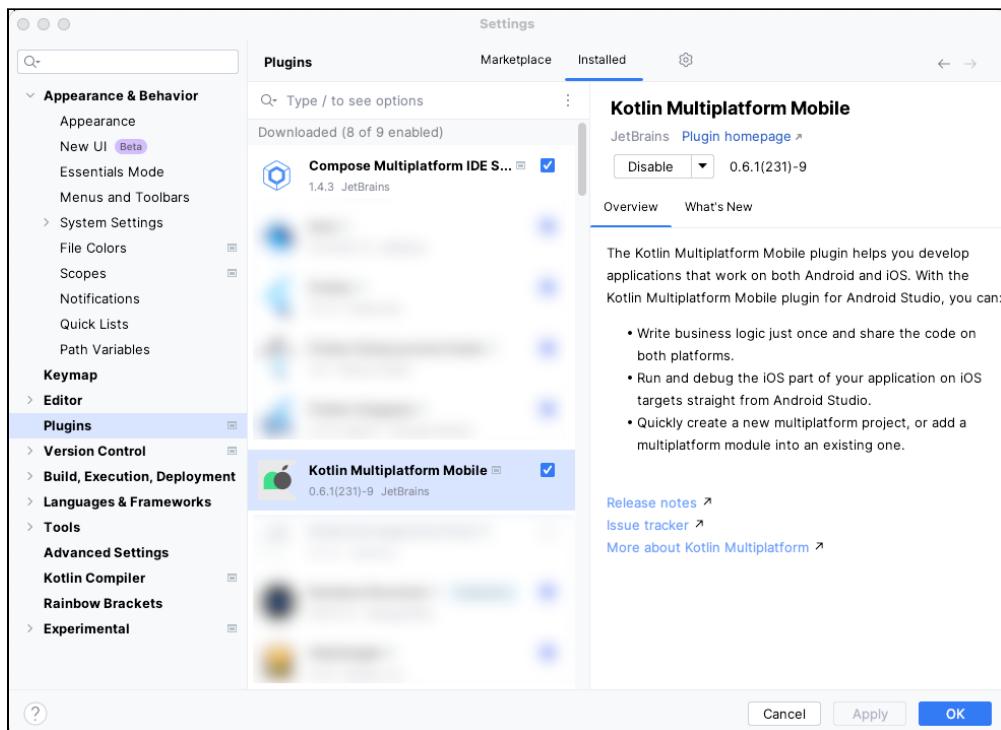


Fig. 1.3 – Android Studio Plugins

Restart Android Studio to enable it. Note that Android Studio has a new Beta UI that you can try out.

Existing UI:

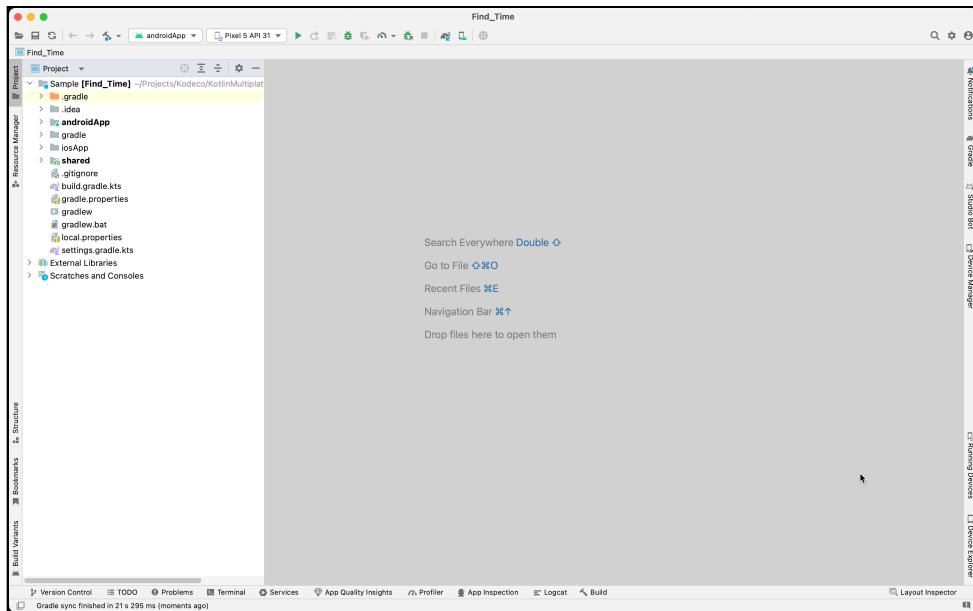


Fig. 1.4 – Android Studio Old UI

And the new UI:

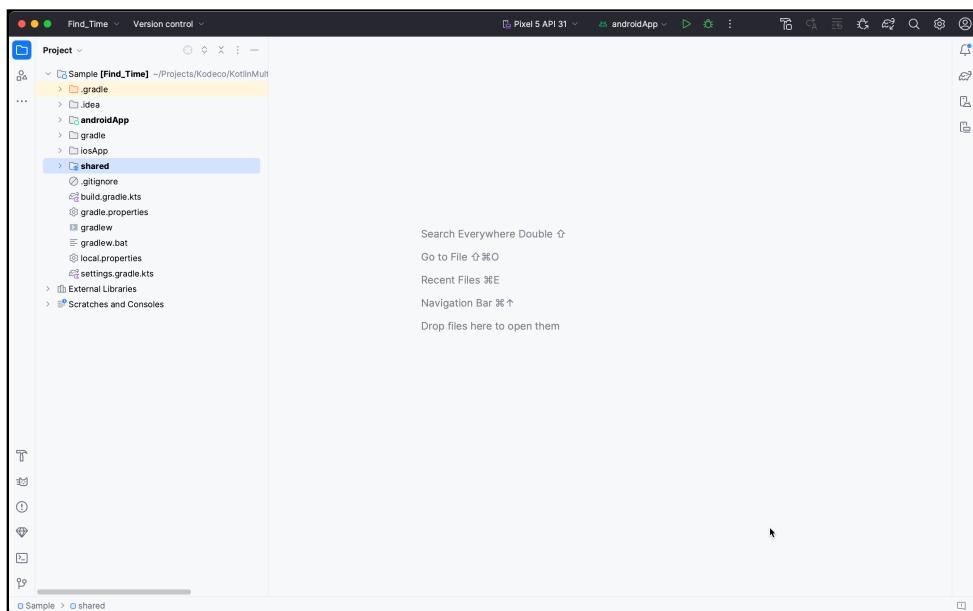


Fig. 1.5 – Android Studio Old UI

Downloading Xcode

To develop for iOS or macOS, you'll need to install Xcode from the App Store onto your Mac. Make sure you open Xcode to install its tools as well.

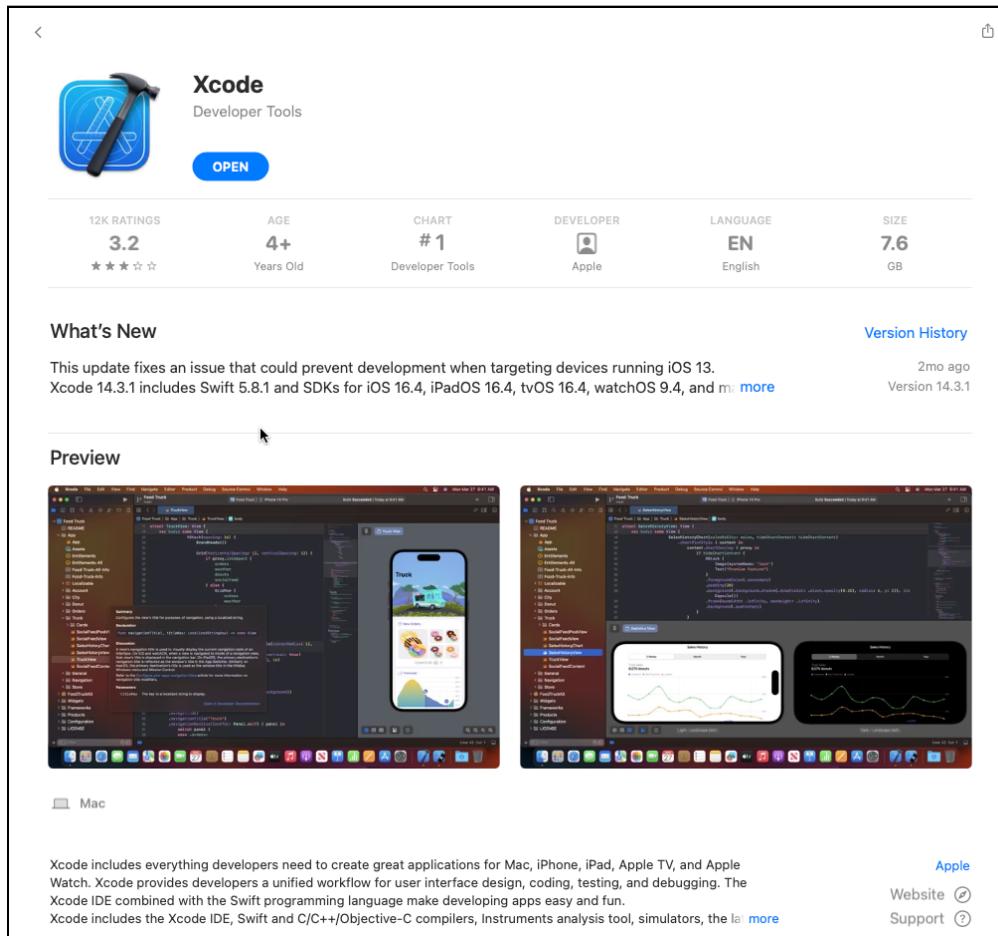


Fig. 1.6 – Xcode on the App Store

Installing CocoaPods

CocoaPods is a dependency manager for iOS. Since CocoaPods has been around for a long time, it's easy to use in Xcode and easy to add dependencies. You won't be using it but if you want to install it, follow the directions below.

If you are on an Intel based Mac, run the following command in Terminal:

```
sudo gem install cocoapods
```

The command above installs cocoapods using the default Ruby installation available on macOS.

If you are on an Apple silicon based Mac, run the following command instead:

```
brew install cocoapods
```

Verifying Using KDoctor

KDoctor is a tool developed by the Kotlin team to help set up the environment needed to develop apps using KMP.

KDoctor runs a series of checks to verify that Java, Android Studio, Xcode and Cocoapods are correctly installed and configured.

Install KDoctor using the following command:

```
brew install kdoctor
```

Once installed, run the following command:

```
kdoctor
```

Ensure that all steps are successful. If you had skipped installing Cocoapods in the previous section, the Cocoapods step of KDoctor will fail. This is fine considering that you won't be using Cocoapods for the projects in this book.

In case any other steps fail, KDoctor will provide you with the information needed to fix the issues.

Creating Your First Project

It's time to create your first project! In Android Studio, open the **File** menu and choose **New ▶ New Project**.

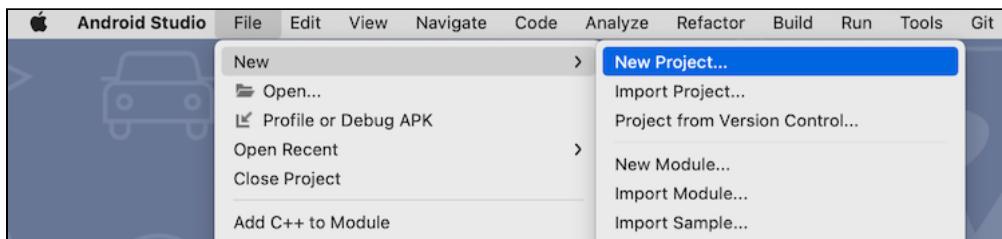


Fig. 1.7 – New Project Menu

In the New Project window, scroll down to the bottom and choose **Kotlin Multiplatform App**. If you don't see this, make sure you installed the Kotlin Multiplatform Mobile plugin. Also, make sure to restart Android Studio after installing the plugin. Click **Next**.

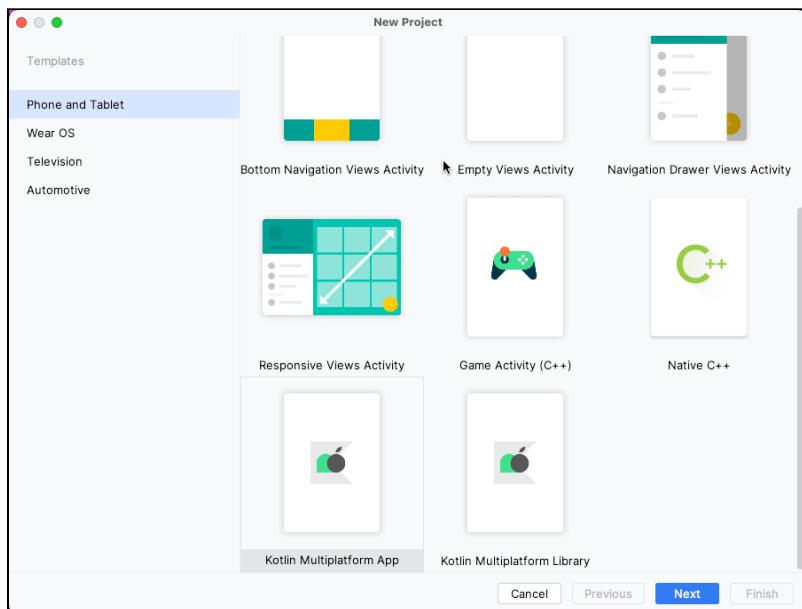


Fig. 1.8 – New Project Templates

In the next dialog, enter the name **Find Time** and a package name of **com.kodeco.findtime** or your own package name. Choose the directory you want to store the project and press **Next**.

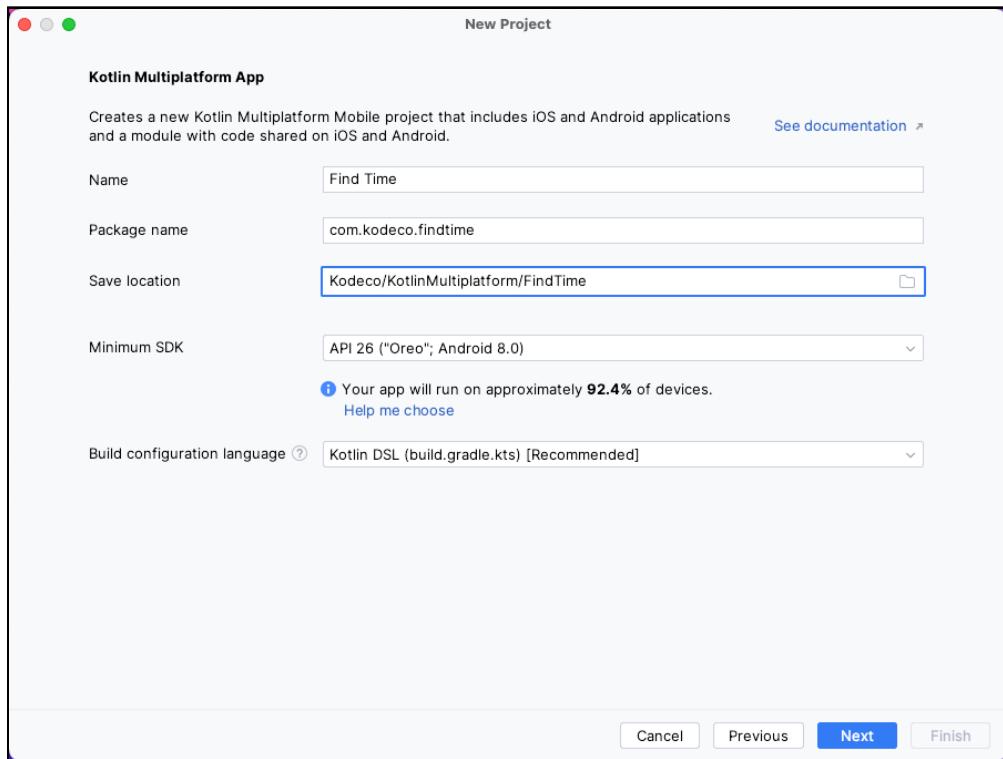


Fig. 1.9 – KMP Application Project Dialog

In the next dialog, leave everything as the default. Here, you’re naming the Android folder **androidApp**, the iOS folder **iosApp**, and the shared folder **shared**. You can use any names you want, but the rest of the book will use these conventions.

KMP now has a “Regular framework” for iOS. This is a bit simpler than CocoaPods, so keep “Regular framework”.

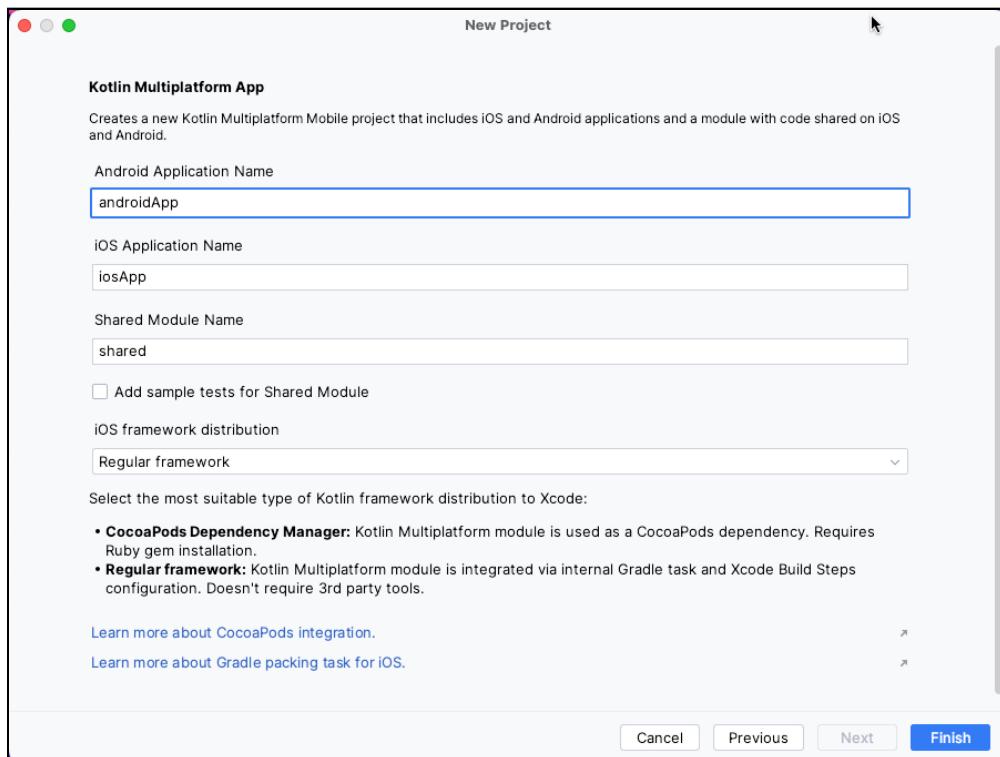


Fig. 1.10 – KMP Application Naming Dialog

Click **Finish**, and after a while, a new project will open.

You'll first see the Android file structure in the left panel, but you want to see all the folders. Choose **Project** from the menu showing **Android**. Here, you can see all the folders and files created for you. You have a hidden folder for Gradle and Android Studio (.idea), and the **androidApp**, **gradle**, **iosApp** and **shared** folders.

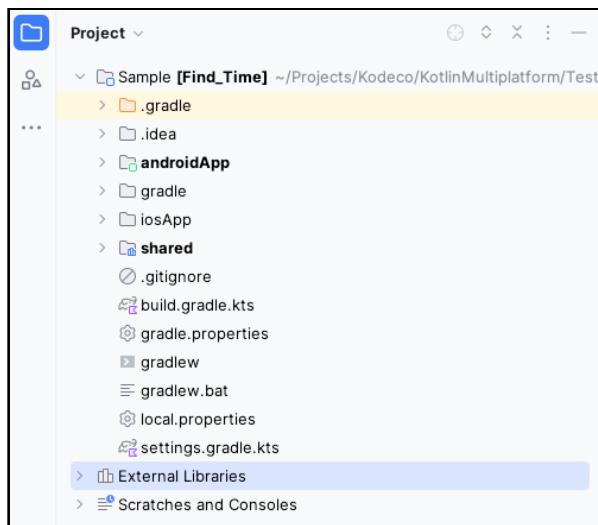


Fig. 1.11 – Project View

Kotlin Multiplatform Keywords

Now that you have the project created, you need to know about two new keywords that were added to the Kotlin language to support KMP: **expect** and **actual**. These let you create classes, functions, interfaces or variables in the shared module using the **expect** keyword. Those functions or variables aren't defined in the **commonMain** folder, but are *expected* in each multiplatform module — like **androidMain** or **iosMain**. **expect** and **actual** will be discussed in more detail in a later chapter. If you open the shared folder, you'll see:

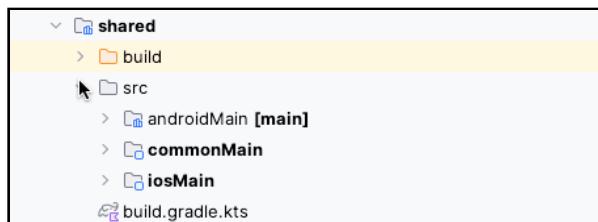


Fig. 1.12 – Shared Module

Here, you can see folders for Android, common and iOS. You'll add the shared classes and code to **commonMain/kotlin**. If you need to write code that is platform specific, you'll write an **expect** function or variable in the common folder and the **actual** code in both the Android and iOS folders. Open the **commonMain** folder and open **Platform.kt**.

```
package com.kodeco.findtime

interface Platform {
    val name: String
}

expect fun getPlatform(): Platform
```

Here, you see how to use the **expect** keyword. This says that you expect each platform to have a function named **getPlatform** that returns a **Platform** that has a name that is a string. Now, open the Android and iOS **Platform.kt** files.

Android

Open **androidMain/kotlin/com/kodeco/findtime/Platform.android.kt**.

```
package com.kodeco.findtime

class AndroidPlatform : Platform {
    override val name: String = "Android ${android.os.Build.VERSION.SDK_INT}"
}

actual fun getPlatform(): Platform = AndroidPlatform()
```

In the Android class, the new keyword **actual** states that this is the actual implementation for the **expected** **getPlatform** function. The **name** variable overrides the name variable and provides an implementation for the variable to return the string “Android” and the Android SDK number.

Run the Android app from Android Studio by making sure you have the **androidApp** selected in the toolbar and an emulator or phone selected. Then, click the green **Play** button.



Fig. 1.13 – Run Configuration

Note: If you get a build error about shared test files, open up **shared/build.gradle.kts** and comment out the **commonTest** section.

You'll see:

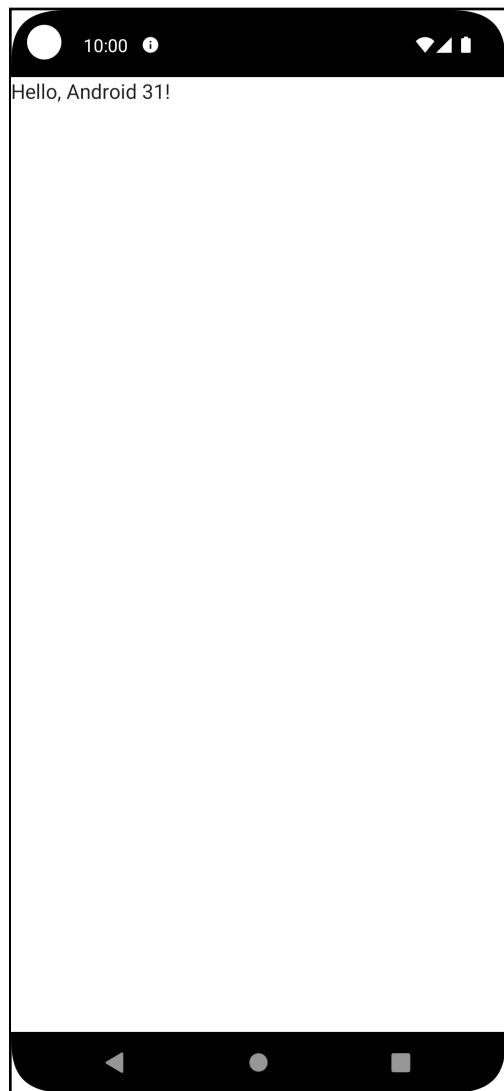


Fig. 1.14 – Android app running

Your screen now shows the words **Hello, Android** and the Android version.

iOS

Open `Platform.ios.kt` in `shared/src/iosMain/kotlin/com/kodeco/findtime`:

```
package com.kodeco.findtime

import platformUIKit.UIDevice

class IOSPlatform: Platform {
    override val name: String =
        UIDevice.currentDevice.systemName() + " " +
        UIDevice.currentDevice.systemVersion
}

actual fun getPlatform(): Platform = IOSPlatform()
```

Notice that this class is written in Kotlin but uses iOS platform code. Wow, you can write iOS code in Kotlin! When you build your project, the KMP plugin will compile this class into a framework. Open Xcode. From the **File** menu, choose **Open** and navigate to your project and into the **iosApp** folder. Select the workspace file (**iosApp.xcworkspace**).

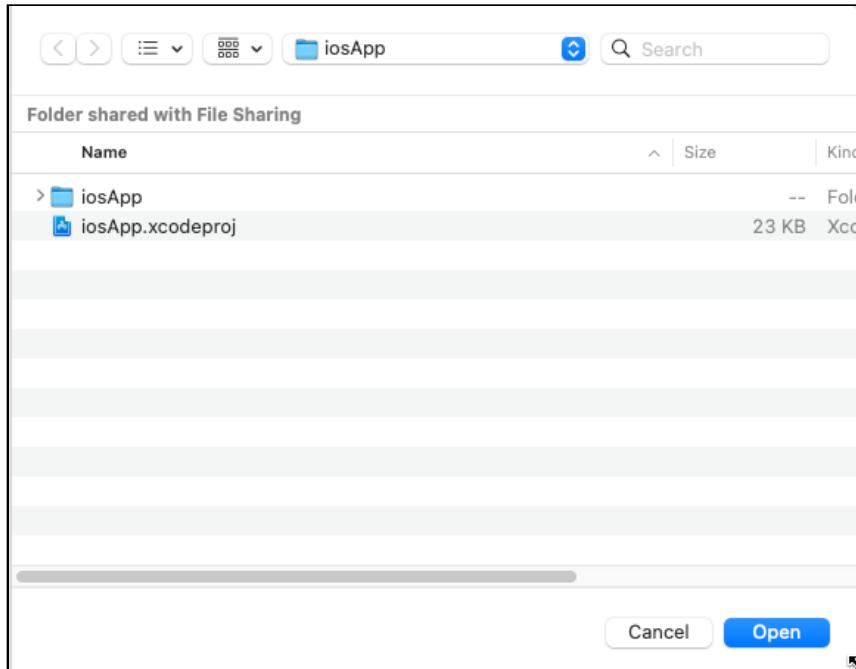


Fig. 1.15 – XCode Open Project

For the project to run without errors, you'll need to build in Xcode. Select **Product ▾ Build** or press **Command-B**. Once the project is built, open **ContentView.swift**.

```
import SwiftUI
import shared

struct ContentView: View {
    let greet = Greeting().greet()

    var body: some View {
        Text(greet)
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

This file has been generated for you and is written in SwiftUI. You'll get a crash course in SwiftUI in a later chapter. Hover over **greet()**, press **Command-Control** and click to jump to its definition.

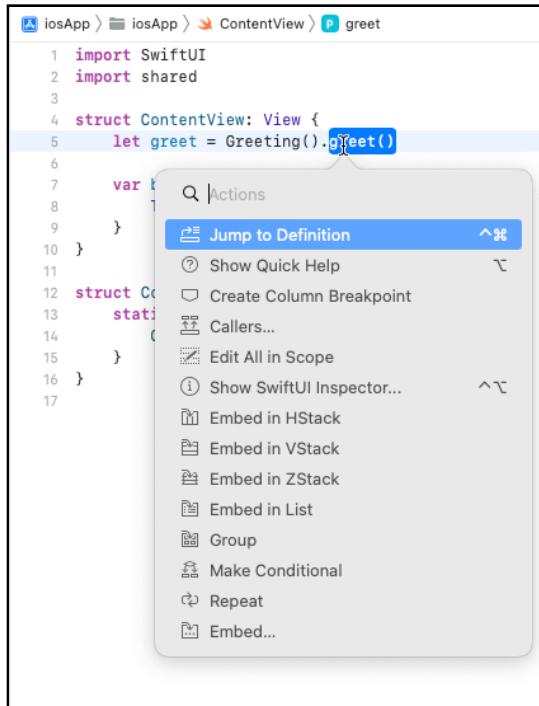


Fig. 1.16 – Xcode method definition

This will open the `shared.h` file. It's written in Objective-C, but it allows you to use all the code from the shared project. Scroll to the bottom of the file and you will see:

```
__attribute__((objc_subclassing_restricted))
__attribute__((swift_name("Greeting")))
@interface SharedGreeting : SharedBase
- (instancetype)init __attribute__((swift_name("init())))
__attribute__((objc_designated_initializer));
+ (instancetype)new __attribute__((availability(swift,
unavailable, message="use object initializers instead")));
- (NSString *)greet __attribute__((swift_name("greet())));
@end
```

Run the app in a simulator in Xcode by clicking **Play** or by pressing **Command-R**. You'll see:



Fig. 1.17 – iOS app running

The screen now shows the code written in Kotlin using the device name and the device version.

Key Points

- KMP refers to Kotlin Multiplatform.
- KMP helps write common code for **networking, database and business logic**.
- You can't use KMP for UI work. You'll need to use **native frameworks** instead.
- It's easy to create a KMP project for mobile by using the **Kotlin Multiplatform Mobile plugin**.

Where to Go From Here?

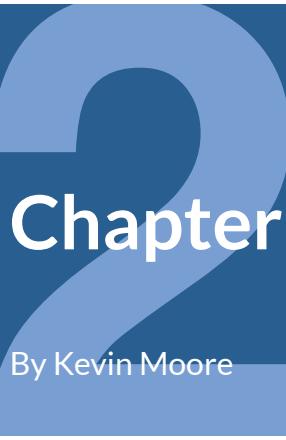
In this chapter, you've learned a bit about KMP.

- If you want to learn more, go to <https://kotlinlang.org/docs/multiplatform.html>
- For the latest in what's new with KMP, go to <https://kotlinlang.org/docs/multiplatform-mobile-getting-started.html>.
- To find the current state of each platform, go to <https://kotlinlang.org/docs/components-stability.html>.
- To see some of the top companies using KMP, go to <https://www.netguru.com/blog/top-apps-built-with-kotlin-multiplatform> and <https://kotlinlang.org/lp/multiplatform/case-studies/>.

To learn more about Kotlin:

- Kodeco.com Learning Paths at <https://www.kodeco.com/android/paths/learn>
- Kotlinlang Docs at <https://kotlinlang.org/docs/home.html>

In the next chapter, you'll build on this project to create the Find Time project.



Chapter 2: Getting Started

By Kevin Moore

In the last chapter, you created your first KMP project. To get started, you'll need to understand the build system KMP uses. For Android and desktop, that's Gradle. For iOS, Android Studio will use the “Regular Framework” for building a library.

Getting to Know Gradle

If you come from the Android world, you already have some experience with Gradle — but it's probably with the one written in the **Groovy** language. These files are named **build.gradle**. If you don't know Android, Gradle is a build system for developing software like in Android. It can run tasks to compile, package, test, deploy, and even publish artifacts to a distribution center. Many programming systems use it. Gradle simplifies the often complex process of building and managing projects by making use of the different plugins for almost any purpose.

For KMP, you'll use the Kotlin scripting version of Gradle. These files are named **build.gradle.kts**. The kts extension stands for "Kotlin script." This version uses Kotlin for the Gradle DSL (Domain Specific Language) — which makes it much easier to use if you already know Kotlin. This project has a number of these build scripts in different directories. Each serves a specific purpose. Open the starter project or the project from the last chapter in Android Studio, switch to the Project view on the left and follow along to learn more about these build files.

If you don't know what a particular command does, you can click the name while pressing the command key and Android Studio will open the class.

Version Catalog

One of the nice features of the newer Gradle versions is the concept of **Version Catalogs**. This is a file where you can define version, library, plugin and bundle variables. This allows you to have all of your versions defined in one place so that each build file uses the same one. This file was created for you when you created your project. Open **gradle/libs.versions.toml**. This file has four sections:

- versions
- libraries
- plugins
- bundles

The versions section is where you define the version numbers for libraries or plugins. The libraries section is where you define the library using the version defined above. Plugins are of course for plugins like Android, Compose, etc. Bundles are a very convenient section for defining bundles of libraries.

Compose is a great example as it needs a lot of libraries. Define them all in one bundle and you can add just the bundle to your dependency list. You will dig in a bit more about all of these 4 sections.

Versions

Let's start with the versions section. The first step is to add all of the versions for the libraries needed. Replace the current list of versions with the following:

```
[versions]
agp = "8.1.2"
kotlin = "1.9.10"
core-ktx = "1.12.0"
junit = "4.13.2"
androidx-test-ext-junit = "1.1.5"
espresso-core = "3.5.1"
appcompat = "1.6.1"
material = "1.6.0-alpha07"
compose-bom = "2023.10.00"
compose-ui = "1.5.3"
viewModelVersion = "2.6.2"
androidx-activity = "1.8.0"
activity-compose = "1.8.0"
navigation="2.7.4"
material3 = "1.2.0-alpha09"
kotlinxDateTime = "0.4.1"
napier = "2.6.1"
composePlugin = "1.5.1"
multiplatformPlugin = "1.9.10"
```

- agp: agp is short for Android Gradle Plugin
- compose-bom: This is the version needed for the BOM (bill of material) for compose.
- kotlinxDateTime: This is a library needed for dealing with dates & timezones.
- napier: Multiplatform logging library

The BOM is a way to insure that you are using all of the correct versions for each library. The different compose libraries are changing every month and keeping track of each library version is very difficult. By keeping track of just the version for the BOM, you no longer have to know what version numbers to use for each library. You only need to specify the BOM version.

Libraries

Now that you have the versions defined, list the libraries that will be used in the app. Replace the **libraries** section with:

```
[libraries]
core-ktx = { group = "androidx.core", name = "core-ktx",
version.ref = "core-ktx" }
junit = { group = "junit", name = "junit", version.ref = "junit"
}
androidx-test-ext-junit = { group = "androidx.test.ext", name =
"junit", version.ref = "androidx-test-ext-junit" }
espresso-core = { group = "androidx.test.espresso", name =
"espresso-core", version.ref = "espresso-core" }
appcompat = { group = "androidx.appcompat", name = "appcompat",
version.ref = "appcompat" }

# AndroidX
androidx-activity = { module = "androidx.activity:activity",
version.ref = "androidx-activity" }
androidx-activity-ktx = { module = "androidx.activity:activity-
ktx", version.ref = "androidx-activity" }

# Compose
compose-bom = { group = "androidx.compose", name = "compose-
bom", version.ref = "compose-bom" }
ui = { group = "androidx.compose.ui", name = "ui" }
ui-graphics = { group = "androidx.compose.ui", name = "ui-
graphics" }
ui-ui = { module = "androidx.compose.ui:ui", version.ref =
"compose-ui" }
ui-tooling = { group = "androidx.compose.ui", name = "ui-
tooling" }
ui-tooling-preview = { group = "androidx.compose.ui", name =
"ui-tooling-preview" }
activity-compose = { group = "androidx.activity", name =
"activity-compose", version.ref = "activity-compose" }
compose-viewmodel = {module = "androidx.lifecycle:lifecycle-
viewmodel-compose", version.ref="viewModelVersion" }
androidx-compose-foundation = { group =
"androidx.compose.foundation", name = "foundation" }
ui-test-manifest = { group = "androidx.compose.ui", name = "ui-
test-manifest" }
ui-test-junit4 = { group = "androidx.compose.ui", name = "ui-
test-junit4" }
navigation = { module = "androidx.navigation:navigation-compose",
version.ref="navigation" }

# Material
# Material design icons
material-icons = { module = "androidx.compose.material:material-
```

```
    icons-core", version.ref = "compose-ui" }
material-iconsExtended = { module =
    "androidx.compose.material:material-icons-extended", version.ref
    = "compose-ui" }
material3 = { group = "androidx.compose.material3", name =
    "material3" , version.ref="material3"}
material = { group = "androidx.compose.material", name =
    "material", version.ref="material" }

# DateTime
datetime = { module = "org.jetbrains.kotlinx:kotlinx-datetime",
version.ref = "kotlinxDateTime" }

#Napier
napier = { module = "io.github.aakira:napier", version.ref
= "napier" }
```

The top set of libraries are the same but the rest are new. Notice the makeup of the library:

```
<library-name> = { module = "<full module name>",
version.ref=<name of version>"}
or
<library-name> = { group = "<group name>", name=<library
name>, version.ref=<name of version>"}
```

These are two different ways to define a library. The first one uses the full name (including the “:”). The second version separates the group and name into separate parameters. The `version.ref` uses the string version name defined above.

Comments separate the different types of libraries. These are:

- AndroidX: Android specific libraries.
- Compose: There are a lot of these libraries. These libraries will be used to define the UI with the Composable functions available in the libraries.
- Material: Both Material & Material3 (newer) libraries.
- DateTime: Library for handling dates & times.
- Napier: For multiplatform logging.

Plugins

Now replace the plugins section with:

```
[plugins]
androidApplication = { id = "com.android.application",
version.ref = "agp" }
kotlinAndroid = { id = "org.jetbrains.kotlin.android",
version.ref = "kotlin" }
composePlugin = { id = "org.jetbrains.compose", version.ref =
"composePlugin" }
multiplatformPlugin = { id =
"org.jetbrains.kotlin.multiplatform", version.ref =
"multiplatformPlugin" }
```

This just adds two new plugins to our project: compose and multiplatform plugin. In general, a typical plugin declaration includes two components:

- **plugin name:** In the updated code, composePlugin and multiplatformPlugin are the names of the plugin that you will be using in the project to identify your plugin.
- **id:** The id (short for identifier) is used to specify which plugin you want to include.
- **version.ref:** The version reference allows you to specify the versions of the plugin. We have defined the versions required at the top in the **versions** section.

Bundles

The bundles section is where things get interesting. By creating a bundle you define one variable that can represent multiple libraries. Add the following at the bottom of the file:

```
[bundles]
androidx-activity = ["androidx-activity", "androidx-activity-
ktx", "compose-viewmodel"]
material = ["material-icons", "material-iconsExtended",
"material", "material3"]
compose-ui = ["activity-compose", "androidx-compose-foundation",
"ui", "ui-ui", "ui-tooling", "ui-tooling-preview"]
```

Notice how this is a list of libraries identified with a user-defined identifier. To use this in a gradle file you would just add this to the dependency section:

```
implementation(libs.bundles.compose.ui)
```

Notice that “-” is replaced by “.”. Now you’re done making updates in the build files. Press the **Sync Now** text at the top of the file to sync changes into the project.

Build Files

Open **build.gradle.kts** in the root folder. Replace the current plugins with:

```
alias(libs.plugins.androidApplication) apply false
alias(libs.plugins.kotlinAndroid) apply false
alias(libs.plugins.multiplatformPlugin) apply false
```

Notice that there aren’t any versions here. The `apply false` means that it does not use the plugin in this file. You want to use `false` in the top-level Gradle file. Press the **Sync Now** text at the top of the file.

There will be the following build files:

- Root level **build.gradle.kts**. This just has a few plugins & the clean task
- Root level **settings.gradle.kts**. This is where all of the settings for the Gradle project are stored. You set the project name and include any sub-projects here. Also, a good place to define where Gradle should fetch the plugins from.
- **androidApp/build.gradle.kts**: This defines the build for Android.
- **shared/build.gradle.kts**: This defines the build for shared files. Because of this, it needs to define all the targets used. (Android, iOS, desktop, etc).

Notice that iOS does not have a gradle file.

Android Build File

Now, open **androidApp/build.gradle.kts**. Except for the different Kotlin syntax, this should look familiar to Android developers. Replace the current plugins with:

```
alias(libs.plugins.androidApplication)
alias(libs.plugins.kotlinAndroid)
```

Android needs the application and the Android Kotlin plugins. Normally, you’ll see either `id` or `kotlin` keywords. Using the newer `alias` keyword, you can reference your version catalog entry.

Next, update the Android-specific settings. Change the following:

- `kotlinCompilerExtensionVersion` to **1.5.3**.
- Java version to **17**
- Kotlin `jvmTarget` to Java 17
- Kotlin compiler flags: These remove the need to add the experimental annotations all around the code (which can get very annoying)

```
// 1
android {
    // 2
    namespace = "com.kodeco.findtime.android"
    // 3
    compileSdk = 34
    defaultConfig {
        // 4
        applicationId = "com.kodeco.findtime.android"
        // 5
        minSdk = 26
        targetSdk = 34
        versionCode = 1
        versionName = "1.0"
    }
    // 6
    buildFeatures {
        compose = true
    }
    // 7
    composeOptions {
        kotlinCompilerExtensionVersion = "1.5.3"
    }
    // 8
    packaging {
        resources {
            excludes += "/META-INF/{AL2.0,LGPL2.1}"
        }
    }
    // 9
    buildTypes {
        getByName("release") {
            isMinifyEnabled = false
        }
    }
    // 10
    compileOptions {
        sourceCompatibility = JavaVersion.VERSION_17
        targetCompatibility = JavaVersion.VERSION_17
    }
    // 11
```

```
kotlinOptions {  
    jvmTarget = JavaVersion.VERSION_17.toString()  
    freeCompilerArgs = freeCompilerArgs + listOf(  
        "-opt-  
        in=androidx.compose.animation.ExperimentalAnimationApi",  
        "    "-opt-  
        in=androidx.compose.ui.ExperimentalComposeUiApi",  
        "    "-opt-  
        in=androidx.compose.material.ExperimentalMaterialApi",  
        "    "-opt-  
        in=androidx.lifecycle.viewmodel.compose.SavedStateHandleSaveable  
        Api",  
        "    "-opt-  
        in=androidx.compose.material3.ExperimentalMaterial3Api",  
        "    )  
    }  
}
```

Here's what the above code does:

1. Start the Android section.
2. Define the app namespace. This will be your app's unique id. (Note that this is a new requirement. The applicationID was used before.)
3. **compileSdk** specifies the Android SDK version to compile against.
4. **applicationId** is the ID for the Android App. This has to be unique for every app on the Google Play store.
5. **minSdk** refers to the lowest Android version your app will run on, while **targetSdk** refers to the latest Android version you support. **versionCode** is the number you'll use internally to differentiate between builds while **versionName** is the version that will be displayed on the Play Store.
6. Specify compose as a build feature. This enables the use of Jetpack Compose in the project.
7. Set the Compose compiler version.
8. Set up any packaging options. This enables you to customize the packaging of resources by specifying the exclusion rules for specific files.
9. Specify debug or release settings here. For the release version, this sets **isMinifyEnabled** to false, but you'll probably want to set it to true when you're ready to release your app. See: [https://developer.android.com/reference/tools/gradle-api/8.2/com/android/build/api/dsl/BuildType#isMinifyEnabled\(\)](https://developer.android.com/reference/tools/gradle-api/8.2/com/android/build/api/dsl/BuildType#isMinifyEnabled().).

10. Set the Java version compatibility to 17.
11. Target compile version of 17. Add the flags so that you don't need to add the experimental annotations.

The next section shows which dependencies you need:

```
dependencies {  
    // 1  
    implementation(project(":shared"))  
    // 2  
    implementation(platform(libs.compose.bom))  
    implementation(libs.bundles.compose.ui)  
    implementation(libs.bundles.androidx.activity)  
    implementation(libs.bundles.material)  
    implementation(libs.napier)  
}
```

1. Android depends on the `shared` module (where all shared business logic will reside).
2. Android-specific libraries (Compose libraries).

Notice this line:

```
implementation(platform(libs.compose.bom))
```

This adds the Compose BoM file which syncs the correct versions of all of the Compose libraries.

The bundle below adds all the compose libraries.

```
implementation(libs.bundles.compose.ui)
```

Now, press the **Sync Now** text at the top of the file.

Shared Build File

Open `shared/build.gradle.kts`. This is the build script for the shared module. If you open the `src` directory, you'll see the `androidMain`, `commonMain` and `iosMain` directories. These contain the shared files for Android and iOS, as well as files that all modules share.

The first section is the plugins:

```
plugins {
    kotlin("multiplatform")
    id("com.android.library")
}
```

The first plugin is for KMP and defines this module as a multiplatform module. The second plugin is for Android. You'll use this to create an Android library for use in an Android app.

The Kotlin section is next. This section uses the multiplatform plugin above to configure this module for KMP. Change it to the following:

```
kotlin {
    // 1
    androidTarget {
        compilations.all {
            kotlinOptions {
                jvmTarget = JavaVersion.VERSION_17.toString()
            }
        }
    }

    // 2
    ios()
    iosSimulatorArm64()

    // 3
    listOf(
        iosX64(),
        iosArm64(),
        iosSimulatorArm64()
    ).forEach {
        it.binaries.framework {
            basePath = "shared"
        }
    }

    // 4
    jvm("desktop")

    // 5
    sourceSets {
        val commonMain by getting {
            kotlin.srcDirs("src/commonMain/kotlin")
            dependencies {
                implementation(libs.datetime)
                implementation(libs.napier)
            }
        }
    }
}
```

```
    val androidMain by getting {
        kotlin.srcDirs("src/androidMain/kotlin")
    }
    val iosMain by getting {
        kotlin.srcDirs("src/iosMain/kotlin")
    }
    val iosTest by getting
    val iosSimulatorArm64Main by getting
    val iosSimulatorArm64Test by getting

}
```

Remember to remove the `@OptIn` and `targetHierarchy.default()` blocks of code.

Here's an explanation of the code:

1. Use the **androidTarget** method to define an Android target.
2. Define iOS targets.
3. **iosX64** defines a target for the iOS simulator on x86_64 platforms, while **iosArm64** defines a target for iOS on ARM64 platforms. This will create a shared library with the name "shared"
4. Defines a desktop version.

Next, change the Android section to the following:

```
android {
    namespace = "com.kodeco.findtime"
    compileSdk = 34
    defaultConfig {
        minSdk = 26
    }
    compileOptions {
        sourceCompatibility = JavaVersion.VERSION_17
        targetCompatibility = JavaVersion.VERSION_17
    }
}
```

This is similar to Android's build file, except it just has the minimum information needed.

Do a Gradle sync to make sure everything still works.

Build and run the app on Android. You'll see a screen similar to the one shown below:

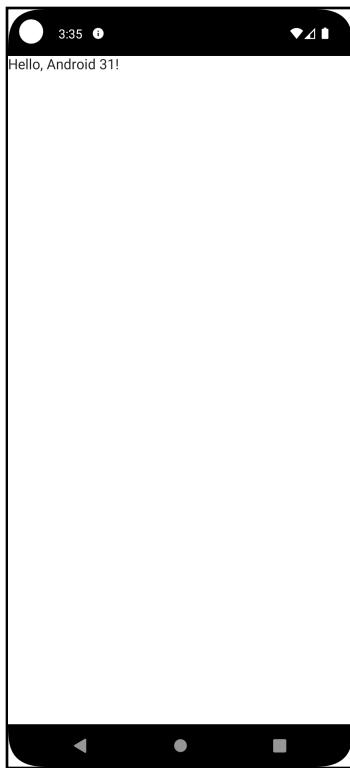


Fig. 2.1 – Android app running fine

You haven't changed the UI yet, but this ensures that the Gradle files are still working.

Find Time

Have you ever needed to schedule a meeting with colleagues who work in different time zones? It can be a real pain. Are they awake at the hour you want to schedule? What are good times to schedule a meeting? You're going to write the **Find Time** app to help find those hours that work best. To do that, you need to write some time zone logic to figure out the best hours to meet. If you were to write separate apps for iOS and Android, you would have to write that business logic twice.

Business Logic

One of the main benefits of KMP is you can share business logic among all your platforms. You'll write your business logic in the shared module. This module is a multiplatform module you can use for iOS, Android, desktop and the web.

Open the **shared/src** folder and then **androidMain**, **commonMain** and **iosMain** folders. These are directories for:

- Android
- Shared
- iOS

You'll find the **Platform** and **Greeting** classes in these folders. Delete these classes as you won't use them. Note that both the Android and iOS platforms won't run until you delete the code that called them. Android Studio will complain about their usages. Click the **View Usages** button.

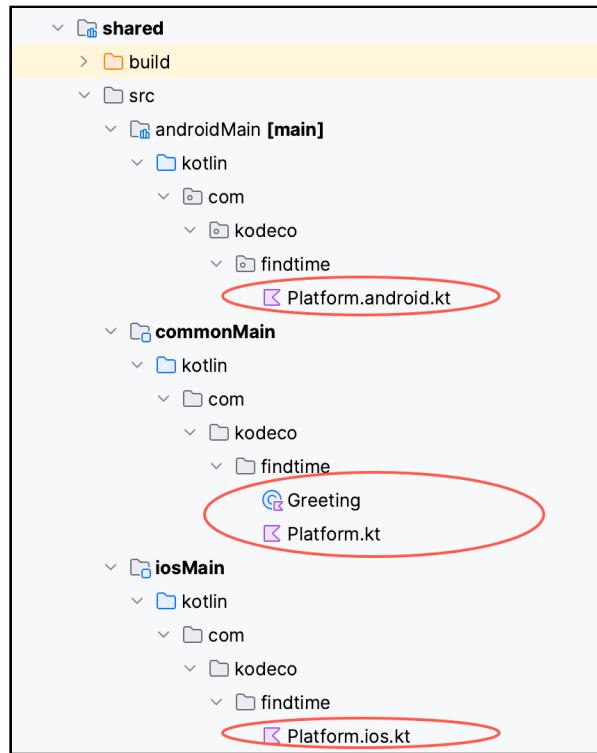


Fig. 2.2 – Usages of the Greeting class

Double-click `GreetingView(Greeting().greet())` to open `MainActivity`. Delete the `GreetingView` function, the imports and the reference to it below:

```

import com.kodeco.findtime.Greeting

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            MyApplicationTheme {
                Surface(
                    modifier = Modifier.fillMaxSize(),
                    color = MaterialTheme.colors.background
                ) {
                    GreetingView(Greeting().greet())
                }
            }
        }
    }

    @Composable
    fun GreetingView(text: String) {
        Text(text = text)
    }

    @Preview
    @Composable
    fun DefaultPreview() {
        MyApplicationTheme {
            GreetingView( text: "Hello, Android!")
        }
    }
}

```

Fig. 2.3 – Remove references to the Greeting class

Run the delete again, and you'll be able to delete the files without any problems.

Date Time Calculations

You'll use JetBrains' `kotlinx-datetime` library to help with datetime calculations. Open `shared/build.gradle.kts` and find `val commonMain`. You can see that you added the `datetime` and `napier` libraries:

```

val commonMain by getting {
    dependencies {

```

```
// 1  
implementation(libs.datetime)  
// 2  
implementation(libs.napier)  
}  
}
```

This will import two libraries:

1. JetBrains datetime library.
2. Napier logging library.

DateTime Library

Kotlin's `kotlinx-datetime` is an easy-to-use multiplatform library that helps with date- and time-based calculations. It uses several data types:

- `Instant` represents a moment in time.
- `Clock` imitates a real-world clock and provides the current instant.
- `LocalDateTime` represents a date with time but no associated time zone.
- `LocalDate` represents only a date.
- `TimeZone` and `ZoneOffset` help you convert between `Instant` and `LocalDateTime`.
- `Month` is an enum representing all months in the year.
- `DayOfWeek` is an enum representing all days of the week. It uses values like `MONDAY`, `TUESDAY`, etc. instead of integers.
- `DateTimePeriod` represents the difference between 2 instants.
- `DatePeriod` is a subclass of `DateTimePeriod` and represents the difference between two `LocalDate` instances.
- `DateTimeUnit` provides a set of units such as `NANOSECOND`, `WEEK`, `CENTURY`, etc. that you can use to perform arithmetic operations on `Instant` and `LocalDate`.

Time Zone Helper

Go to the `com/kodeco/findtime` package in `shared/commonMain` and create a new Kotlin interface named `TimeZoneHelper`. Add the following:

```
interface TimeZoneHelper {  
    fun getTimeZoneStrings(): List<String>  
    fun currentTime(): String  
    fun currentTimeZone(): String  
    fun hoursFromTimeZone(otherTimeZoneId: String): Double  
    fun getTime(timezoneId: String): String  
    fun getDate(timezoneId: String): String  
    fun search(startHour: Int, endHour: Int, timezoneStrings:  
              List<String>): List<Int>  
}
```

This defines an interface that has seven functions.

- Return a list of time zone strings. (This is a list of all time zones from the JetBrains kotlinx-datetime library)
- Return the current formatted time.
- Return the current time zone id.
- Return the number of hours from the given time zone.
- Return the formatted time for the given time zone.
- Return the formatted date for the given time zone.
- Search for a list of hours that start at `startHour`, end at `endHour` and are in all the given time zone strings.

Creating an interface makes it easy to test. This chapter doesn't cover tests, but using an interface makes it easy to create mocked time zone helpers. Now, create an instance of that interface. Right-click the `findtime` folder and create a new Kotlin class named `TimeZoneHelperImpl`. This class will implement the interface. Update the class to extend `TimeZoneHelper` as follows:

```
class TimeZoneHelperImpl: TimeZoneHelper {  
}
```

You'll see a red line underneath the class because you haven't yet implemented the methods that are defined in `TimeZoneHelper` interface class. Press **Option-Return** while keeping your cursor on `TimeZoneHelperImpl` and choose **Implement Members**.

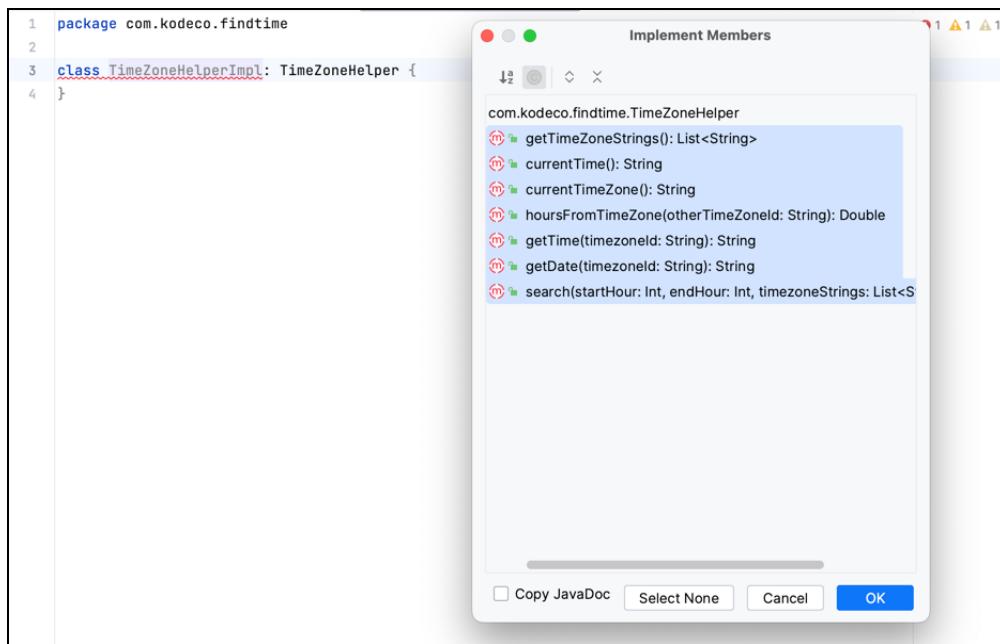


Fig. 2.4 – Implement all methods of `TimeZoneHelper`

Select all and click **OK**. You'll see lots of TODOs.

Start with `getTimeZoneStrings`. This sounds like it could be really hard, but the `kotlinx-datetime` library makes it easy. Replace the TODO with:

```
return TimeZone.availableZoneIds.sorted()
```

This line returns the available time zone IDs and sorts them. `TimeZone` will be red. Place your cursor on `TimeZone` and press **Option-Return** to import the library. You can use this technique to import the required classes in the next sections as well.

Next, you need a method to format datetime's `LocalDateTime` class. Add the following method at the bottom of the class:

```
fun formatDateDateTime(dateTime: LocalDateTime): String {
    // 1
    val stringBuilder = StringBuilder()
    // 2
```

```
    val minute = dateTime.minute
    var hour = dateTime.hour % 12
    if (hour == 0) hour = 12
    // 3
    val amPm = if (dateTime.hour < 12) " am" else " pm"
    // 4
    stringBuilder.append(hour.toString())
    stringBuilder.append(":")
    // 5
    if (minute < 10) {
        stringBuilder.append('0')
    }
    stringBuilder.append(minute.toString())
    stringBuilder.append(amPm)
    // 6
    return stringBuilder.toString()
}
```

In the code above, you:

1. Use a `StringBuilder` to build the string piece by piece.
2. Get the hour and minutes from the `dateTime` argument.
3. Since you want a string with am/pm, check if the hour is greater than noon (12).
4. Build the hour and colon.
5. Check to make sure numbers 0-9 are padded.
6. Return the final string.

Get rid of the “Unresolved reference: `LocalDateTime`” by placing your cursor on `LocalDateTime` and pressing **Option-Return** to import the library.

Now update the `currentTime` method to the below code and import the required libraries:

```
override fun currentTime(): String {
    // 1
    val currentMoment: Instant = Clock.System.now()
    // 2
    val dateTime: LocalDateTime =
    currentMoment.toLocalDateTime(TimeZone.currentSystemDefault())
    // 3
    return formatDateTime(dateTime)
}
```

In the previous code, you:

1. Get the current time as an Instant.
2. Convert the current moment into a LocalDateTime that's based on the current user's time zone.
3. Format the given date using the `formatDateTime` method you defined earlier.

Now implement the `getTime` method:

```
override fun getTime(timezoneId: String): String {  
    // 1  
    val timezone = TimeZone.of(timezoneId)  
    // 2  
    val currentMoment: Instant = Clock.System.now()  
    // 3  
    val dateTime: LocalDateTime =  
        currentMoment.toLocalDateTime(timezone)  
    // 4  
    return formatDateTime(dateTime)  
}
```

In the code above, you:

1. Get the time zone with the given ID.
2. Get the current time as an Instant.
3. Convert the current moment into a LocalDateTime that's based on the passed-in time zone.
4. Format the given date.

`getDate` is similar to `getTime`. Replace `getDate` with the following code:

```
override fun getDate(timezoneId: String): String {  
    val timezone = TimeZone.of(timezoneId)  
    val currentMoment: Instant = Clock.System.now()  
    val dateTime: LocalDateTime =  
        currentMoment.toLocalDateTime(timezone)  
    // 1  
    return "$  
        ${dateTime.dayOfWeek.name.lowercase().replaceFirstChar{ it.uppercase() }}, "  
        "+  
        "${dateTime.month.name.lowercase().replaceFirstChar{ it.uppercase() }} ${dateTime.date.dayOfMonth}"  
}
```

This takes the different parts of the `DateTime` to create a string like: “Monday, October 4.”

The `currentTimeZone` method is pretty easy. Return the current time zone as a string:

```
override fun currentTimeZone(): String {
    val currentTimeZone = TimeZone.currentSystemDefault()
    return currentTimeZone.toString()
}
```

The `hoursFromTimeZone` method is a bit tricky. You want to return the number of hours from the given time zone:

```
override fun hoursFromTimeZone(otherTimeZoneId: String): Double
{
    // 1
    val currentTimeZone = TimeZone.currentSystemDefault()
    // 2
    val currentUTCInstant: Instant = Clock.System.now()
    // Date time in other timezone
    // 3
    val otherTimeZone = TimeZone.of(otherTimeZoneId)
    // 4
    val currentDateTime: LocalDateTime =
    currentUTCInstant.toLocalDateTime(currentTimeZone)
    // 5
    val currentOtherDateTime: LocalDateTime =
    currentUTCInstant.toLocalDateTime(otherTimeZone)
    // 6
    return abs((currentDateTime.hour -
    currentOtherDateTime.hour) * 1.0)
}
```

In the code above, you:

1. Get the current time zone.
2. Get the current time/instant.
3. Get the other time zone.
4. Convert the current time into a `LocalDateTime` class.
5. Convert the current time in another time zone into a `LocalDateTime` class.
6. Return the absolute difference between the hours (shouldn’t be negative), making sure the result is a double.

Searching

Searching is a bit harder. Given a starting hour (like 8 a.m.), an ending hour (say 5 p.m.) and the list of time zones that everyone is in, you want to return a list of integers that represent the hours (0-23) that fit in everyone's time zones. So, if you pass in 8 a.m. - 5 p.m. for Los Angeles and New York, you will get a list of hours:

```
[8, 9, 10, 11, 12, 13, 14]
```

All these hours for Los Angeles also work for New York. Los Angeles can go up to 2 p.m. (14), while New York will start at 11 a.m. and go until 5 p.m. To see if an hour is valid, add the `isValid` method after the `search` method:

```
private fun isValid(
    timeRange: IntRange,
    hour: Int,
    currentTimeZone: TimeZone,
    otherTimeZone: TimeZone
): Boolean {
    if (hour !in timeRange) {
        return false
    }
    // TODO: Add Current Time
}
```

This method takes a time range (like 8..17), the given hour to check, the current time zone for the user and the other time zone that you're checking against. The first check verifies if the hour is in the time range. If not, it isn't valid.

Now, replace `// TODO: Add Current Time` with:

```
// 1
val currentUTCInstant: Instant = Clock.System.now()
// 2
val currentOtherDateTime: LocalDateTime =
    currentUTCInstant.toLocalDateTime(otherTimeZone)
// 3
val otherDateTimeWithHour = LocalDateTime(
    currentOtherDateTime.year,
    currentOtherDateTime.monthNumber,
    currentOtherDateTime.dayOfMonth,
    hour,
    0,
    0,
    0
)
// TODO: Add Conversions
```

1. Use datetime's `Clock.System.now` method to get the current instant in the UTC time zone.
2. Convert the instant into another time zone with `toLocalDateTime`, passing in the other time zone.
3. Get a `LocalDateTime` with the given hour. (Minutes, seconds and nanoseconds aren't needed)

Now, replace // TODO: Add Conversions with:

```
// 1
val localInstant =
    otherDateTimeWithHour.toInstant(currentTimeZone)
// 2
val convertedTime = localInstant.toLocalDateTime(otherTimeZone)
Napier.d("Hour $hour in Time Range ${otherTimeZone.id} is ${
    convertedTime.hour}")
// 3
return convertedTime.hour in timeRange
```

Napier is the logging library and needs to be imported. Place your cursor on Napier and press **Option-Return** on it to import the library.

In the previous code, you:

1. Convert that hour into the current time zone.
2. Convert your time zone hour to the other time zone.
3. Check to see if it's in your time range.

Now that you have the `isValid` method, the `search` method won't be as hard. You just need to go through all the given time zones and hours and check if they are valid. Update the `search` method with:

```
// 1
val goodHours = mutableListOf<Int>()
// 2
val timeRange = IntRange(max(0, startHour), min(23, endHour))
// 3
val currentTimeZone = TimeZone.currentSystemDefault()
// 4
for (hour in timeRange) {
    var isGoodHour = false
    // 5
    for (zone in timezoneStrings) {
        val timezone = TimeZone.of(zone)
        // 6
```

```
    if (timezone == currentTimeZone) {
        continue
    }
    // 7
    if (!isValid(
            timeRange = timeRange,
            hour = hour,
            currentTimeZone = currentTimeZone,
            otherTimeZone = timezone
        )
    ) {
        Napier.d("Hour $hour is not valid for time range")
        isGoodHour = false
        break
    } else {
        Napier.d("Hour $hour is Valid for time range")
        isGoodHour = true
    }
}
// 8
if (isGoodHour) {
    goodHours.add(hour)
}
// 9
return goodHours
```

In this code, you:

1. Create a list to return all the valid hours.
2. Create a time range from start to end hours.
3. Get the current time zone.
4. Go through each hour in the time range.
5. Go through each time zone in the time zone list.
6. If it's the same time zone as the current one, then you know it's good.
7. Check if the hour is valid.
8. If, after going through every hour and it's a good hour, add it to our list.
9. Return the list of hours.

Import the `min` and `max` methods. You've now written the business logic for the Time Finder app! Android, iOS, desktop and web platforms can all share it.

Build the app to make sure it still compiles. You can try to run both Android and iOS again.

To run iOS, you'll have to edit the iOS build configuration and pick a simulator:

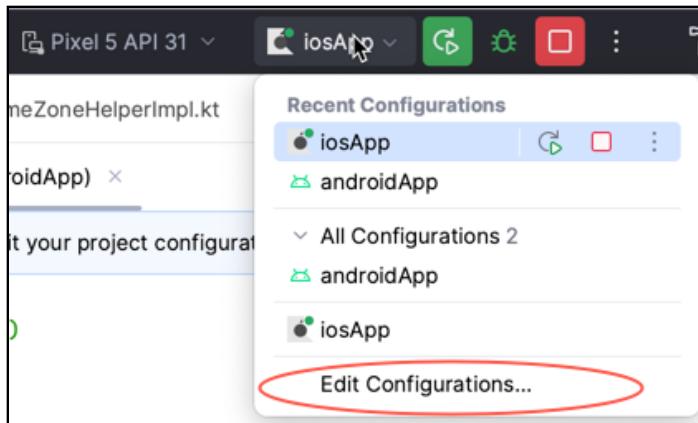


Fig. 2.5 – Edit Configuration

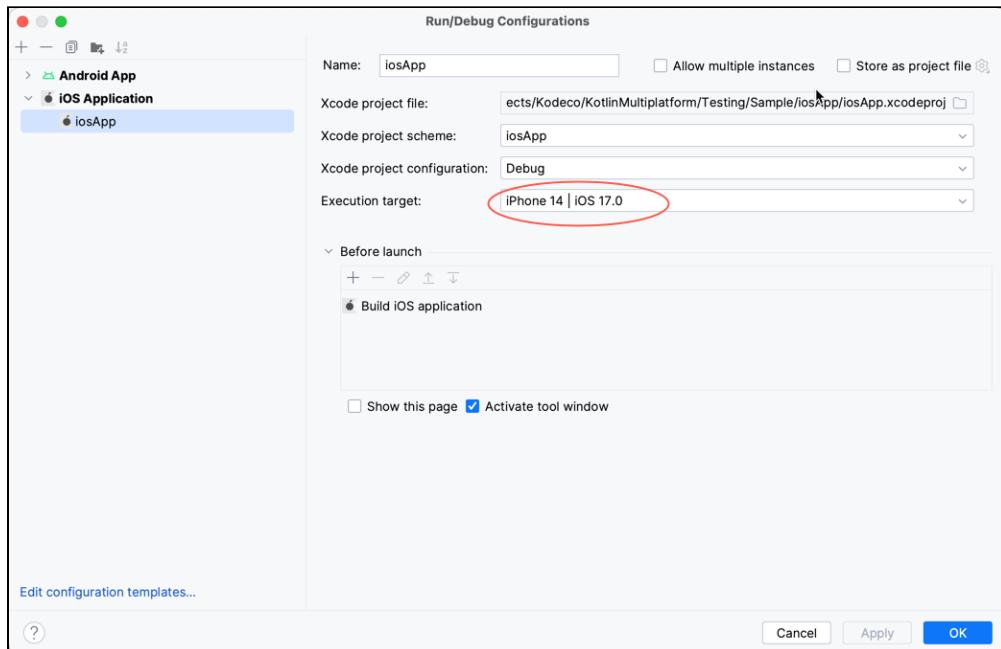
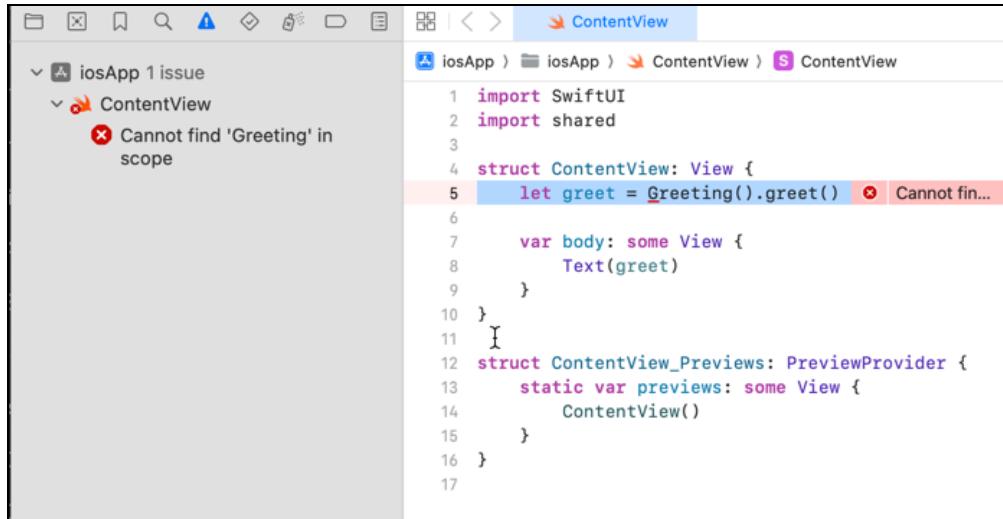


Fig. 2.6 – iOS Run Config

You'll see the iOS build fail with the error shown below if run from Xcode:



The screenshot shows the Xcode interface with the project navigation on the left and the code editor on the right. In the code editor, line 5 of the ContentView.swift file is highlighted in red, indicating a syntax error. The error message 'Cannot find 'Greeting' in scope' is displayed next to the line. The code itself is a simple SwiftUI struct definition.

```
1 import SwiftUI
2 import shared
3
4 struct ContentView: View {
5     let greet = Greeting().greet() ✘ Cannot fin...
6
7     var body: some View {
8         Text(greet)
9     }
10 }
11
12 struct ContentView_Previews: PreviewProvider {
13     static var previews: some View {
14         ContentView()
15     }
16 }
17
```

Fig. 2.7 – Xcode build error details

Remember that you removed the `greet` method. How would you fix this?

Challenge

The iOS app no longer works. Figure out how to find the problem, fix the error and get the iOS App working again. To see the answer, review the file in the challenge folder.

Key Points

- Gradle is the build system for most of KMP projects.
- You write the Gradle build files in Kotlin.
- You use the **libs.versions.toml** file to define variables, libraries, plugins and bundles.
- You write the business logic in the **shared** module.
- The **kotlinx-datetime** library is a multiplatform library for handling dates and times.

Where to Go From Here?

In this chapter, you've learned how to set up your project with the **libs.versions.toml** file to make changing versions easier. You also learned how to work with dates and times in the **shared** module, making it available for all platforms.

- Gradle: <https://gradle.org/>
- Kotlinx Datetime: <https://github.com/Kotlin/kotlinx-datetime>
- Version Catalogs: <https://docs.gradle.org/current/userguide/platforms.html> and <https://developer.android.com/build/migrate-to-catalogs>

In the next chapter, you'll start building the UI for the Find Time project.

3 Chapter 3: Developing UI: Android Jetpack Compose

By Kevin Moore

In the last chapter, you learned about the KMP build system. In this chapter, you'll learn about a new UI toolkit named **Jetpack Compose** that you can use on Android. This won't be an extensive discussion on Jetpack Compose, but it will teach you the basics. Open the starter project from this chapter. It contains the starter code that you'll use while going through this chapter.

UI Frameworks

At the time of writing this book, KMP doesn't provide a stable framework for developing a UI on cross-platform devices, so it is recommended to use each platform's native framework until it is stable. In this chapter, you'll learn about writing the UI for Android with Jetpack Compose. In the next chapter, you'll learn about building the UI for iOS using SwiftUI, which also works on macOS.

Note: You can use Compose to build UI for iOS as well but since its support is in alpha at the moment of writing this book. Therefore, this book will focus on using SwiftUI for building iOS UI.

Current UI System

On Android, you typically use an XML layout system for building your UIs. While Android Studio does provide a UI layout editor, it still uses XML underneath. This means that Android will have to parse XML files to build its view classes to then build the UI. What if you could just build your UI in code?

Jetpack Compose

That's the idea behind Jetpack Compose (JC). JC is a declarative UI system that uses functions to create all or part of your UI. The developers at Google realized the Android View system was getting older and had many flaws. So, they decided to come up with a whole new framework that would use a library instead of the built-in framework — allowing app developers to continue to provide the most up-to-date version of the framework regardless of the Android version.

One of the main tenets of Compose is that it takes less code to do the same things as the old View system. For example, to create a modified button, you don't have to subclass `Button` — instead, just add modifiers to an existing Compose component.

Compose components are also easily reusable. You can use Compose with new projects, and you can use it with existing projects that just use Compose in new screens. Compose can preview your UI in Android Studio, so you don't have to run the app to see what your components will look like. In a declarative UI, the UI will be drawn with the current state. If that state changes, the areas of the screen that have changed will be rerendered. This makes your code much simpler because you only have to draw what's in your current state and don't have to listen for changes.

Getting to Know Jetpack Compose

The one Android component that's still needed in Jetpack Compose (JC) is the `Activity` class. There has to be a starting point, and there's usually one `Activity` that's the main entry point. One of the nice features of JC is that you don't need more than one `Activity` (you can have more if you want to). Also — and more importantly — you don't need to use fragments anymore. If you're familiar with activities, you know that the starting method is `onCreate`. You no longer need to call `setContentView` because you won't be using XML files. Instead, you use `setContent`.

MainActivity

Open `MainActivity` in `androidApp/src/main/java/com/kodeco/findtime/android/`. Your code should look something like:

```
// 1
setContent {
    // 2
    MyApplicationTheme {
        // 3
        Surface(
            modifier = Modifier.fillMaxSize(),
            color = MaterialTheme.colorScheme.background
        ) {
            // 4
            Text("Hello, Android!")
        }
    }
}
```

Here's what the code does:

1. Use the `setContent` function to set the UI for the activity.
2. Set the theme (this was created for you when you created the project). This provides the colors, typography and shapes.
3. Fill the screen with the background color.
4. Use a text to say “Hello, Android!”.

If you look at the source of `setContent`, you'll see that it's an extension method on `ComponentActivity`. The last parameter in this method is your UI. This method is of type `@Composable`, which is a special annotation that you'll need to use on all of your Compose functions. A Compose function will look something like this:

```
@Composable  
fun showName(text: String) {  
    Text(text)  
}
```

The most important part is the `@Composable` annotation. This tells JC this is a function that can be drawn on the screen. No Composable function returns a value. Importantly, you want most of your functions to be stateless. This means that you pass in the data you want to show, and the function doesn't store that data. This makes the function very fast to draw.

Time Finder

You're going to develop a multiplatform app that will allow the user to select multiple time zones and find the best meeting times that work for all people in those time zones. Here's what the first screen looks like:

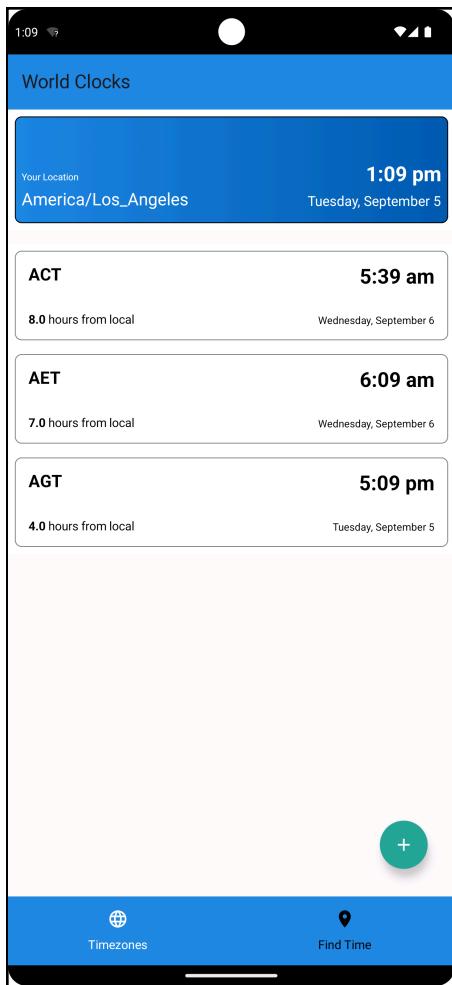


Fig. 3.1 — List of selected time zones

Here, you see the local time zone, time and date. Several different time zones are below that. Your user is trying to find a meeting time in all these locations.

Note: This is just the raw time zone string code. If you're interested, you can challenge yourself to replace the string codes with more readable strings.

When the user wants to add a time zone, they will tap the Floating Action Button (FAB) and a dialog will appear to allow them to select all the time zones they want:

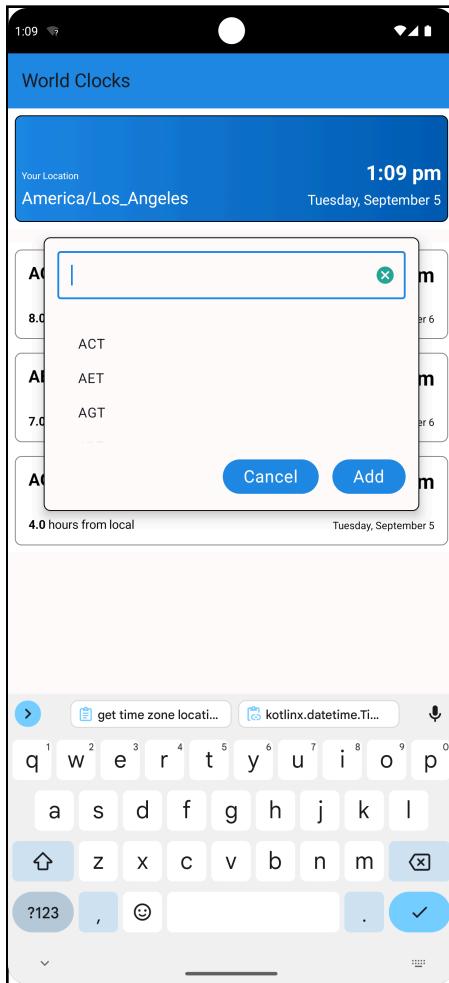


Fig. 3.2 – Dialog to search for time zones

Next up is the search screen, which allows the user to select the start and end times for their day and includes a search button to show the hours available.

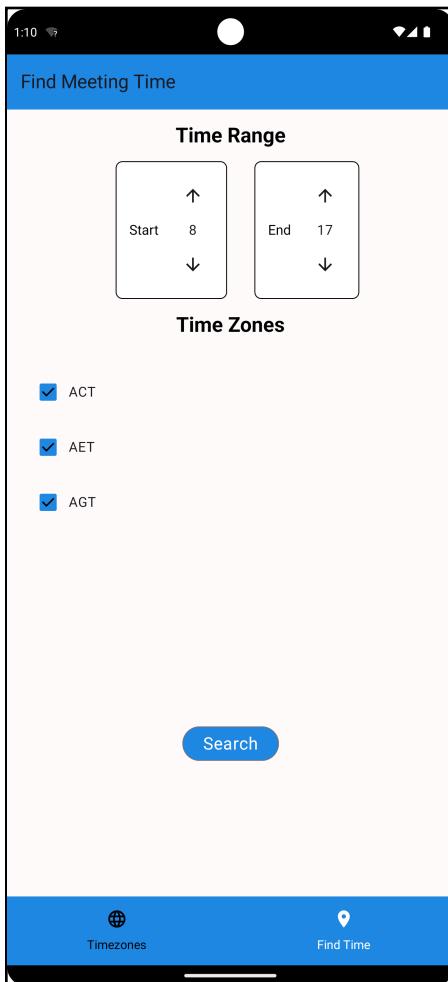


Fig. 3.3 – Select meeting start and end time ranges

Tapping the search button brings up the result dialog:

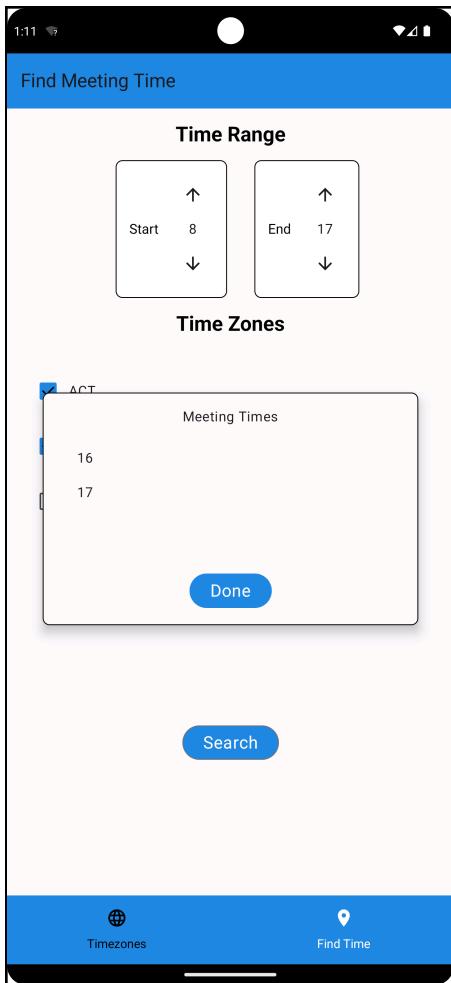


Fig. 3.4 – List of possible times for the meeting

Note: While this chapter goes into some detail about Jetpack Compose, it's not intended to be a thorough examination of how to use it. For a deeper understanding of Jetpack Compose, check out the books at <https://www.kodeco.com/android/books>.

Themes

One of the first Compose functions you need to learn about is the theme. This is the color scheme you'll use for your app. In Android, you would normally have a **style.xml** or **theme.xml** file with specifications for colors, fonts and other areas of UI styling. In Compose, you use a theme function. Since you have included the **Material3** Compose library, you can use the **MaterialTheme** class as a starting point for setting colors, fonts and shapes. Compose can also tell you if the system is using the dark theme. Luckily, Android Studio creates a theme for you. In the starter project, the theme has been changed to use **Material3** instead of the older **Material** library.

Open up **MyApplicationTheme.kt**. This is a Composable function that defines the light and dark colors of the app:

```
@Composable
fun MyApplicationTheme(
    darkTheme: Boolean = isSystemInDarkTheme(),
    content: @Composable () -> Unit
) {
    val colors = if (darkTheme) {
        darkColorScheme(
            primary = Color(0xFF005cb2),
            onPrimary = Color.White,
            secondary = Color(0xFF00766c)
        )
    } else {
        lightColorScheme(
            primary = Color(0xFF1e88e5),
            onPrimary = Color.Black,
            secondary = Color(0xFF26a69a)
        )
    }
}
```

This defines some primary and secondary colors. You can see the colors in the left margin. Change them if you want a different color scheme. Next are the definitions for typography and shapes. If you want to define other text types, define them here.

```
val typography = Typography(
    bodySmall = TextStyle(
        fontFamily = FontFamily.SansSerif,
        fontWeight = FontWeight.Normal,
        fontSize = 16.sp,
        color = Color.White
    ),
    headlineSmall = TextStyle(
        fontFamily = FontFamily.SansSerif,
```

```

        fontWeight = FontWeight.Bold,
        fontSize = 24.sp,
        color = Color.White
    ),
    labelLarge = TextStyle(
        fontFamily = FontFamily.SansSerif,
        fontWeight = FontWeight.Normal,
        fontSize = 20.sp,
        color = Color.White
    ),
    labelSmall = TextStyle(
        fontFamily = FontFamily.SansSerif,
        fontWeight = FontWeight.Normal,
        fontSize = 11.sp,
        color = Color.White
    ),
)
val shapes = Shapes(
    small = RoundedCornerShape(4.dp),
    medium = RoundedCornerShape(4.dp),
    large = RoundedCornerShape(0.dp)
)
)

```

You can also set the letter spacing and many other values defined in `TextStyle`.

The shapes define how you would like your corners on all kinds of controls. From buttons to text field borders to Floating Action Buttons.

Next, the passed-in content is wrapped in a `MaterialTheme` that uses the defined colors, typography and shapes.

```

MaterialTheme(
    colorScheme = colors,
    typography = typography,
    shapes = shapes,
    content = content
)

```

Above the `MyApplicationTheme` definition, you'll find two colors defined as follows:

```

val startGradientColor = Color(0xFF1e88e5)
val endGradientColor = Color(0xFF005cb2)

```

You'll use the above colors later in the chapter.

Types

Before you get to the main screen, you'll need a few custom types that will be used throughout the app. In the `ui` folder, create a new Kotlin file named `Types.kt`. Add the following:

```
import androidx.compose.runtime.Composable

// 1
typealias OnAddType = (List<String>) -> Unit
// 2
typealias onDismissType = () -> Unit
// 3
typealias composeFun = @Composable () -> Unit
// 4
typealias topBarFun = @Composable (Int) -> Unit

// 5
@Composable
fun EmptyComposable() {
}
```

Here's what the above code does:

1. Define an alias named `OnAddType` that takes a list of strings and doesn't return anything.
2. Define an alias used when dismissing a dialog.
3. Define a composable function.
4. Define a function that takes an integer.
5. Define an empty composable function (as a default variable for the Top Bar).

Now that you have your colors and text styles set up, it's time to create your first screen.

Main Screen

Inside the `androidApp` module, create a new Kotlin file named `MainView.kt` in the `ui` folder. You'll start by creating some helper classes and variables. First, add the imports you'll need (this saves some time importing):

```
import androidx.compose.foundation.layout.Box
import androidx.compose.foundation.layout.padding
import androidx.compose.material.icons(Icons)
```

```
import androidx.compose.material.icons.filled.Add
import androidx.compose.material.icons.filled.Language
import androidx.compose.material.icons.filled.Place
import androidx.compose.material3.FloatingActionButton
import androidx.compose.material3.FloatingActionButtonDefaults
import androidx.compose.material3.Icon
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Scaffold
import androidx.compose.material3.Text
import androidx.compose.material3.NavigationBar
import androidx.compose.material3.NavigationBarItem
import androidx.compose.runtime.Composable
import androidx.compose.runtime.mutableIntStateOf
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.snapshots.SnapshotStateList
import androidx.compose.ui.Modifier
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.graphics.vector.ImageVector
import androidx.compose.ui.unit.dp
import com.kodeco.findtime.android.MyApplicationTheme
```

Notice that you're importing the material icons you'll use and a few other compose classes for building your UI.

To keep track of your two screens, create a new sealed class named **Screen**:

```
sealed class Screen(val title: String) {
    object TimeZonesScreen : Screen("Timezones")
    object FindTimeScreen : Screen("Find Time")
}
```

This just defines two screens: `TimeZonesScreen` and `FindTimeScreen`, along with their titles. Here, this sealed class is similar to an enum class that allows you to switch between screens and titles. You can learn more about sealed classes and other advanced classes available in Kotlin in the reference link shared at the end of the chapter.

Next, define a class to handle the bottom navigation item. Write the following code just below the `Screen` class you created above:

```
data class BottomItem(
    val route: String,
    val icon: ImageVector,
    val iconContentDescription: String
)
```

This defines a route, an icon for that route and a content description. Next, create a variable with two items. Add it just below the `BottomItem` class:

```
val bottomNavigationItems = listOf(
    BottomItem(
        Screen.TimeZonesScreen.title,
        Icons.Filled.Language,
        "Timezones"
    ),
    BottomItem(
        Screen.FindTimeScreen.title,
        Icons.Filled.Place,
        "Find Time"
    )
)
```

This list will help you create the bottom navigation UI that you will be using to switch the screens. This uses the material icons and the titles from the screen class. Now, create the `MainView` composable:

```
// 1
@Composable
// 2
fun MainView(actionBarFun: topBarFun = { EmptyComposable() }) {
    // 3
    val showAddDialog = remember { mutableStateOf(false) }
    // 4
    val currentTimezoneStrings = remember
    { SnapshotStateList<String>() }
    // 5
    val selectedIndex = remember { mutableIntStateOf(0) }
    // 6
    MyApplicationTheme {
        // TODO: Add Scaffold
    }
}
```

Here's what the above code does:

1. Define this function as a composable.
2. This function takes a function that can provide a top bar (toolbar on Android) and defaults to an empty composable.
3. Hold the state for showing the add dialog. If the state object is true then the app will show a dialog, otherwise it will hide the add dialog.
4. Hold the state containing a list of current time zone strings.

5. Hold the state containing the currently selected index.
6. Use the current theme composable.

Note that in the above code, you have used `compose remember` and `mutableStateOf` functions to remember the state of the UI.

State

State is any **value** that **can change over time**. Compose uses a few functions for handling state. The most important one is `remember`. This stores the variable so that it's remembered between redraws of the screen. When the user selects between the two bottom buttons, you want to save this selection to update which screen is showing. A `MutableState` is a value holder that tells the Compose engine to redraw whenever the state changes.

Here are some key functions:

1. `remember`: Remembers the variable and retains its value between redraws.
2. `mutableStateOf`: Creates a `MutableState` instance whose state is observed by Compose.
3. `SnapshotStateList`: Creates a `MutableList` whose state is observed by Compose.
4. `collectAsState`: Collects values from a Kotlin coroutine `StateFlow` and is observed by Compose.

Scaffold

Compose uses a function named `Scaffold` that uses the Material Design layout structure with an app bar (toolbar) and an optional floating action button. By using this function, your screen will be laid out properly.

Start by replacing `// TODO: Add Scaffold` with:

```
Scaffold(  
    topBar = {  
        // TODO: Add Toolbar  
    },  
    floatingActionButton = {  
        // TODO: Add Floating action button  
    },  
    bottomBar = {  
        // TODO: Add bottom bar  
    }
```

```

    }
) { padding ->
    Box(modifier = Modifier.padding(padding)) {
        // TODO: Replace with Dialog
        // TODO: Replace with screens
    }
}

```

As you can see, there are places to add composable functions inside the **topBar**, **floatingActionButton** and **bottomBar** parameters.

TopAppBar

The **TopAppBar** is Compose's function for a toolbar. Since every platform handles a toolbar differently – macOS displays menu items in the system toolbar, whereas Windows uses a separate toolbar – this section is optional. If the platform passes in a function that creates one, it will use that. Replace `// TODO: Add Toolbar` with:

```
actionBarFun(selectedIndex.intValue)
```

This calls the passed-in function with the currently selected bottom bar index, whose value is stored in the `selectedIndex` state variable. Since `actionBarFun` gets set to an empty function by default, nothing will happen unless a function is passed in. You'll do this later for the Android app. Now add the code to show a floating action button if you're on the first screen but not on the second screen. Replace `// TODO: Add Floating action button` with:

```

if (selectedIndex.intValue == 0) {
    // 1
    FloatingActionButton(
        // 2
        modifier = Modifier
            .padding(16.dp),
        shape = FloatingActionButtonDefaults.largeShape,
        containerColor = MaterialTheme.colorScheme.secondary,
        // 3
        onClick = {
            showAddDialog.value = true
        }
    ) {
        // 4
        Icon(
            imageVector = Icons.Default.Add,
            contentDescription = "Add Timezone"
        )
    }
}

```

Here's the explanation for the code:

1. For the first page, create a `FloatingActionButton`.
2. Use Compose's `Modifier` function to add padding.
3. Set a click listener. Set the variable to show the add dialog screen. Changing this value will cause a redraw of the screen.
4. Use the Add icon for the FAB.

Bottom Navigation

Compose has a `BottomNavigation` function that creates a bottom bar with icons. Underneath, it's a Compose Row class that you fill with your content.

Replace // TODO: Add bottom bar with:

```
// 1
NavigationBar(
    containerColor = MaterialTheme.colorScheme.primary
) {
// 2
    bottomNavigationItems.forEachIndexed { i, bottomNavigationItem
->
// 3
    NavigationBarItem(
        colors = NavigationBarItemDefaults.colors(
            selectedIconColor = Color.White,
            selectedTextColor = Color.White,
            unselectedIconColor = Color.Black,
            unselectedTextColor = Color.Black,
            indicatorColor = MaterialTheme.colorScheme.primary,
        ),
        label = {
            Text(bottomNavigationItem.route, style =
MaterialTheme.typography.bodyMedium)
        },
// 4
        icon = {
            Icon(
                bottomNavigationItem.icon,
                contentDescription =
bottomNavigationItem.iconContentDescription
            )
        },
// 5
        selected = selectedIndex.intValue == i,
// 6
        onClick = {
    
```

```
        selectedIndex.intValue = i  
    }  
}
```

Here's what the code does:

1. Create a `NavigationBar` composable.
 2. Use `forEachIndexed` to go through each item in your list of navigation items.
 3. Create a new `NavigationBarItem`.
 4. Set the `icon` field to the icon in your list.
 5. Is this screen selected? Only if the `selectedIndex` value is the current index.
 6. Set the click listener. Change the `selectedIndex` value and the screen will redraw.

Next, return to **MainActivity.kt** and add the following imports:

```
import androidx.compose.material3.TopAppBar
import androidx.compose.material3.TopAppBarDefaults
import androidx.compose.ui.res.stringResource
import com.kodeco.findtime.android.ui.MainView
import io.github.aakira.napier.DebugAntilog
import io.github.aakira.napier.Napier
```

Then, replace `setContent` with:

```
// 1
Napier.base(DebugAntilog())
setContent {
    // 2
    MainView {
        // 3
        TopAppBar(
            colors =
TopAppBarDefaults.topAppBarColors(containerColor =
MaterialTheme.colorScheme.primary),
            title = {
                // 4
                when (it) {
                    0 -> Text(text =
stringResource(R.string.world_clocks))
                    else -> Text(text =
stringResource(R.string.findmeeting))
                }
            }
        )
    }
}
```

```
        })  
    }  
}
```

Here's what you did:

1. Initialize the **Napier** logging library. (Be sure to include needed imports.)
2. Set your main content to the **MainView** composable.
3. For Android, you want a top app bar. So you have added a **TopAppBar** composable function.
4. You check the currently selected index for the screen. When the first screen is showing, have the title be **World Clocks**. Otherwise, show **Find Meeting**.

Build and run the app on a device or emulator. Here's what you'll see:

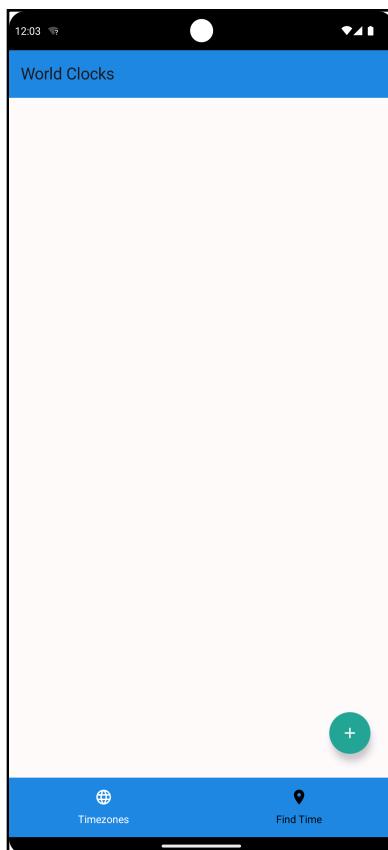


Fig. 3.5 – Basic structure of the World Clocks screen

Now you have a working app that displays a title bar, a floating action button and a bottom navigation bar. Try switching between the two icons. What happens?

Local Time Card

Moving ahead, the first thing you want to show on the screen is the user's local time zone, time and date. This will be in a card with a blue gradient.

It will look like this:



Fig. 3.6 – Card to display the local time zone

In the **ui** folder, create a new Kotlin file named **LocalTimeCard.kt**. First, add the following imports that you will need for creating the card:

```
import androidx.compose.foundation.BorderStroke
import androidx.compose.foundation.background
import androidx.compose.foundation.layout.Box
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.Row
import androidx.compose.foundation.layout.Spacer
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.height
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.shape.RoundedCornerShape
import androidx.compose.material3.Card
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.graphics.Brush
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.unit.dp
import com.kodeco.findtime.android.endGradientColor
import com.kodeco.findtime.android.startGradientColor
```

Then, add the following code below:

```
@Composable
// 1
fun LocalTimeCard(city: String, time: String, date: String) {
```

```
// 2
Box(
    modifier = Modifier
        .fillMaxWidth()
        .height(140.dp)
        .background(MaterialTheme.colorScheme.background)
        .padding(8.dp)
) {
    // 3
    Card(
        shape = RoundedCornerShape(8.dp),
        border = BorderStroke(1.dp, Color.Black),
        modifier = Modifier
            .fillMaxWidth()
    )
    {
        // TODO: Add body
    }
}
}
```

Here's the explanation of the code above:

1. Create a function named `LocalTimeCard` that takes a `city`, `time`, and `date` as a string.
2. Use a `Box` function that fills the current width and has a height of 140 dp and a white background. `Box` is a container that draws elements inside it on top of one another.
3. Use a `Card` with rounded corners and a black border. It also fills the width.

For the body, replace `// TODO: Add body` with:

```
// 1
Box(
    modifier = Modifier
        .background(
            brush = Brush.horizontalGradient(
                colors = listOf(
                    startGradientColor,
                    endGradientColor,
                )
            )
        )
        .padding(8.dp)
) {
    // 2
    Row(
        modifier = Modifier
            .fillMaxWidth()
```

```
) {
    // 3
    Column(
        horizontalAlignment = Alignment.Start
    ) {
        // 4
        Spacer(modifier = Modifier.weight(1.0f))
        Text(
            "Your Location", style =
        MaterialTheme.typography.bodySmall
        )
        Spacer(Modifier.height(8.dp))
        // 5
        Text(
            city, style =
        MaterialTheme.typography.headlineSmall
        )
        Spacer(Modifier.height(8.dp))
    }
    // 6
    Spacer(modifier = Modifier.weight(1.0f))
    // 7
    Column(
        horizontalAlignment = Alignment.End
    ) {
        Spacer(modifier = Modifier.weight(1.0f))
        // 8
        Text(
            time, style =
        MaterialTheme.typography.headlineSmall
        )
        Spacer(Modifier.height(8.dp))
        // 9
        Text(
            date, style = MaterialTheme.typography.bodySmall
        )
        Spacer(Modifier.height(8.dp))
    }
}
```

Here's an explanation of the code above:

1. Use a box to display the gradient background.
2. Create a row that fills the entire width.
3. Create a column for the left side of the card.
4. Use a spacer with a weight modifier to push the text to the bottom.

5. Display the city text with the given typography.
6. Push the right column over by using a spacer with a weight modifier.
7. Create the right column.
8. Show the time with the given typography.
9. Show the date with the given typography.

Time Zone Screen

Now that you have your cards ready, it's time to put them all together in one screen. In the `ui` directory, create a new file named `TimeZoneScreen.kt`. Add the imports and a constant:

```
import androidx.compose.foundation.background
import androidx.compose.foundation.layout.*
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.foundation.lazy.rememberLazyListState
import androidx.compose.material.Icon
import androidx.compose.material.icons(Icons)
import androidx.compose.material.icons.filled.Delete
import androidx.compose.runtime.*
import androidx.compose.runtime.snapshots.SnapshotStateList
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.unit.dp
import com.kodeco.findtime.TimeZoneHelper
import com.kodeco.findtime.TimeZoneHelperImpl
import kotlinx.coroutines.delay

const val timeMillis = 1000 * 60L // 1 second
```

Next, create the composable:

```
@Composable
fun TimeZoneScreen(
    currentTimezoneStrings: SnapshotStateList<String>
) {
    // 1
    val timezoneHelper: TimeZoneHelper = TimeZoneHelperImpl()
    // 2
    val listState = rememberLazyListState()
    // 3
    Column(
        modifier = Modifier
```

```

        .fillMaxSize()
    ) {
    // TODO: Add Content
}
}

```

This function takes a list of current time zones. This list is a `SnapshotStateList` so that this class can change the values, and other functions will be notified of the changes.

Finally, here's the explanation of the remaining code:

1. Create an instance of your `TimeZoneHelper` class.
2. Remember the state of the list that will be defined later.
3. Create a vertical column that takes up the full width.

Moving ahead. Replace `// TODO: Add Content` with:

```

// 1
var time by remember
{ mutableStateOf(timezoneHelper.currentTime()) }
// 2
LaunchedEffect(Unit) {
    while (true) {
        time = timezoneHelper.currentTime()
        delay(timeMillis) // Every minute
    }
}
// 3
LocalTimeCard(
    city = timezoneHelper.currentTimeZone(),
    time = time, date =
timezoneHelper.getDate(timezoneHelper.currentTimeZone())
)
Spacer(modifier = Modifier.size(16.dp))

// TODO: Add Timezone items

```

Here's what the above code does:

1. Remember the current time. Note that here you are using `by` instead of `=`. In this case, the `time` variable is the current time in a string format.
2. Use Compose's `LaunchedEffect`. It will be launched once but continue to run. The method will get the updated time every minute. You pass `Unit` as a parameter to `LaunchedEffect` so that it is not canceled and re-launched when `LaunchedEffect` is recomposed.

3. Use the `LocalTimeCard` function you created earlier. Use `TimeZoneHelper`'s methods to get the current time zone and current date.

Return to **MainView**. Replace `// TODO: Replace with screens` with the following:

```
when (selectedIndex.intValue) {  
    0 -> TimeZoneScreen(currentTimezoneStrings)  
    // 1 -> FindMeetingScreen(currentTimezoneStrings)  
}
```

If the index is 0, the app shows the Time Zone screen, otherwise, it will show the Find Meeting screen. The Find Meeting screen is commented out until you write it.

Build and run the app. It will look like this:

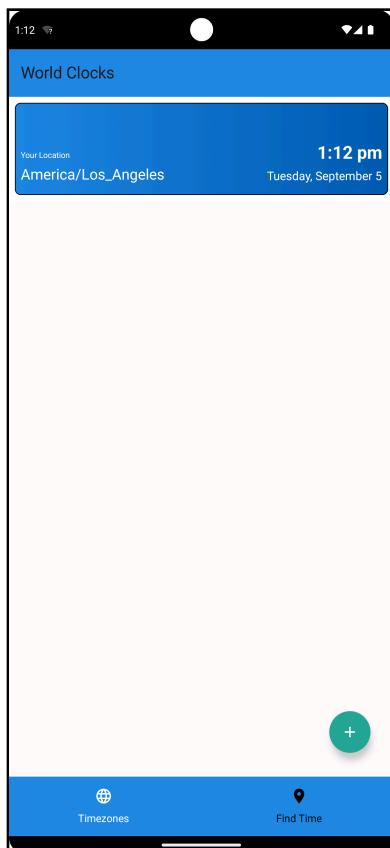


Fig. 3.7 – World Clocks screen with local time zone

Nicely done! Your app is really starting to take shape now.

Time Card

Next up, we will create the time card that will display the time, date, and time difference from local and other timezones. At the end, the time card will look like this:



Fig. 3.8 – Card to display a time zone

In the **ui** folder, create a new Kotlin file named **TimeCard.kt**. Add the following:

```
import androidx.compose.foundation.BorderStroke
import androidx.compose.foundation.background
import androidx.compose.foundation.layout.Box
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.Row
import androidx.compose.foundation.layout.Spacer
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.height
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.shape.RoundedCornerShape
import androidx.compose.material3.Card
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.text.TextStyle
import androidx.compose.ui.text.font.FontWeight
import androidx.compose.ui.unit.dp
import androidx.compose.ui.unit.sp

@Composable
// 1
fun TimeCard(timezone: String, hours: Double, time: String,
date: String) {
    // 2
    Box(
        modifier = Modifier
            .fillMaxSize()
            .height(120.dp)
            .background(Color.White)
            .padding(8.dp)
    )
}
```

```
// 3
Card(
    shape = RoundedCornerShape(8.dp),
    border = BorderStroke(1.dp, Color.Gray),
    modifier = Modifier
        .fillMaxWidth()
)
{
    // TODO: Add Content
}
}
```

Here's what's happening in this code:

1. This function takes a time zone, hours, time and date that will be used in the time card.
2. Use a Box to take up the full width and give it a white background.
3. Create a nice-looking card using the Card Composable with rounded corners and a grey border.

Now that you have the card, add some content by adding a few rows and columns inside the card. Replace `// TODO: Add Content` with:

```
// 1
Box(
    modifier = Modifier
        .background(
            color = Color.White
        )
        .padding(16.dp)
) {
    // 2
    Row(
        modifier = Modifier
            .fillMaxWidth()
    ) {
        // 3
        Column(
            horizontalAlignment = Alignment.Start
        ) {
            // 4
            Text(
                timezone, style = TextStyle(
                    color = Color.Black,
                    fontWeight = FontWeight.Bold,
                    fontSize = 20.sp
            )
        }
    }
}
```

```
)  
    Spacer(modifier = Modifier.weight(1.0f))  
    // 5  
    Row {  
        // 6  
        Text(  
            hours.toString(), style = TextStyle(  
                color = Color.Black,  
                fontWeight = FontWeight.Bold,  
                fontSize = 14.sp  
            )  
        )  
        // 7  
        Text(  
            " hours from local", style = TextStyle(  
                color = Color.Black,  
                fontSize = 14.sp  
            )  
        )  
    }  
    Spacer(modifier = Modifier.weight(1.0f))  
    // 8  
    Column(  
        horizontalAlignment = Alignment.End  
    ) {  
        // 9  
        Text(  
            time, style = TextStyle(  
                color = Color.Black,  
                fontWeight = FontWeight.Bold,  
                fontSize = 24.sp  
            )  
        )  
        Spacer(modifier = Modifier.weight(1.0f))  
        // 10  
        Text(  
            date, style = TextStyle(  
                color = Color.Black,  
                fontSize = 12.sp  
            )  
        )  
    }  
}
```

Here's the explanation of the code you have added:

1. Use a box to set the background to white.
2. Create a row that fills the width.

3. Create a column on the left side.
4. Show the time zone.
5. Create a row underneath the previous one.
6. Show the hours in bold.
7. Show the text “hours from local.”
8. Create a column on the right side.
9. Show the time.
10. Show the date.

Notice how you’re building up the screen section by section. You can’t quite use these cards yet, as you need a way to add a new time zone. You’ll do this later by creating a dialog that will allow the user to pick many time zones to add.

Next, you will be writing code to build a user-selected list of timezone item cards that we just created above. The following code will go through the list of current time zone strings and wrap the item in an `AnimatedSwipeDismiss` to allow the user to swipe and delete the card and then use the new time card. Return to `TimezoneScreen` and replace // TODO: Add Timezone items with:

```
// 1
LazyColumn(
    state = listState,
) {
    // 2
    items(currentTimezoneStrings.size,
        // 3
        key = { timezone ->
            timezone
        }) { index ->
    val timezoneString = currentTimezoneStrings[index]
    // 4
    AnimatedSwipeDismiss(
        item = timezoneString,
        // 5
        background = { _ ->
            Box(
                modifier = Modifier
                    .fillMaxSize()
                    .height(50.dp)
                    .background(Color.Red)
                    .padding(
                        start = 20.dp,
                        end = 20.dp
                    )
            )
        }
    )
}
```

```
        )
    ) {
        val alpha = 1f
        Icon(
            Icons.Filled.Delete,
            contentDescription = "Delete",
            modifier = Modifier
                .align(Alignment.CenterEnd),
            tint = Color.White.copy(alpha = alpha)
        )
    }
},
content = {
    // 6
    TimeCard(
        timezone = timezoneString,
        hours =
timezoneHelper.hoursFromTimeZone(timezoneString),
        time =
timezoneHelper.getTime(timezoneString),
        date =
timezoneHelper.getDate(timezoneString)
    )
},
// 7
onDismiss = { zone ->
    if (currentTimezoneStrings.contains(zone)) {
        currentTimezoneStrings.remove(zone)
    }
}
)
}
```

Here's the explanation of the above code:

1. Use Compose's `LazyColumn` function, which is like Android's `RecyclerView` or iOS's `UITableView` for building the vertical list items of time zone cards.
 2. Use `LazyColumn`'s `items` method to go through the list of time zones.
 3. Use the `key` field to set the unique key for each time zone card. This is important if you need to delete items.
 4. Use the included `AnimatedSwipeDismiss` composable to handle swiping away a time zone card.
 5. Set the background to red which will be shown when swiping. You have also added a delete icon on the background to let the user know what swiping the card will do.

6. Set the content as a time zone card that will be shown over the background.
7. Define the function `onDismiss` to remove the time zone string from your list when the time zone card is swiped away and remove that time zone card from the view.

Finally, return to `MainView`. Now you want to show the Add Timezone Dialog when the `showAddDialog` Boolean is `true`. Replace `// TODO: Replace with Dialog` with:

```
// 1
if (showAddDialog.value) {
    AddTimeZoneDialog(
        // 2
        onAdd = { newTimezones ->
            showAddDialog.value = false
            for (zone in newTimezones) {
                // 3
                if (!currentZoneStrings.contains(zone)) {
                    currentZoneStrings.add(zone)
                }
            }
        },
        onDismiss = {
            // 4
            showAddDialog.value = false
        },
    )
}
```

Here's what you just did:

1. If your variable to show the dialog is true, call the `AddTimeZoneDialog` composable.
2. Your `onAdd` lambda will receive a list of new time zones.
3. If your current list doesn't already contain the time zone, then add it to your list.
4. Set the `show` variable back to false.

Build and run the app again. Click the FAB. You'll see the dialog as follows:

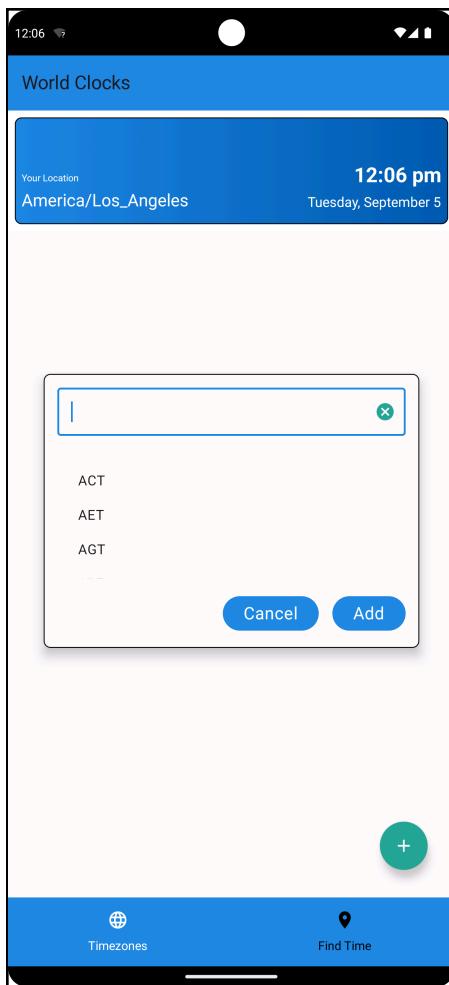


Fig. 3.9 – Time zone search is functional

Search for a time zone and select it. Hit the clear button, search for another time zone, and select it. Finally, press the add button. If you selected Los Angeles and New York, you would see something like:

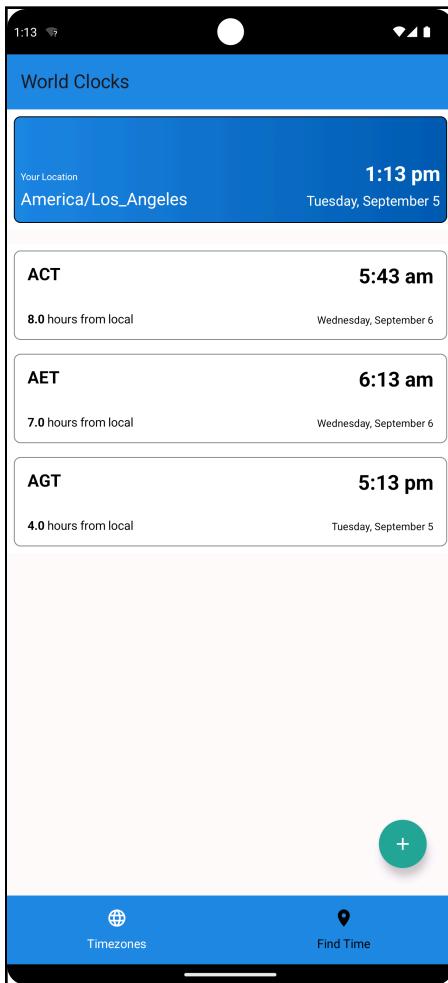


Fig. 3.10 – List of selected time zones

Find Meeting Time Screen

Now that you have the Time Zone screen finished, it's time to write the Find Meeting Time screen. This screen will allow the user to choose the hour range they want to meet, select the time zones to search against and perform a search that will bring up a dialog with the list of hours found.

Since a composable is made up of many parts, you'll use the included number picker composable that will look like this:

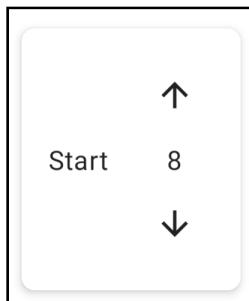


Fig. 3.11 – Time chooser UI component

This has a text field on the left, an up arrow, a number, and a down arrow. You'll use this for both the start and end hours.

Number Time Card

In the **ui** folder, create a new file named **NumberTimeCard.kt**. This will display a card with the label and number picker. Add the following code:

```
import androidx.compose.foundation.BorderStroke
import androidx.compose.foundation.layout.Row
import androidx.compose.foundation.layout.Spacer
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.layout.size
import androidx.compose.foundation.shape.RoundedCornerShape
import androidx.compose.material3.Card
import androidx.compose.material3.CardDefaults
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.runtime.MutableState
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.unit.dp

// 1
@Composable
fun NumberTimeCard(label: String, hour: MutableState<Int>) {
    // 2
    Card(
        shape = RoundedCornerShape(8.dp),
        colors = CardDefaults.cardColors(containerColor =
Color.White),
        border = BorderStroke(1.dp, Color.Black),
```

```
) {
    // 3
    Row(
        modifier = Modifier
            .padding(16.dp)
    ) {
        // 4
        Text(
            modifier = Modifier
                .align(Alignment.CenterVertically),
            text = label,
            style =
        MaterialTheme.typography.bodySmall.copy(color = Color.Black)
        )
        Spacer(modifier = Modifier.size(16.dp))
        // 5
        NumberPicker(hour = hour, range = IntRange(0, 23),
            onStateChanged = {
                hour.value = it
            })
    }
}
```

Here's an explanation of the numbered comments:

1. Create a composable that will take a `label` and an `hour` as arguments. Notice that `hour` is of the type `MutableState<Int>`.
2. Create a rounded card having black border and white color.
3. Use a row to lay out the items horizontally.
4. Create and center the label using `Alignment.CenterVertically`.
5. Use `NumberPicker` to show the hour with up/down arrows. It also contains the `onStateChanged` callback to change the hour values based on clicking the up/down arrows.

This creates a card with a text field on the left and a number picker on the right.

Creating the Find Meeting Time Screen

Now you can put together the Find Meeting Time screen. In the **ui** folder, create a new file named **FindMeetingScreen.kt**. Add the following code:

```
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.PaddingValues
import androidx.compose.foundation.layout.Row
import androidx.compose.foundation.layout.Spacer
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.layout.size
import androidx.compose.foundation.layout.wrapContentSize
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.foundation.lazy.itemsIndexed
import androidx.compose.foundation.lazy.rememberLazyListState
import androidx.compose.material3.Checkbox
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.OutlinedButton
import androidx.compose.material3.ButtonDefaults
import androidx.compose.material3.Surface
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.runtime.mutableIntStateOf
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.snapshots.SnapshotStateList
import androidx.compose.runtime.snapshots.SnapshotStateMap
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.unit.dp
import androidx.compose.ui.graphics.Color
import com.kodeco.findtime.TimeZoneHelper
import com.kodeco.findtime.TimeZoneHelperImpl

// 1
@Composable
fun FindMeetingScreen(
    timezoneStrings: List<String>
) {
    val listState = rememberLazyListState()
    // 2
    // 8am
    val startTime = remember {
        mutableIntStateOf(8)
    }
    // 5pm
    val endTime = remember {
        mutableIntStateOf(17)
    }
}
```

```
// 3
val selectedTimeZones = remember {
    val selected = SnapshotStateMap<Int, Boolean>()
    for (i in timezoneStrings.indices) selected[i] = true
    selected
}
// 4
val timezoneHelper: TimeZoneHelper = TimeZoneHelperImpl()
val showMeetingDialog = remember { mutableStateOf(false) }
val meetingHours = remember { SnapshotStateList<Int>() }

// 5
if (showMeetingDialog.value) {
    MeetingDialog(
        hours = meetingHours,
        onDismiss = {
            showMeetingDialog.value = false
        }
    )
}
// TODO: Add Content
}

// TODO: Add getSelectedTimeZones
```

Here's an explanation of the code you just added:

1. Create a composable that takes a list of time zone strings.
2. Create some variables to hold the start and end hours. Default to 8 a.m. and 5 p.m.
3. Remember the selected time zones.
4. Create your time zone helper and remember a few more states that you will need like `showMeetingDialog` and `meetingHours`.
5. If the boolean for `showMeetingDialog.value` is true, then show the `MeetingDialog` results.

Here, you've set up all of your variables and put in a small bit of code to show the Add Meeting Dialog when the variable is true. Now, replace // TODO: Add `getSelectedTimeZones` with:

```
fun getSelectedTimeZones(
    timezoneStrings: List<String>,
    selectedStates: Map<Int, Boolean>
): List<String> {
    val selectedTimezones = mutableListOf<String>()
    selectedStates.keys.map {
```

```

        val timezone = timezoneStrings[it]
        if (isSelected(selectedStates, it) && !selectedTimezones.contains(timezone)) {
            selectedTimezones.add(timezone)
        }
    }
    return selectedTimezones
}

```

This is a helper function that will return a list of selected time zones based on the selected state map. Now, add the contents in the `FindMeetingScreen` composable. Replace // TODO: Add Content with:

```

// 1
Column(
    modifier = Modifier
        .fillMaxSize()
) {
    Spacer(modifier = Modifier.size(16.dp))
// 2
    Text(
        modifier = Modifier
            .fillMaxWidth()
            .wrapContentWidth(Alignment.CenterHorizontally),
        text = "Time Range",
        style = MaterialTheme.typography.headlineSmall.copy(color
= MaterialTheme.colorScheme.onBackground)
    )
    Spacer(modifier = Modifier.size(16.dp))
// 3
    Row(
        modifier = Modifier
            .fillMaxWidth()
            .padding(start = 4.dp, end = 4.dp)
            .wrapContentWidth(Alignment.CenterHorizontally),
    ) {
        // 4
        Spacer(modifier = Modifier.size(16.dp))
        NumberTimeCard("Start", startTime)
        Spacer(modifier = Modifier.size(32.dp))
        NumberTimeCard("End", endTime)
    }
    Spacer(modifier = Modifier.size(16.dp))
// 5
    Row(
        modifier = Modifier
            .fillMaxWidth()
            .padding(start = 4.dp, end = 4.dp)
    ) {

```

```

    Text(
        modifier = Modifier
            .fillMaxWidth()
            .wrapContentWidth(Alignment.CenterHorizontally),
        text = "Time Zones",
        style =
    MaterialTheme.typography.headlineSmall.copy(color =
    MaterialTheme.colorScheme.onBackground)
    )
}
Spacer(modifier = Modifier.size(16.dp))
// TODO: Add LazyColumn
}

```

Here's what this code does:

1. Create a column that takes up the full width.
2. Add a **Time Range** header in the center of the column.
3. Add a row that is centered horizontally.
4. Add two NumberTimeCards with their labels and hours with some white space between them by using the Spacer composable.
5. Add a row that takes up the full width and has a “Time Zones” header.

This creates a column with a text field, a start & end hour picker, and another text field. Next replace // TODO: Add LazyColumn with:

```

// 1
LazyColumn(
    modifier = Modifier
        .weight(0.6F)
        .fillMaxWidth(),
    contentPadding = PaddingValues(16.dp),
    state = listState,
) {
// 2
    itemsIndexed(timezoneStrings) { i, timezone ->
        Surface(
            modifier = Modifier
                .padding(8.dp)
                .fillMaxWidth(),
        ) {
            Row(
                modifier = Modifier
                    .fillMaxWidth(),
            ) {
// 3

```

```
        Checkbox(checked = isSelected(selectedTimeZones,
    i),
            onCheckedChange = {
                selectedTimeZones[i] = it
            })
        Text(timezone, modifier =
Modifier.align(Alignment.CenterVertically))
    }
}
Spacer(Modifier.weight(0.1f))
Row(
    modifier = Modifier
        .fillMaxWidth()
        .weight(0.2F)
        .wrapContentWidth(Alignment.CenterHorizontally)
        .padding(start = 4.dp, end = 4.dp)
) {
    // 4
    OutlinedButton(
        colors =
ButtonDefaults.outlinedButtonColors(containerColor =
MaterialTheme.colorScheme.primary),
        onClick = {
            meetingHours.clear()
            meetingHours.addAll(
                timezoneHelper.search(
                    startTime.intValue,
                    endTime.intValue,
                    getSelectedTimeZones(timezoneStrings,
selectedTimeZones)
                )
            )
            showMeetingDialog.value = true
        } {
            Text("Search")
        }
}
Spacer(Modifier.size(16.dp))
```

Here's what you just did:

1. Add a `LazyColumn` for the list of selected time zones. Give it a weight and padding.
2. For each selected time zone, create a surface and row for adding a checkbox.
3. Create a checkbox that sets the selected map when clicked.
4. Create a button to start the search process and show the meeting dialog.

Remember that `LazyColumn` is used for lists. You use the `items` or `itemsIndexed` functions to show an item in a list. Each row will have a checkbox and text with the time zone name. At the bottom will be a button that will start the search process, get all the meeting hours and then show the meeting dialog.

Return to `MainView` and uncomment the `FindMeetingScreen` call. Build and run the app.

Switch between the **World Clocks** and the **Find Meeting Time** views. Add a few time zones and press the search button. If no hours appear, try increasing the end time.

Wow, that was a lot of work, but you now have a working Meeting Finder app in Android using Jetpack Compose!

Key Points

- In Android, you can create your UI in both traditional XML layouts or in the new Jetpack Compose framework.
- Jetpack Compose is made up of composable functions.
- Break up your UI into smaller composables.
- You can create a theme for your app that includes colors and typography.
- Jetpack Compose uses concepts like **Scaffold**, **TopAppBar** and **BottomNavigation** to simplify creating screens.

Where to Go From Here?

To learn more about Jetpack Compose and other references that are mentioned in the chapter, check out these resources:

- The book: <https://www.kodeco.com/books/jetpack-compose-by-tutorials>
- Official site: <https://developer.android.com/jetpack/compose>
- Video course: <https://www.kodeco.com/21959310-jetpack-compose/>
- Advanced Classes in Kotlin: <https://www.kodeco.com/books/kotlin-apprentice/v3.0/chapters/15-advanced-classes>

Congratulations! You've written a Jetpack Compose app that uses a shared library for the business logic. The next chapter will show you how to create the iOS app.

Chapter 4: Developing UI: iOS SwiftUI

By Kevin Moore

As you learned in the last chapter, KMP does not provide a framework for developing UI. You'll need to use a different framework for each platform. In this chapter, you'll learn about writing the UI for iOS with SwiftUI. SwiftUI is a declarative UI toolkit that works on iOS, macOS, watchOS and tvOS. This won't be an extensive discussion on SwiftUI, but it will teach you the basics.

This chapter assumes you're working on a Mac with the Xcode 15 app from Apple. If you're not on a Mac, feel free to skip this chapter. If you don't have Xcode, you can open any Swift files in Android Studio.

IDE

Xcode is Apple's IDE for iOS, iPadOS, watchOS, macOS and tvOS development. In this chapter, you can edit your Swift files in either Xcode or Android Studio. Android Studio has a good editor, but Xcode can preview your SwiftUI Views for you. The choice of the IDE is up to you and this section will walk you through using both IDEs. The starter project for this chapter has few extra files to help you get started.

Android Studio

Open the starter project in Android Studio and select the iOS configuration. You might see a red x in the icon.

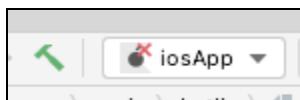


Fig. 4.1 – Android Studio iosApp Configuration

Select **Edit Configurations...** from the drop-down menu. You'll see:

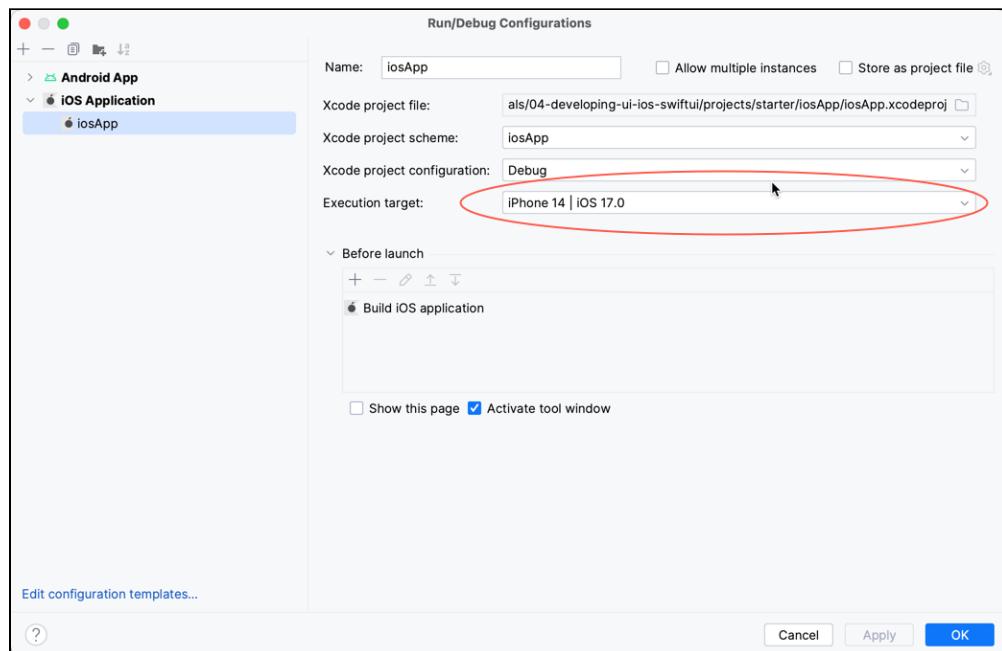


Fig. 4.2 – Android Studio Edit Configuration

Select a phone and a target, such as iPhone 14 | iOS 17.0 and click **OK**.

Now, click the **play icon** (or press Command-F9) to build. This will create the shared framework needed for iOS.

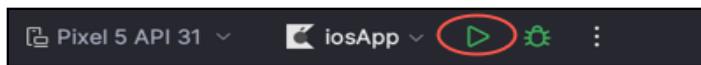


Fig. 4.3 – Android Studio Build Button

Xcode

Launch Xcode and open the **iosApp** directory under the starter project for this chapter. You don't have to select the xcworkspace or the xcodeproj file. Click **Open**.

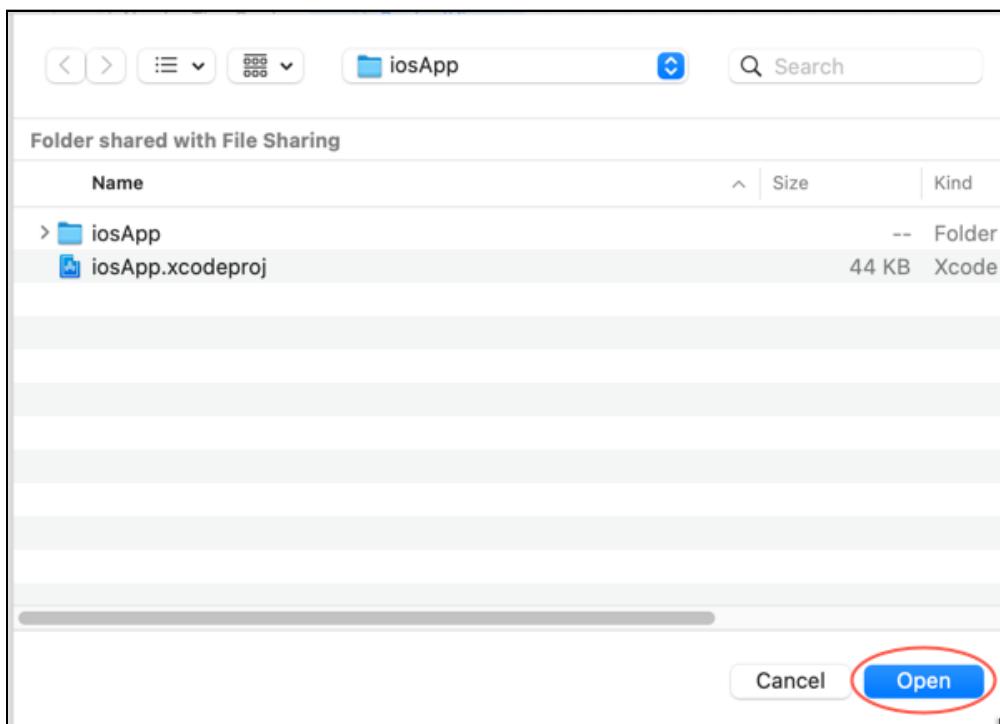


Fig. 4.4 – Xcode Open Project Dialog

Once the project is open, you'll see the two `iosApp` folders on the left:

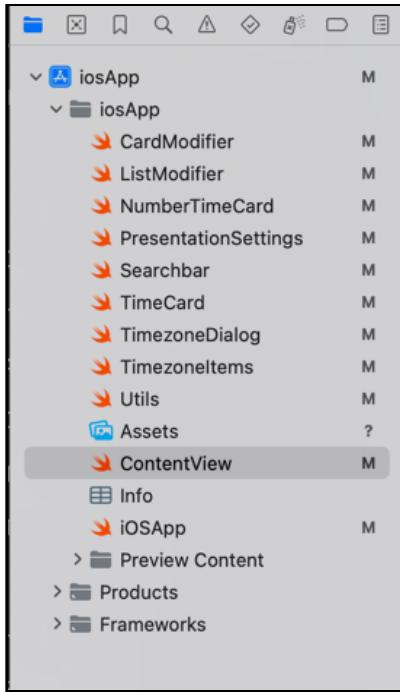


Fig. 4.5 – Xcode Project Files Sidebar

Current UI System

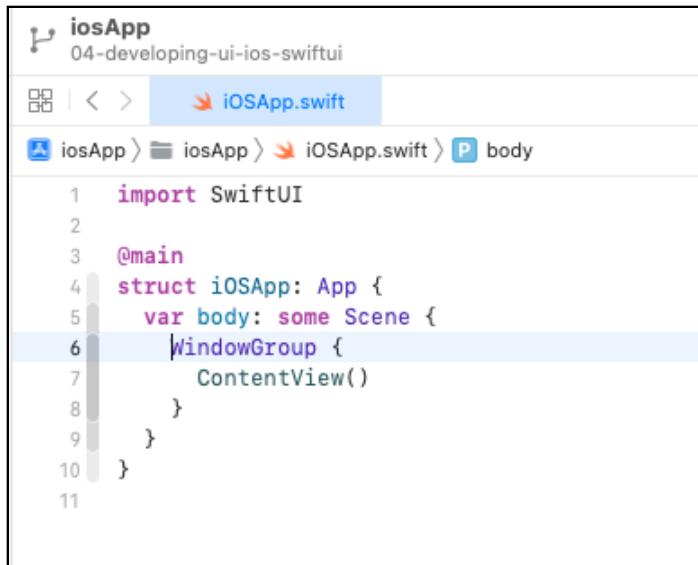
On iOS, you would typically use storyboards or create your UI in code if you were developing in UIKit to design your UI. Underneath those storyboards is a complex XML file. While the layout editor in Xcode is nice, it still takes quite a bit of work to design and then hook up to code. SwiftUI is a declarative UI system that's written entirely in code. No layouts or storyboards. It's a lot simpler to use and allows a lot of code reuse with smaller views. Xcode provides previews so that you can build small components and view them next to the editor.

Getting to Know SwiftUI

Creating the project using the Kotion Multiplatform Mobile plugin creates two Swift files: **ContentView.swift** and **iOSApp.swift**. These two files make up something like a 'Hello World' app. They show a text field in the center of the screen with the word "Hello". The plugin adds several files to make development easier.

App

Open **iOSApp.swift**:



```
iosApp
04-developing-ui-ios-swiftui

iOSApp.swift

iosApp > iOSApp > iOSApp.swift > body

1 import SwiftUI
2
3 @main
4 struct iOSApp: App {
5     var body: some Scene {
6         WindowGroup {
7             ContentView()
8         }
9     }
10}
11
```

Fig. 4.6 – *iOSApp.swift*

The starting point in a SwiftUI app is a `struct` that's marked with the `@main` attribute above the `struct`. This `struct` usually implements the `App` protocol. The `App` protocol requires you to create a variable named `body` that returns a `Scene`. A `Scene` is a container for the root view of a view hierarchy. A `WindowGroup` is a `Scene` and is also a container for your views. On iOS, this will contain only one window, but on macOS and iPadOS, it can contain multiple windows. Since it's a single expression, a return isn't required. None of the names of these files are special — the only important piece is to instruct the compiler where to start the app, and you do that with the `@main` tag.

Since **iOSApp** doesn't describe what your app does, rename the file to **TimezoneApp** by selecting it in the left sidebar and pressing **Return**. Type **TimezoneApp** and press return again. Next, change `struct iOSApp: App` to `struct TimezoneApp: App`.

Next, add the following code before `var body` to change the color of the tab bar to a nice shade of blue:

```
init() {
    let tabBarItemAppearance = UITabBarItemAppearance()
    tabBarItemAppearance.configureWithDefault(for: .stacked)
    tabBarItemAppearance.normal.titleTextAttributes =
    [.foregroundColor: UIColor.black]
    tabBarItemAppearance.selected.titleTextAttributes =
    [.foregroundColor: UIColor.white]
    tabBarItemAppearance.normal.iconColor = .black
    tabBarItemAppearance.selected.iconColor = .white

    let appearance = UITabBarAppearance()
    appearance.configureWithOpaqueBackground()
    appearance.stackedLayoutAppearance = tabBarItemAppearance
    appearance.backgroundColor = .systemBlue

    UITabBar.appearance().standardAppearance = appearance
    if #available(iOS 15.0, *) {
        UITabBar.appearance().scrollEdgeAppearance = appearance
    }
}
```

Build the app from the **Product** menu.

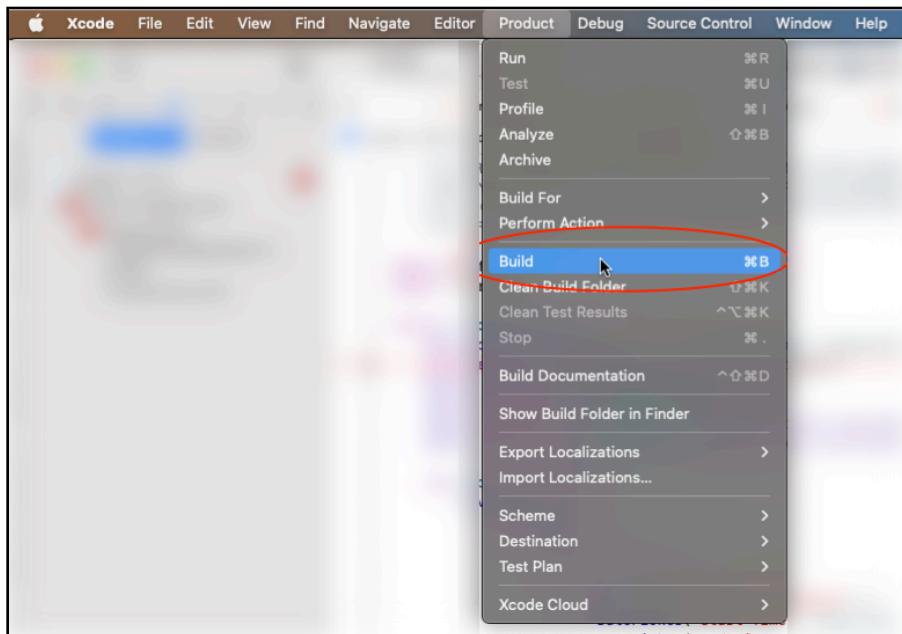


Fig. 4.7 – Xcode Build Menu Item

Next, run the app in an iPhone simulator.

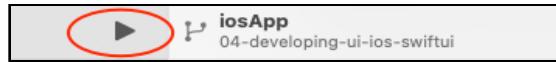


Fig. 4.8 – Xcode Run Button

It should look like this:



Fig. 4.9 – Starter Screen on iOS

You can also run the app in Android Studio. In Android Studio, make sure iOSApp is selected from the configuration menu:

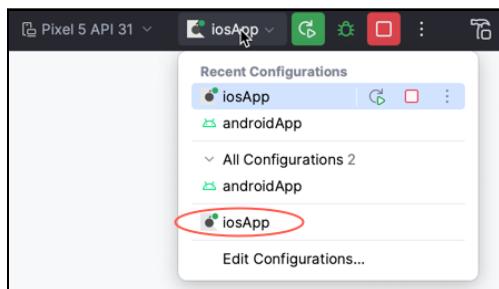


Fig. 4.10 – Android Studio Configuration List

Then, press the **Run** button:

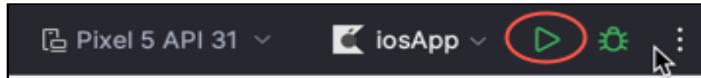


Fig. 4.11 – Android Studio Run Button

ContentView

Open **ContentView.swift**. Delete `Text("Hello")`. Add the following as the first line in the struct:

```
@StateObject private var timezoneItems = TimezoneItems()
```

Like `remember` in Jetpack Compose (JC), `StateObject` creates an observable object that's created once. Each time the view is redrawn, it will reuse the existing object. Other objects can listen for changes, and SwiftUI will update those objects. If you open the **TimezoneItems.swift** file, you'll see that it's an `ObservableObject` that **Publishes** a list of time zones and selected time zones. It also asynchronously gets the list of time zones from the shared library.

TabView

TabView is the SwiftUI equivalent of Jetpack Compose's `BottomNavigation`. You can use it to display a tab bar at the bottom of the screen and let the user switch between different views of the app.

Back in **ContentView.swift**, define body as follows:

```
var body: some View {
    // 1
    TabView {
        // 2
        TimezoneView()
        // 3
        .tabItem {
            Label("Time Zones", systemImage: "network")
        }
    // 4
    // FindMeeting()
    //     .tabItem {
    //         Label("Find Meeting", systemImage: "clock")
    //     }
    }
    .accentColor(Color.white)
    // 5
```

```
    .environmentObject(timezoneItems)  
}
```

1. Create a SwiftUI TabView.
2. The first tab will be the TimezoneView that you'll create next.
3. Apply the `tabItem` with a system network icon and the word **Time Zones**.
4. The second tab will be the `FindMeeting` view that you haven't created yet. (It's commented out for now.)
5. Set the `timezoneItems` object as an `environmentObject`.

There are several ways to pass objects around to different views. Here, you pass `timezoneItems` via an Environment Object. The users of this object i.e., any child view, will declare an `@EnvironmentObject` variable that will receive that object.

Time Zone View

Right-click in the **iosApp** folder and select **New File....**

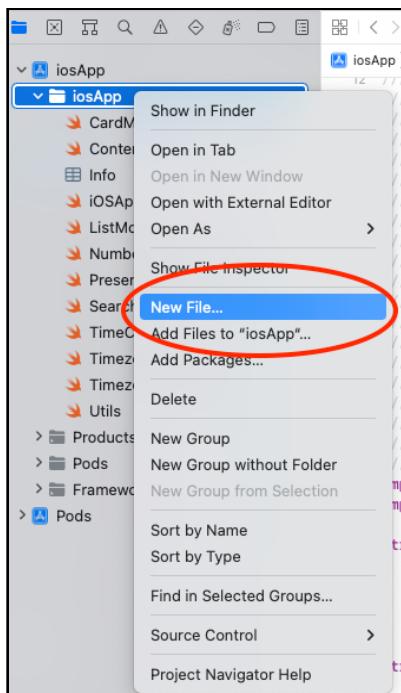


Fig. 4.12 – Xcode File Options

Next, select **SwiftUI View** file and click **Next**:

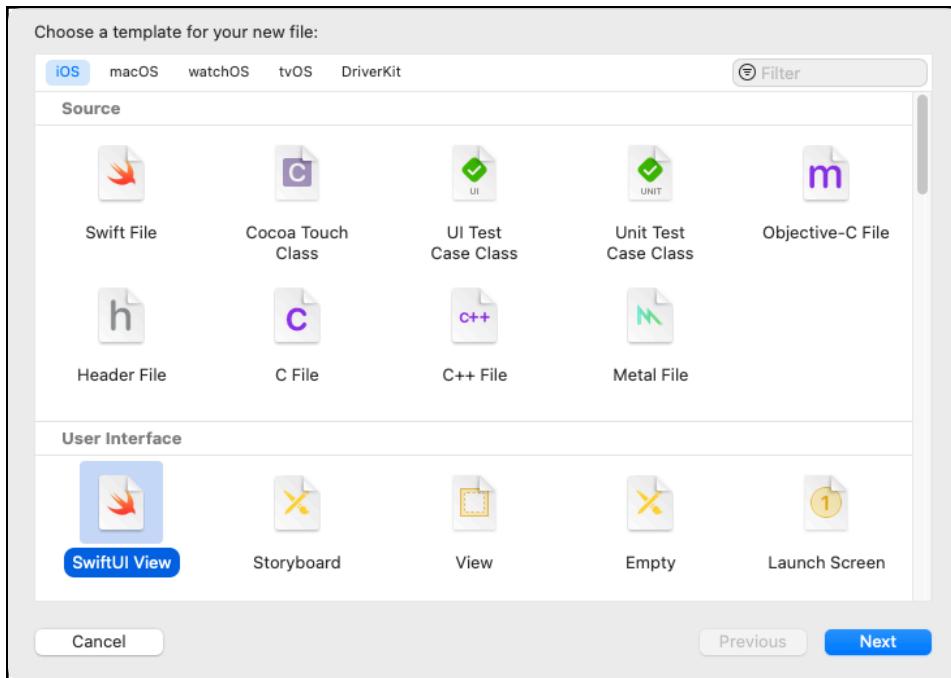


Fig. 4.13 – Xcode New File Type Dialog

Then, save as **TimezoneView.swift**:

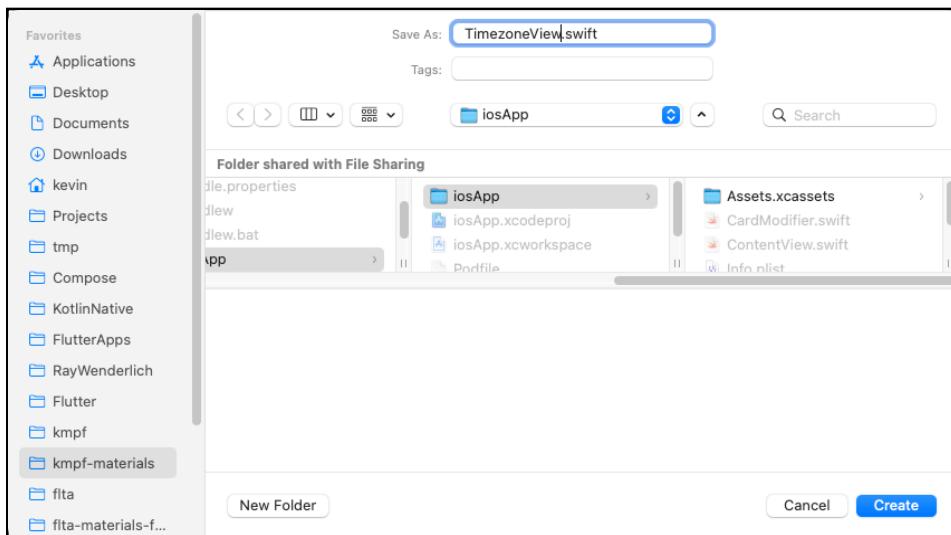


Fig. 4.14 – Xcode New File Name Dialog

Inside the file, first add the import for the shared library under the SwiftUI import:

```
import shared
```

Inside of **struct TimezoneView** add the following variables:

```
// 1
@EnvironmentObject private var timezoneItems: TimezoneItems
// 2
private var timezoneHelper = TimeZoneHelperImpl()
// 3
@State private var currentDate = Date()
// 4
let timer = Timer.publish(every: 1000, on: .main,
in: .common).autoconnect()
// 5
@State private var showTimezoneDialog = false
```

1. This is the `timezoneItems` object passed in from `ContentView`.
2. Create an instance of `TimeZoneHelperImpl`.
3. Get the current date.
4. Create a timer to update every second.
5. State variable on whether to show the time zone dialog.

`@State` is used with simple struct types, and its state is saved between redraws. Any `@State` property wrapper means the current view owns this data. SwiftUI keeps track of when this `@State` variable changes and redraws the view when its value changes.

`@StateObject` is used with classes. You'll mostly see `@State` used as SwiftUI views are structs.

Replace `Text("Hello, World")` with the following code:

```
// 1
NavigationView {
// 2
    VStack {
// 3
        TimeCard(timezone: timezoneHelper.currentTimeZone(),
                  time: DateFormatter.short.string(from:
currentDate),
                  date: DateFormatter.long.string(from: currentDate))
        Spacer()
// TODO: Add List
    } // VStack
// 4
```

```
.onReceive(timer) { input in
    currentDate = input
}
.navigationTitle("World Clocks")
// TODO: Add toolbar
} // NavigationView
```

1. A `NavigationView` allows you to display new screens with a title and will animate the view.
2. A `VStack` is a vertical stack. It's basically the same as a `Column` in JC.
3. Call the `TimeCard` class to show the time zone in a nice card format. Use the `short` and `long` `DateFormatter` extensions from the `Utils` class.
4. Use your timer. Every time the timer changes, update the date, which will then update the other elements.

If you look at the `Utils.swift` file, you'll see the definition of the `short` and `long` `DateFormatter` extension fields. Go ahead and run the app. Here's what it will look like:



Fig. 4.15 – World Clocks Screen

List of Time Zones

Next, replace // TODO: Add List with:

```
// 1
List {
    // 2
    ForEach(Array(timezoneItems.selectedTimezones), id: \.self) { timezone in
        // 3
        NumberTimeCard(timezone: timezone,
                        time: timezoneHelper.getTime(timezoneId: timezone),
                        hours: "\n\(timezoneHelper.hoursFromTimeZone(otherTimeZoneId: timezone))\nhours from local",
                        date: timezoneHelper.getDate(timezoneId: timezone))
            .withListModifier()
    } // ForEach
    // 4
    .onDelete(perform: deleteItems)
} // List
// 5
.listStyle(.plain)
.Spacer()
```

1. Create a List of items.
2. Create an array of selected time zones, and create a card for each one.
3. Show the time zone in a nice time card. Use a custom list modifier to remove the row separator and insets. (See **ListModifier.swift**.)
4. Add the ability to swipe to delete. You'll define the `deleteItems` method later.
5. Make the list style plain.

The `ForEach` is a special SwiftUI view struct that can return a View, unlike a regular `forEach()` function.

Next, replace // TODO: Add toolbar with the following code:

```
// 1
.toolbar {
    // 2
    ToolbarItem(placement: .navigationBarTrailing) {
        // 3
        Button(action: {
            showTimezoneDialog = true
        }) {
```

```
        Image(systemName: "plus")
            .frame(alignment: .trailing)
            .foregroundColor(.black)
    }
} // ToolbarItem
} // toolbar
```

1. Add a Toolbar item to the **NavigationView**.
2. Place it on the trailing edge (right side for languages that read left to right).
3. Create a **Button** with a plus sign that will set the `showTimezoneDialog` variable to true when pressed.

Next, add the following code after `// NavigationView`:

```
.fullScreenCover(isPresented: $showTimezoneDialog) {
    TimezoneDialog()
        .environmentObject(timezoneItems)
}
```

`fullScreenCover` is a way to present a full-screen modal view over your current view. This will show the time zone dialog as a full-screen sheet. Since it's modal, there has to be a way to dismiss it. So, there's a dismiss button in the dialog for that.

The button in the toolbar sets the `showTimezoneDialog` variable to true, which is a state variable managed by SwiftUI. When this value changes, the full-screen modal is shown.

Next, add the `deleteItems` method after the `var` body code:

```
func deleteItems(at offsets: IndexSet) {
    let timezoneArray = Array(timezoneItems.selectedTimezones)
    for index in offsets {
        let element = timezoneArray[index]
        timezoneItems.selectedTimezones.remove(element)
    }
}
```

The code above goes through the indices in the `IndexSet`, finds the time zone selected, and removes it from your selected list. Build and run the app. Click the + button at the top.

You will see:

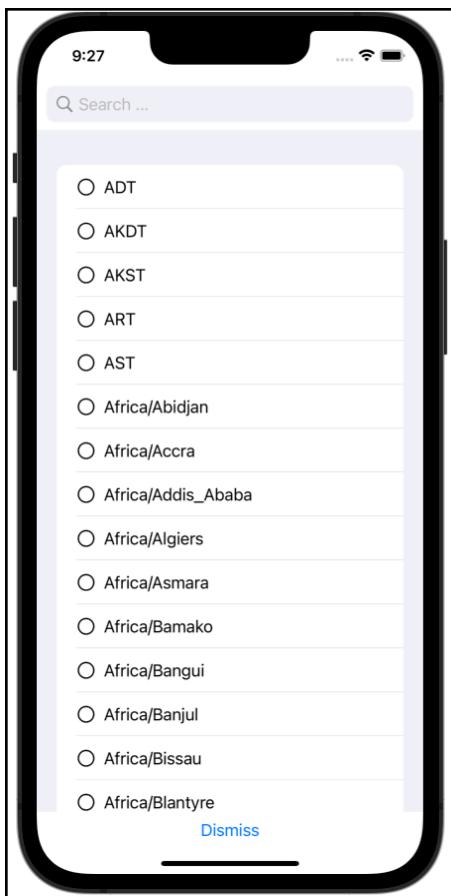


Fig. 4.16 – Search Time Zones Screen

Try searching for your favorite time zones, select the time zone and then search again.

When you search for New York, here's what you'll see:



Fig. 4.17 – Search Time Zones Screen

When you're finished, tap the **Dismiss** button.

This is what it looks like with New York and Lisbon:

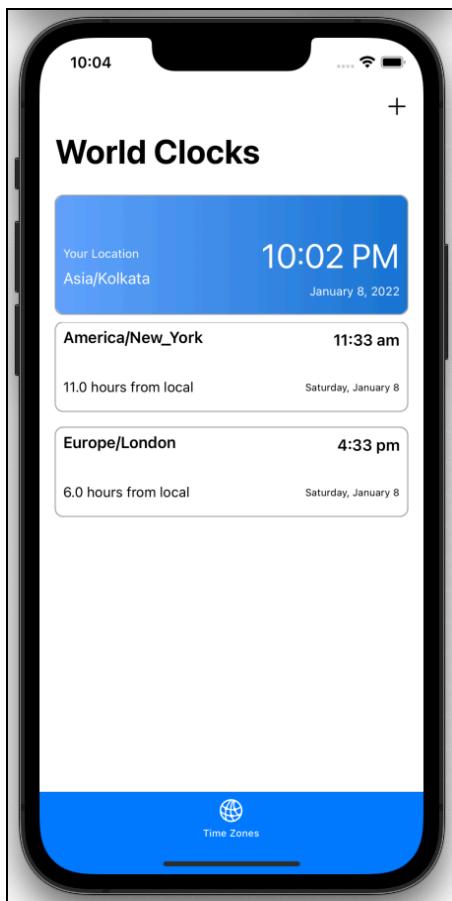


Fig. 4.18 — Selected Time Zones

If you want to delete a time zone, simply swipe left:

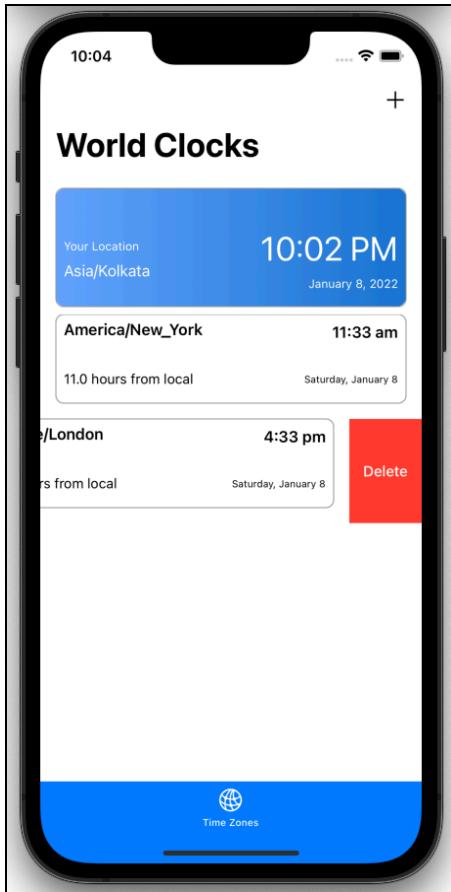


Fig. 4.19 — Delete Selected Time Zone

Hour Sheet

You'll want to show the hours that are available to meet. You can do that by showing the hours in a sheet, which in this case, is a modal dialog. This is a simple view with a list of hours and a dismiss button. Create a new SwiftUI View named **HourSheet.swift** in the `iosApp` folder. Remove the `Text` view, and then add the following two variables right before the body definition:

```
@Binding var hours: [Int]
@Environment(\.presentationMode) var presentationMode
```

The first variable is an array of hours the caller will pass in. The second one is the `showHoursDialog` Boolean. This will hide the dialog by setting this variable to false. Add the following inside body:

```
// 1
NavigationView {
    // 2
    VStack {
        // 3
        List {
            // 4
            ForEach(hours, id: \.self) { hour in
                Text("\(hour)")
            }
        } // List
    } // VStack
    .navigationTitle("Found Meeting Hours")
    // 5
    .toolbar {
        ToolbarItem(placement: .navigationBarTrailing) {
            Button(action: {
                presentationMode.wrappedValue.dismiss()
            }) {
                Text("Dismiss")
                    .frame(alignment: .trailing)
                    .foregroundColor(.black)
            }
        } // ToolbarItem
    } // toolbar
} // NavigationView
```

1. Use a `NavigationView` to show a toolbar.
2. Use a `VStack` for the title.
3. Use a `List` to show each hour.
4. Use the `ForEach` view to show a `Text` view for each hour.
5. Show a Toolbar with a Dismiss button.

This creates a list for each hour and shows it in a `Text` view. To get the preview to work, change the `HourSheet()` constructor inside `HourSheet_Previews` to:

```
HourSheet(hours: .constant([8, 9, 10]))
```

Find Meeting

The next screen is the find meeting screen. This is the screen where you can choose the hours you want to search for meetings and then find the hours that work for everyone. Create a new SwiftUI View file named **FindMeeting.swift**.

First, add the `shared` import under the `SwiftUI` import:

```
import shared
```

Then, add the following variables before `var body`:

```
// 1
@EnvironmentObject private var timezoneItems: TimezoneItems
// 2
private var timezoneHelper = TimeZoneHelperImpl()
// 3
@State private var meetingHours: [Int] = []
@State private var showHoursDialog = false
// 4
@State private var startDate =
Calendar.current.date(bySettingHour: 8, minute: 0, second: 0,
of: Date())!
@State private var endDate =
Calendar.current.date(bySettingHour: 17, minute: 0, second: 0,
of: Date())!
```

1. Create a `timezoneItems` environment variable. This will come from **ContentView**.
2. Create an instance of the `TimeZoneHelperImpl` class.
3. An array of meeting hours that all can meet at.
4. The start and end dates are 8 a.m. and 5 p.m.

This gives you all the variables you'll need for your screen. Now you can start work on the body. Remove `Text("Hello, World")` and add the following code inside `body`:

```
NavigationView {
    VStack {
        Spacer()
            .frame(height: 8)
        // TODO: Add Form
    } // VStack
    .navigationTitle("Find Meeting Time")
    // TODO: Add sheet
```

```
    } // NavigationView
```

This will be a vertical stack with a navigation view, which has a title and some spacers around the title. Now, add the form that has two sections: a time range with the start and end time pickers and the list of time zones selected. Replace TODO: Add Form with:

```
Form {
    Section(header: Text("Time Range")) {
        // 1
        DatePicker("Start Time", selection: $startDate,
displayedComponents: .hourAndMinute)
        // 2
        DatePicker("End Time", selection: $endDate,
displayedComponents: .hourAndMinute)
    }
    Section(header: Text("Time Zones")) {
        // 3
        ForEach(Array(timezoneItems.selectedTimezones), id: \.self) { timezone in
            HStack {
                Text(timezone)
                Spacer()
            }
        }
    }
} // Form
// TODO: Add Button
```

1. Start time date picker.
2. End time date picker.
3. List of selected time zones.

Now comes the button that does the time zone calculation. It will call the shared library's search method. Replace // TODO: Add Button with:

```
Spacer()
Button(action: {
    // 1
    meetingHours.removeAll()
    // 2
    let startHour = Calendar.current.component(.hour, from:
startDate)
    let endHour = Calendar.current.component(.hour, from: endDate)
    // 3
    let hours = timezoneHelper.search(
        startHour: Int32(startHour),
        endHour: Int32(endHour),
```

```
    timezoneStrings: Array(timezoneItems.selectedTimezones))  
    // 4  
    let hourInts = hours.map { kotlinHour in  
        Int(truncating: kotlinHour)  
    }  
    meetingHours += hourInts  
    // 5  
    showHoursDialog = true  
, label: {  
    Text("Search")  
        .foregroundColor(Color.black)  
}  
Spacer()  
    .frame(height: 8)
```

1. Clear your array of any previous values.
2. Get the start and end hours.
3. Call the shared library search method, converting the hours to ints.
4. Create another array of ints from the hours returned. Convert to iOS ints.
5. Set the flag to show the hours dialog.

Notice that there is a bit of conversion going on. You need to convert the Swift **Int** to **32bit Int** for Kotlin. Then, when you get the value back from the shared library, you need to convert the values back to Swift Int. Now that the button sets the flag to show the hours dialog, you need a way of showing that dialog. You'll use a sheet — a type of dialog that shows up at the bottom of the screen. Replace // TODO: Add sheet with:

```
.sheet(isPresented: $showHoursDialog) {  
    HourSheet(hours: $meetingHours)  
}
```

You are almost there. Finally, you need to add the **Find Meeting** tab to the TabView.

ContentView

Return to **ContentView** and uncomment the **FindMeeting** section.

Build and run the app. Try to add several time zones. Go to the **Find Meeting** page, tap the **Search** button and see if any hours show up. If you have problems and don't see any hours, start with a one-time zone and work your way up to more. It's quite possible that there are no compatible hours. Try increasing your end time to 17 or 19. That will increase the range.

Here's an example of hours between Los Angeles and New York time zones:

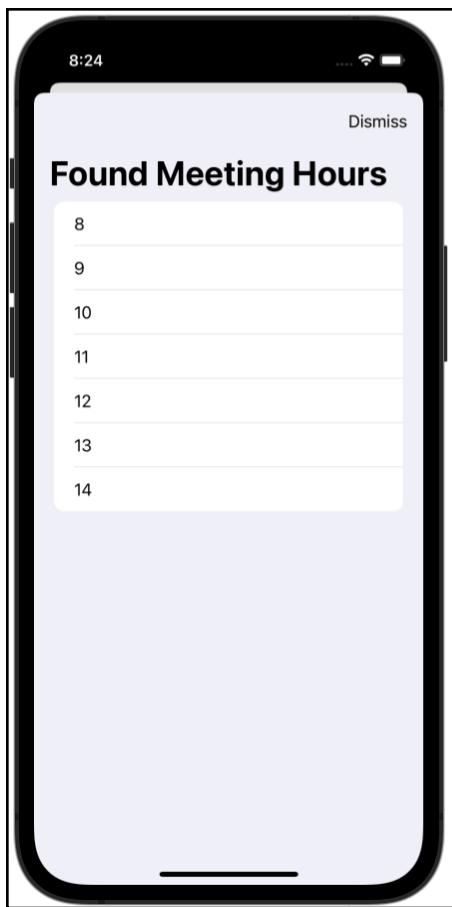


Fig. 4.20 – Found Meeting Hours Screen

Congratulations! You now have both an Android and iOS app that you can show off to your friends.

Key Points

- SwiftUI is a new declarative way to create UIs for Apple platforms.
- You can use Xcode or Android Studio to develop your SwiftUI code.
- Use `@State`, `@StateObject`, `@ObservedObject` and `@EnvironmentObject` for holding state.
- Use SwiftUI views like `VStack`, `HStack`, `NavigationView` and `Text` to build your UIs.
- Use `List` views to show many items.
- `ForEach` can return a view that you can use inside `List` as well as other views.
- Use `sheet` and `fullScreenCover` for modal dialog-type screens.
- Use `Int32` to convert integers for Kotlin.
- Use `Int` to convert Kotlin integers to Swift integers.

Where to Go From Here?

To learn about:

Xcode: <https://developer.apple.com/xcode/>

SwiftUI:

- The SwiftUI Apprentice book: <https://www.kodeco.com/books/swiftui-apprentice>
- Official SwiftUI documentation: <https://developer.apple.com/documentation/swiftui/>
- The kodeco.com video course library on SwiftUI: https://www.kodeco.com/library?q=swiftui&domain_ids%5B%5D=1&content_types%5B%5D=collection
- `@StateObject` documentation: <https://developer.apple.com/documentation/swiftui/stateobject>

Congratulations! You've written a SwiftUI app that uses a shared library for the business logic. Now that you have both the Android and the iOS apps written, the next chapter will show you how to create a desktop app.

Chapter 5: Developing UI: Compose Multiplatform

By Kevin Moore

If you come from a mobile background, it's exciting to know that you can build desktop apps with the knowledge you gained from learning Jetpack Compose (JC). JetBrains, the maker of the technology behind Android Studio and IntelliJ, has worked with Google to create Compose Multiplatform (CM). This uses some of the same code from Jetpack Compose and extends it to be used for multiple platforms. This chapter will focus on building UI for the desktop using CM, but it will work on the web (experimental) as well. Currently, CM is in alpha for iOS which is pretty exciting too!

Getting to Know Compose Multiplatform

CM uses the Java Virtual Machine (JVM) under the hood so that you can still use the older **Swing** technology for creating UI if you want. It uses the **Skia** graphics library that allows hardware acceleration (like JC). Finally and most importantly, apps built with CM can run on macOS, Windows and Linux.

Differences in Desktop

Before moving ahead, we need to understand the differences between mobile and desktop app and their requirements. This will help us understand and finally build the desktop app. Unlike mobile, the desktop has features like menus, multiple windows and system notifications. Menus can also have shortcuts and windows will have different sizes and positions on the screen. Lastly, the desktop doesn't usually use app bars like mobile apps. You'll usually use menus to handle actions instead.

Creating a Desktop App

To create a desktop app, you're going to do several things:

1. Update the existing Gradle files.
2. Create a desktop module.
3. Create a shared UI module.
4. Move most of the Android code to the shared UI module.
5. Create some wrappers so that Android and desktop can have unique functionality.

As usual, the desktop module will contain the desktop platform-specific code. The shared UI module will contain the UI code that you will use cross-platform. In this chapter, those platforms are - Android, Mac, and Windows.

Updating Gradle Files

To start, you'll need to update a few of your current Gradle files. Open the starter project in Android Studio and open the main **settings.gradle.kts**. Under **mavenCentral** and at the end of **pluginManagement/repositories** add:

```
maven("https://maven.pkg.jetbrains.space/public/p/compose/dev")
```

This adds the repository for the Compose Multiplatform library. Similarly, under **dependencyResolutionManagement/repositories** add:

```
maven("https://maven.pkg.jetbrains.space/public/p/compose/dev")
```

Press the **Sync Now** button at the top of this file.

Now, go to **shared** ▶ **build.gradle.kts** file. You already have the target setup for the desktop:

```
jvm("desktop")
```

The code above creates a new JVM target with the name **desktop**. Next, you will create desktop module for writing your desktop platform specific code.

Desktop Module

There isn't an easy way to create a desktop module, except by hand. At the time of writing, JetBrains is working to improve this but it isn't that hard to do it manually. Right-click the top-level folder in the project window and choose **New** ▶ **Directory**:

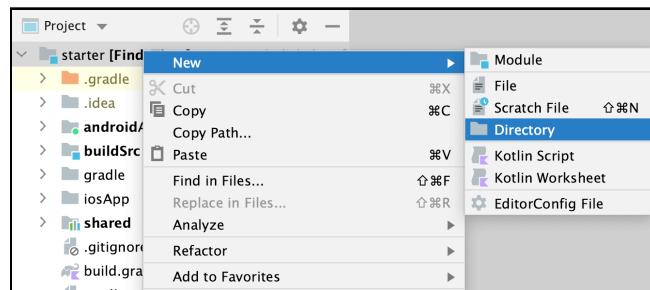


Fig. 5.1 – Creating a New Directory

Name the directory **desktop**. Next, right-click on the **desktop** folder and choose **New ▶ File**. Name the file **build.gradle.kts**. This build file is similar to the **shared** module's build file. Now, add the following:

```
import org.jetbrains.compose.desktop.application.dsl.TargetFormat

// 1
plugins {
    kotlin("multiplatform")
    alias(libs.plugins.composePlugin)
}

// 2
kotlin {
    // TODO: Add Kotlin
}

// 3
// TODO: Add Compose Desktop
```

Here's an explanation of the above code:

1. Adds the Multiplatform and Desktop Compose plugins.
2. It contains a **TODO** that you will soon complete. It will set the **jvm** version that will be used, specify the source Kotlin source files for the shared codes that will be used, and any dependencies you will need.
3. This also contains a **TODO**. It will setup the Kotlin desktop settings that you will see in moment.

Starting with first todo. Replace `// TODO: Add Kotlin` with the following code:

```
// 1
jvm {
    compilations.all {
        kotlinOptions.jvmTarget = "17"
    }
}
// 2
sourceSets {
    val jvmMain by getting {
        // 3
        kotlin.srcDirs("src/jvmMain/kotlin")
        dependencies {
            // 4
            implementation(compose.desktop.currentOs)
            // 5
            api(compose.runtime)
```

```
        api(compose.foundation)
        api(compose.material)
        api(compose.ui)
        api(compose.materialIconsExtended)

        // 6
    implementation(project(":shared"))
    implementation(project(":shared-ui"))
}
}
```

Here's what the above code does:

1. Set up a JVM target that uses Java 17 (11 or above is required).
2. Set up a group of sources and resources for the JVM.
3. Set the source directory path where the Kotlin files for UI will be located.
4. Use the pre-defined variable to bring in the current OS library for Compose.
5. Bring in the Compose libraries, using the variables defined in the Compose plugin.
6. Import your shared libraries. Leave **shared-ui** commented out until you create it.

There's a lot here, but the desktop Gradle setup is a bit more complex. This sets up the libraries and source file locations needed for the desktop module.

Moving ahead, replace `// TODO: Add Compose Desktop` with the following:

```
// 1
compose.desktop {
    // 2
    application() {
        // 3
        mainClass = "MainKt"
        // 4
        nativeDistributions {
            targetFormats(TargetFormat.Dmg, TargetFormat.Msi,
TargetFormat.Deb)
            packageName = "FindTime"
            macOS {
                bundleID = "com.kodeco.findtime"
            }
        }
    }
}
```

Here's what you have done:

1. Configuration for Compose desktop.
2. Define a desktop application with the details below.
3. Set the main class. You'll create a **Main.kt** file in a bit.
4. Set up packaging information for when you're ready to ship. Here, you have provided three types of information - target formats, package name, and in the case of macOS bundle ID.

Click **Sync Now** from the top right portion of the window. Open **settings.gradle.kts** from the root directory and add the new project at the end of the file:

```
include(":desktop")
```

Do another sync.

Next, right-click on the **desktop** folder and choose **New > Directory**. Add **src/jvmMain/kotlin** as the new directory name. This will create three folders: **src**, **jvmMain**, and **kotlin** each inside the previous one. Next, right-click on the **kotlin** folder and choose **New > Kotlin Class/File**. Type **Main** and press return. This will create **Main.kt** file inside **src/jvmMain/kotlin**.

Now, replace the contents of the file with the following code:

```
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.material.Surface
import androidx.compose.ui.Modifier
import androidx.compose.ui.window.Window
import androidx.compose.ui.window.application
import androidx.compose.ui.window.rememberWindowState

// 1
fun main() {
    // 2
    application {
        // 3
        val windowState = rememberWindowState()

        // 4
        Window(
            onCloseRequest = ::exitApplication,
            state = windowState,
            title = "TimeZone"
        ) {
            // 5
            Surface(modifier = Modifier.fillMaxSize()) {
```

```
        } // TODO: Add Theme and MainView
    }
}
```

Here's what the above code does:

1. Entry point to the application. Just like in Kotlin or Java programs, the starting function is `main`.
2. Create a new application.
3. Remember the current default window state. Change this if you want the window positioned in a different position or size.
4. Create a new window with the window state. If the user closes the window, exit the application.
5. Set up a `Surface` that takes the full screen. `Surface` is a composable that implements material surface.

Other than the commented `TODO`, this is the extent of the desktop code. The next task is to create a **shared-ui** module where you will move the Compose files.

Shared UI

You created the Android Compose files earlier. And you put a lot of work into those files too. You could duplicate those files for the desktop, but why not share them? That's the idea behind the **shared-ui** module. You'll move the Android files over and make a few modifications to allow them to be used for both Android and the desktop.

From the project window, right-click on the top-level folder and choose **New ▶ Directory**. Name the directory **shared-ui**. Next, right-click on the **shared-ui** folder and choose **New ▶ File**. Name the file **build.gradle.kts**.

Add the following:

```
plugins {
    kotlin("multiplatform")
    id("com.android.library")
    alias(libs.plugins.composePlugin)
}

kotlin {
```

```
// TODO: Add Desktop Info  
}  
  
android {  
    // TODO: Add Android Info  
}
```

The above code is similar to what you did for the build Gradle file for desktop. The only change is that instead of desktop configuration you are providing minimum android configuration. You will see it shortly. For Kotlin, replace // TODO: Add Desktop Info with:

```
// 1  
androidTarget {  
    compilations.all {  
        kotlinOptions {  
            jvmTarget = JavaVersion.VERSION_17.toString()  
        }  
    }  
}  
// 2  
jvm("desktop") {  
    compilations.all {  
        kotlinOptions.jvmTarget = "17"  
    }  
}  
  
sourceSets {  
    val commonMain by getting {  
        // 3  
        dependencies {  
            implementation(project(":shared"))  
            api(compose.foundation)  
            api(compose.runtime)  
            api(compose.foundation)  
            api(compose.material)  
            api(compose.material3)  
            api(compose.materialIconsExtended)  
            api(compose.ui)  
            api(compose.uiTooling)  
        }  
    }  
    // 4  
    val androidMain by getting {  
        dependencies {  
            implementation(project(":shared"))  
        }  
    }  
    // 5  
    val desktopMain by getting {  
        dependencies {
```

```
        implementation(project(":shared"))
    }
}
```

Here's what the following code does:

1. Set an Android target.
2. Set a desktop target.
3. Define the common main sources. This includes the shared library and Compose for Desktop.
4. Set Android's dependencies.
5. Set desktop's dependencies.

Note that this defines the dependencies for each target. The shared project is used by all targets but the common target gets the Compose dependencies.

Now replace // TODO: Add Android Info with:

```
namespace = "com.kodeco.findtime.android"
compileSdk = 34
defaultConfig {
    minSdk = 26
}
compileOptions {
    sourceCompatibility = JavaVersion.VERSION_17
    targetCompatibility = JavaVersion.VERSION_17
}
```

This sets the namespace, compile Android version, the minimum supported version of Android and the Java version to compile with.

One of the nice features of CM is that it can be used with both Android and desktop. For the **shared-ui** folder, you'll need three different source directories. One for Android, one for a common source and one for desktop. Right-click on **shared-ui** and choose **New > Directory**.

Type **src/androidMain/kotlin/com/kodeco/compose/ui** and press return to create a new director with the following path.

This will create several folders. Next, do the same for **commonMain**. Select the **src** directory you just created and create a new directory named **commonMain/kotlin/com/kodeco/compose**.

Last but not the least. Do the same for the desktop. Create a new directory using: **desktopMain/kotlin/com/kodeco/compose/ui**.

This will create three main directories. The first for Android, the second for all common code and the third for the desktop.

Open **settings.gradle.kts** from the root directory and add the new project:

```
include(":shared-ui")
```

Click **Sync Now** to sync all the changes.

Move UI Files

Now comes the fun part. Instead of recreating all of the Compose UI for the desktop, you'll steal it from Android. From **androidApp/src/main/java/com/kodeco/findtime/android/**, select the **ui** folder and drag it to **shared-ui/src/commonMain/kotlin/com/kodeco/compose** folder. You'll get a conflicts dialog but go ahead and press continue. You'll fix these problems next. Then move the **MyApplicationTheme.kt** file to **shared-ui/src/commonMain/kotlin/com/kodeco/compose/ui** as well (press continue on the conflicts dialog).

Update Android app

While moving all the UI code was great for the desktop, it broke the Android app. But, you can fix that. First, you need to update the **build.gradle.kts** file in the **androidMain** module. Add the **shared-ui** library after the **shared** library:

```
implementation(project(":shared-ui"))
```

Run a Gradle sync and build the Android app. You'll see a lot of errors that you'll fix next.

AddTimeZoneDialog

Open **AddTimeZoneDialog.kt** from the **commonMain/kotlin/com/kodeco/compose/ui** folder inside the **shared-ui** module. You'll see several errors for the following imports:

```
import androidx.compose.ui.res.stringResource
import com.kodeco.findtime.android.R
```

These two imports don't exist for the **shared-ui** module. Remove them. After the imports, add:

```
@Composable  
expect fun AddTimeDialogWrapper(onDismiss: onDismissType,  
content: @Composable () -> Unit)
```

This will show an error since it hasn't been implemented yet.

This is a Composable function that uses KMP's `expect` keyword. This means that each target this module uses needs to implement this function. Now, change the signature for the `AddTimeZoneDialog` function and the code up to the first `Surface` with the following code:

```
fun AddTimeZoneDialog(  
    onAdd: OnAddType,  
    onDismiss: onDismissType  
) {  
    val timezoneHelper: TimeZoneHelper = TimeZoneHelperImpl()  
  
    AddTimeDialogWrapper(onDismiss) {
```

Make sure to add a closing } at the end of the function. This just uses the `AddTimeDialogWrapper` function to wrap the existing code. The `AddTimeDialogWrapper` function will handle platform-specific code. Android will handle the dialog one way, and the desktop another way.

You can now remove the unused `Dialog` import:

```
import androidx.compose.ui.window.Dialog
```

One issue in using Compose for Desktop is resource handling. That's beyond the scope of this chapter. For now, you will just change the string resources to hard-coded strings. Make the following changes:

```
stringResource(id = R.string.cancel)
```

To:

```
"Cancel"
```

Similarly change:

```
stringResource(id = R.string.add)
```

To:

```
"Add"
```

From the **shared-ui/src/androidMain/kotlin/com/kodeco/compose/ui** folder, right-click and create a new Kotlin file named **AddTimeDialogWrapper.kt**. This will be the Android version that implements the expect defined in **commonMain**. Add:

```
import androidx.compose.runtime.Composable
import androidx.compose.ui.window.Dialog

@Composable
actual fun AddTimeDialogWrapper(onDismiss: onDismissType,
content: @Composable () -> Unit) {
    Dialog(
        onDismissRequest = onDismiss) {
        content()
    }
}
```

This creates a function that takes a dismiss callback and the content for the dialog. The reason you need this is that **Dialog** is specific to JC and not CM. Make sure that when the file is added, it's part of the **com.kodeco.compose.ui** package, if it isn't already.

Going ahead. Right-click on **desktopMain/kotlin/com/kodeco/compose/ui** and create the same class **AddTimeDialogWrapper.kt**.

Now, add the following code:

```
import androidx.compose.runtime.Composable
import androidx.compose.ui.Alignment
import androidx.compose.ui.window.DialogWindow
import androidx.compose.ui.window.WindowPosition
import androidx.compose.ui.window.rememberDialogState

@Composable
actual fun AddTimeDialogWrapper(onDismiss: onDismissType,
content: @Composable () -> Unit) {
    DialogWindow(onCloseRequest = { onDismiss() },
        state = rememberDialogState(
            position = WindowPosition(Alignment.Center),
        ),
        title = "Add Timezones",
```

```
        content = {
            content()
        }
    }
```

This class just uses the desktop `DialogWindow` instead of a `Dialog`. Here this `DialogWindow` takes a dismiss callback, a state, title and content. Luckily this is not that much code :]. The bulk of the Compose code is in `AddTimeZoneDialog`.

MeetingDialog

Much like `AddTimeZoneDialog`, you need to change `MeetingDialog`. Open `MeetingDialog.kt` and remove the imports that show up in red and the dialog import. Add another wrapper:

```
@Composable
expect fun MeetingDialogWrapper(onDismiss: onDismissType,
                                content: @Composable () -> Unit)
```

This is just like the other dialog wrapper. Now change the `MeetingDialog` method up to `Surface` with the following:

```
fun MeetingDialog(
    hours: List<Int>,
    onDismiss: onDismissType
) {
    MeetingDialogWrapper(onDismiss) {
```

This adds a wrapper around the dialog. Make sure to add a closing `}` like before.

Then change:

```
    stringResource(id = R.string.done)
```

To:

```
"Done"
```

From the `src/androidMain/kotlin/com/kodeco/compose/ui` folder, right-click and create a new Kotlin file called `MeetingDialogWrapper.kt`. This will be the Android version that implements the `expect` defined in `commonMain`.

Add the following code:

```
import androidx.compose.runtime.Composable
import androidx.compose.ui.window.Dialog

@Composable
actual fun MeetingDialogWrapper(onDismiss: onDismissType,
content: @Composable () -> Unit) {
    Dialog(
        onDismissRequest = onDismiss) {
        content()
    }
}
```

This creates a function that takes a dismiss callback and the content for the dialog. Right-click on **desktopMain/kotlin/com/kodeco/compose/ui** and create the same **MeetingDialogWrapper.kt** class. Add the following code:

```
import androidx.compose.runtime.Composable
import androidx.compose.ui.window.DialogWindow
import androidx.compose.ui.window.rememberDialogState

@Composable
actual fun MeetingDialogWrapper(onDismiss: onDismissType,
content: @Composable () -> Unit) {
    DialogWindow(
        onCloseRequest = { onDismiss() },
        title = "Meetings",
        state = rememberDialogState(),
        content = {
            content()
        })
}
```

This adds a close handler, a title of “Meetings”, the dialog state and the content. Next, open **AnimatedSwipeDismiss.kt**. You’ll see a bunch of errors. Click on the first error and hit option-return:

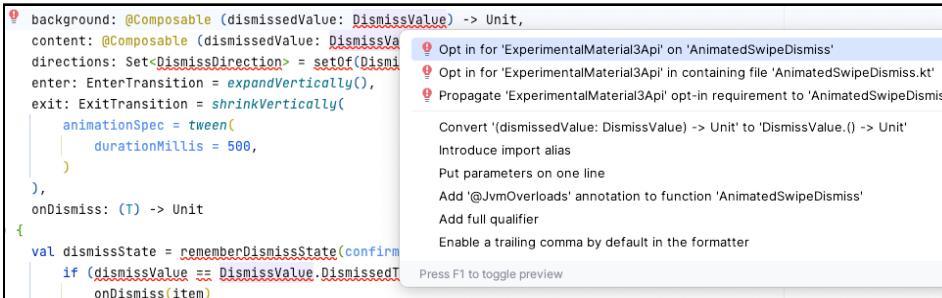


Fig. 5.2 – Edit Configurations

Choose the first entry to add the OptIn annotation. This will add `@OptIn(ExperimentalMaterial3Api::class)` above the class. This will allow you to use experimental Material3 API classes and all the errors will disappear.

If you try to run the Android app now, you will see a few more errors. The first one is in **LocalTimeCard**. Just delete the two bottom imports.

Next, in **MainView**, delete the last import. Finally in **TimeZoneScreen**, hit option-return to Opt in to use the Material3 experimental API by adding the following OptIn annotation - `@OptIn(ExperimentalMaterial3Api::class)`

Build and run the Android app. Phew! the app should finally start functioning as usual again. :]

Now that everything has been moved to the a shared ui module, you can try to run the app on the desktop. To run your new desktop app, you'll need to create a new configuration for running desktop app. From the configuration dropdown, choose **Edit Configurations**:

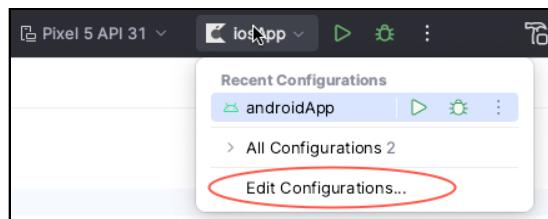


Fig. 5.3 – Edit Configurations

Next, click the plus symbol and choose **Gradle**.

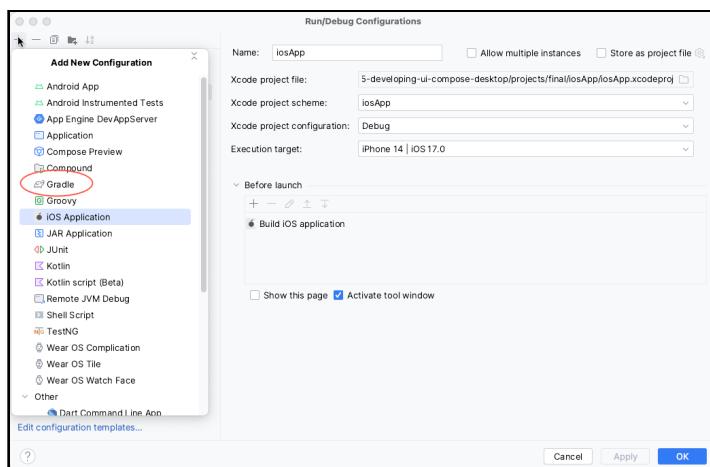


Fig. 5.4 – Add New Configuration

Then, do the following:

1. Set the **Name** to **Desktop**.
2. For the **Run** text field, enter the **desktop:run** command.

This will run the desktop:run Gradle task when Desktop configuration is selected which will start the desktop app.

Finally, click **OK**.

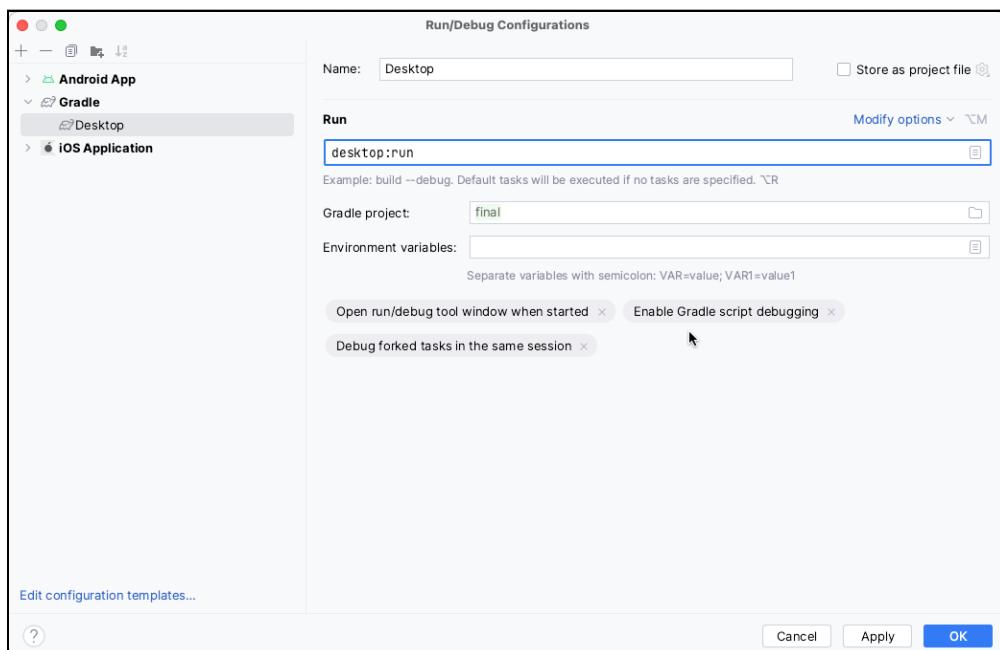


Fig. 5.5 – Desktop Configuration

Run the desktop app:



Fig. 5.6 – Run Desktop Configuration

Wait, what is this?

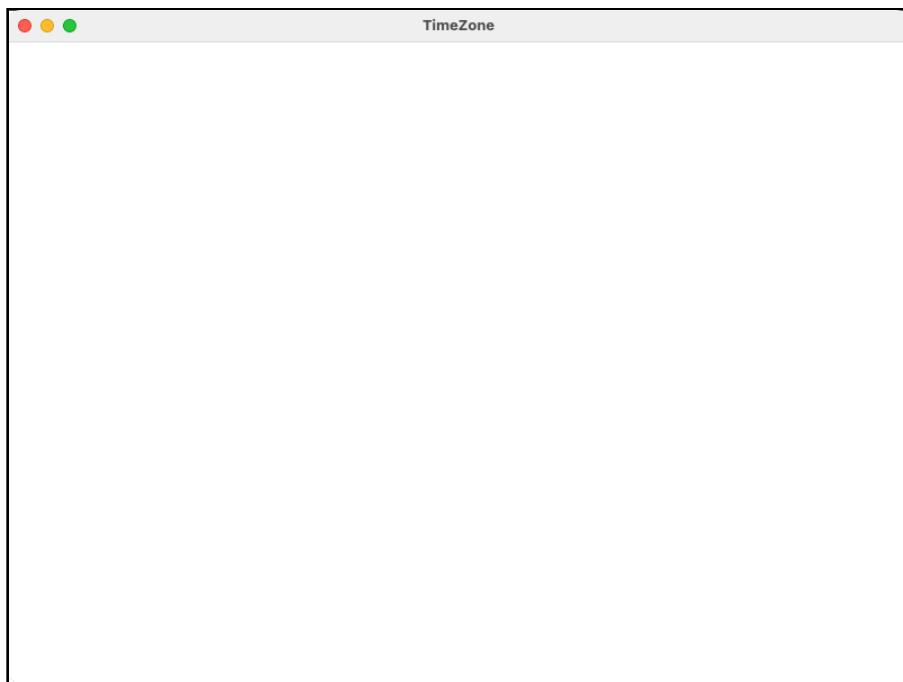


Fig. 5.7 – Blank Desktop app

The good news is the app ran. The bad news is there isn't any content. Do you know why? Right — you never added any content to **Main.kt**. Go back to **Main.kt** in the **desktop** module. Inside of Surface, replace // TODO: Add Theme and MainView:

```
MyApplicationTheme {  
    MainView()  
}
```

This shows errors. Any ideas? Take a look at the **desktop build.gradle.kts** file. Looks like you need to uncomment out the **shared-ui** project. You'll have to stop running the desktop to do any other Gradle tasks. Hit the red stop button, uncomment the **shared-ui** project and then resync Gradle.

Now, add the missing imports to **Main.kt** and run the app again.

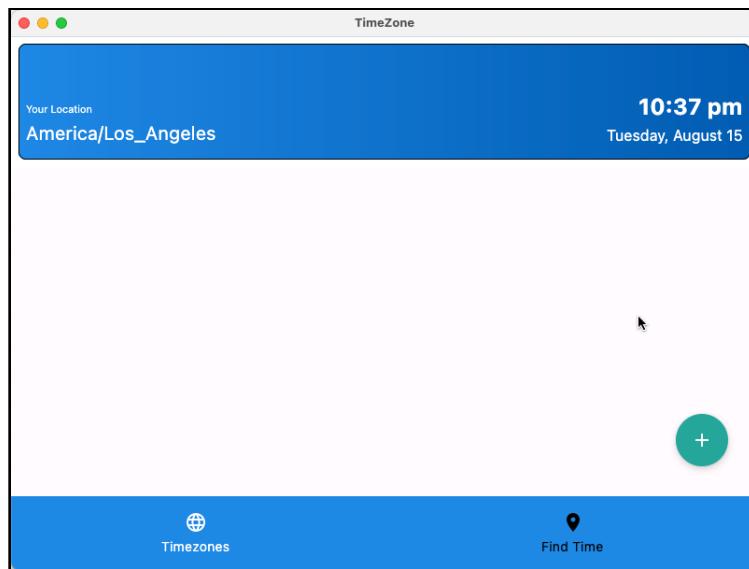


Fig. 5.8 – Desktop Time Zones screen

Much better. Try using the app by adding some time zones and see if you're missing anything.

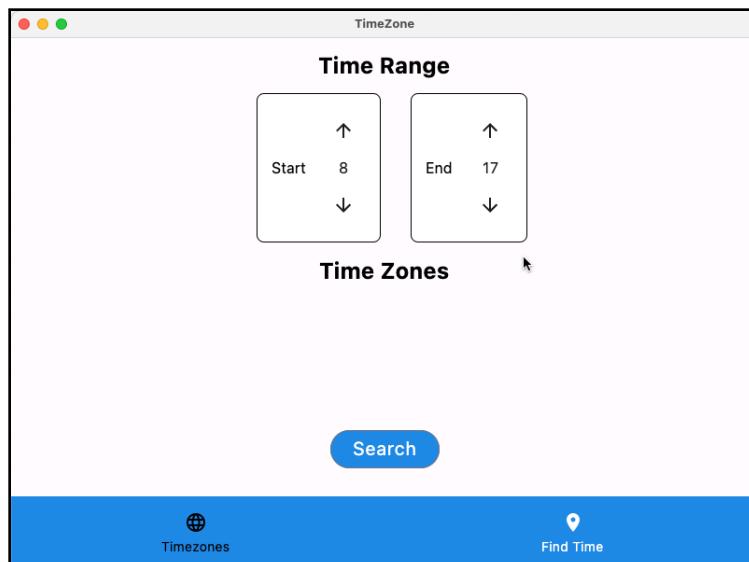


Fig. 5.9 – Desktop Time Range screen

Note: The window background can vary depending on whether you are using Dark Theme on your computer or not.

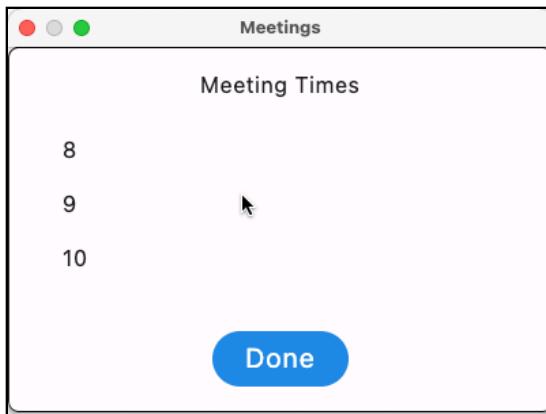


Fig. 5.10 – Desktop Meeting Times screen

Window Sizes

If you bring up the Add Timezones dialog, you'll see the buttons get cut off:

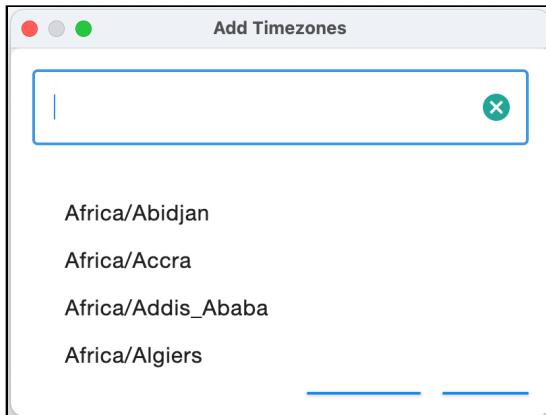


Fig. 5.11 – Desktop window buttons cropped

How can you fix that? Dialogs have a `DialogState` class that allows you to set the position and size. To fix this dialog, open `AddTimeDialogWrapper` inside `desktopMain` and replace the `rememberDialogState` method so that it looks like this:

```
state = rememberDialogState(  
    position = WindowPosition(Alignment.Center),  
    size = DpSize(width = 400.dp, height = Dp.Unspecified),  
) ,
```

This sets a fixed width of 400dp and an unspecified height. This will allow the height to expand to a good size.

For `MeetingDialogWrapper`, replace the `rememberDialogState` method with:

```
rememberDialogState(size = DpSize(width = 400.dp, height =  
Dp.Unspecified)),
```

Build and run the desktop app. You'll see that the buttons are no longer cropped:

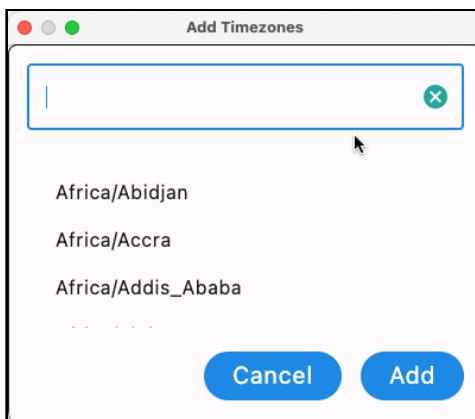


Fig. 5.12 – Desktop window buttons not cropped

Adding Multiple Windows Support

Your app can have a single window or multiple windows. If you have just one window, you can use `singleWindowApplication` instead of `application`. For multiple windows, you need to call the `Window` function for each window.

Open `Main.kt` in the desktop module. Before `fun main()`, add:

```
data class WindowInfo(val windowName: String, val windowState:  
WindowState)
```

Add any imports needed. The `WindowInfo` class just holds the window name and the window state.

Remove `val windowState = rememberWindowState()`, then add:

```
var initialized by remember { mutableStateOf(false) }
var windowCount by remember { mutableStateOf(1) }
val windowList = remember { SnapshotStateList<WindowInfo>() }
// Add initial window
if (!initialized) {
    windowList.add(WindowInfo("Timezone-${windowCount}",
        rememberWindowState()))
    initialized = true
}
```

Add any needed imports like `import androidx.compose.runtime.*`. The code above creates three variables:

1. A one-time `initialized` flag.
2. The number of windows open (starting at one).
3. The list of windows.

Then, it adds the first window entry (only once). This will be the first window to show up.

Replace the `Window` function with:

```
// 1
windowList.forEachIndexed { i, _ ->
    Window(
        onCloseRequest = {
            // 2
            windowList.removeAt(i)
        },
        state = windowList[i].windowState,
        // 3
        title = windowList[i].windowName
    ) {
```

Here's the explanation of the above code:

1. For each `WindowInfo` class in your list, create a new window.
2. When the window is closed, remove it from the list.
3. Set the title to the name from the `WindowInfo` class.

Then, add an ending } at the end of application. With the above code, you can now have multiple windows of your desktop application. You'll see this in action in the next section. Run the app again to make sure it's working fine.

Menus

If you look at the menu bar on macOS, you'll notice that your app doesn't have any menus as a regular app would:

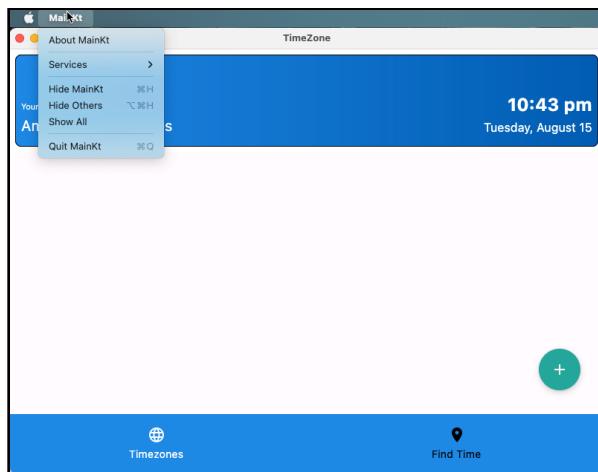


Fig. 5.13 – Desktop macOS menu

You'll now add a few menu items — like a File and Edit menu, as well as an exit menu option underneath the File menu to let the user exit the app.

Before the **Surface** function, add the code for a **MenuBar** as follows:

```
// 1
MenuBar {
    // 2
    Menu("File", mnemonic = 'F') {
        val nextWindowState = rememberWindowState()
        // 3
        Item(
            "New", onClick = {
                // 4
                windowCount++
                windowList.add(
                    WindowInfo(
                        "Timezone-${windowCount}",
                        nextWindowState
                    )
                )
            }
        )
    }
}
```

```
        }, shortcut = KeyShortcut(
            Key.N, ctrl = true
        )
    )
    Item("Open", onClick = { }, shortcut =
KeyShortcut(Key.O, ctrl = true))
    // 5
    Item("Close", onClick = {
        windowList.removeAt(i)
    },
        shortcut = KeyShortcut(Key.W, ctrl = true))
    Item("Save", onClick = { }, shortcut =
KeyShortcut(Key.S, ctrl = true))
    // 6
    Separator()
    // 7
    Item(
        "Exit",
        onClick = { windowList.clear() },
    )
}
Menu("Edit", mnemonic = 'E') {
    Item(
        "Cut", onClick = { }, shortcut = KeyShortcut(
            Key.X, ctrl = true
        )
    )
    Item(
        "Copy", onClick = { }, shortcut = KeyShortcut(
            Key.C, ctrl = true
        )
    )
    Item("Paste", onClick = { }, shortcut = KeyShortcut(Key.V,
ctrl = true))
}
```

You'll use the Compose `MenuBar` import as well as the Compose `Key` import. Here's what the above code does:

1. Create a `MenuBar` to hold all of your menus.
2. Create a new menu named **File**.
3. Create a menu item named **New** and also assign a keyboard shortcut by providing value to the `shortcut` argument.
4. Increment the window count and add a new `WindowInfo` class to the list. This will cause the function to execute again.
5. Close the current window by removing it from the list.

6. Add a separator.
7. Add the exit menu. This clears the window list, which will cause the app to close.

Most of these menus don't do anything. The File menu item will increment the window count, the close menu will remove the window from the window list and the exit menu will clear the list (causing the app to quit). Run the app. Here's what you'll see:

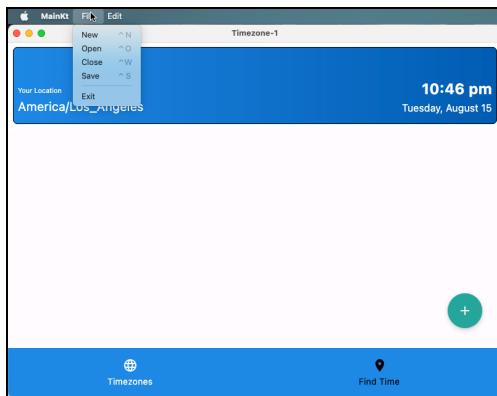


Fig. 5.14 – Desktop multiple windows

Try creating new windows and closing them. See what happens when you close the last window.

When writing your app, you may want to create your own menu file that handles menus. It could create a different menu system based on application state.

Distributing the App

When you're finally satisfied with your app, you'll want to distribute it to your users. The first step is to package it up into a distributable file. There isn't cross-compilation support available at the moment, so the formats can only be built using your current machine.

Note: At the time of writing, macOS distribution builds required Java 15 or greater. On M1 MacBook pros, you'll find that the Adoptium Arm-based JVM distribution works well and is easy to install. There are many third-party packages out there. Do a Google search to find one that's easy to install for your machine. In case you face issues while packaging the app, make sure to install a compatible JDK.

To create a **dmg** installer for the Mac, you need to run the **package** Gradle task. You can run it from Android Studio:

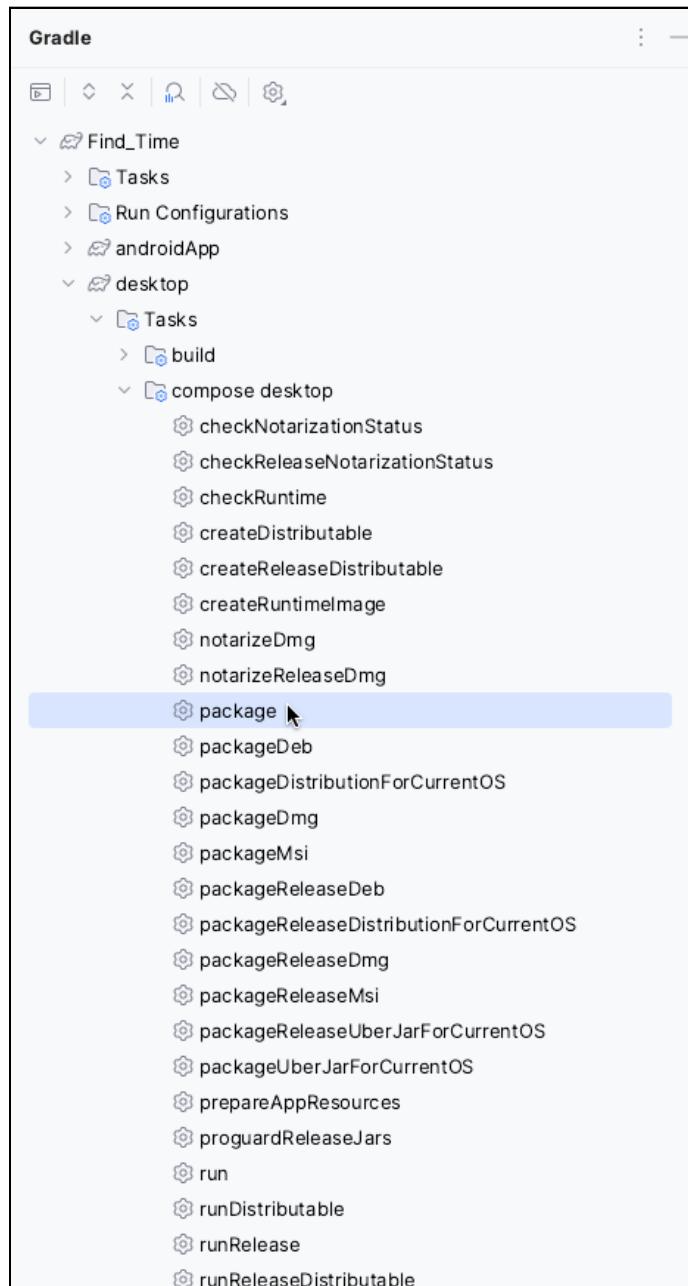


Fig. 5.15 – Gradle desktop package task

Or, run the following from the command line (in the project directory):

```
./gradlew :desktop:package
```

This will create the package in the **./desktop/build/compose/binaries/main/dmg** folder. Open it and you'll see your app. You can double-click the app to run it or drag it into your Applications folder.

Here's what your final app will look like:

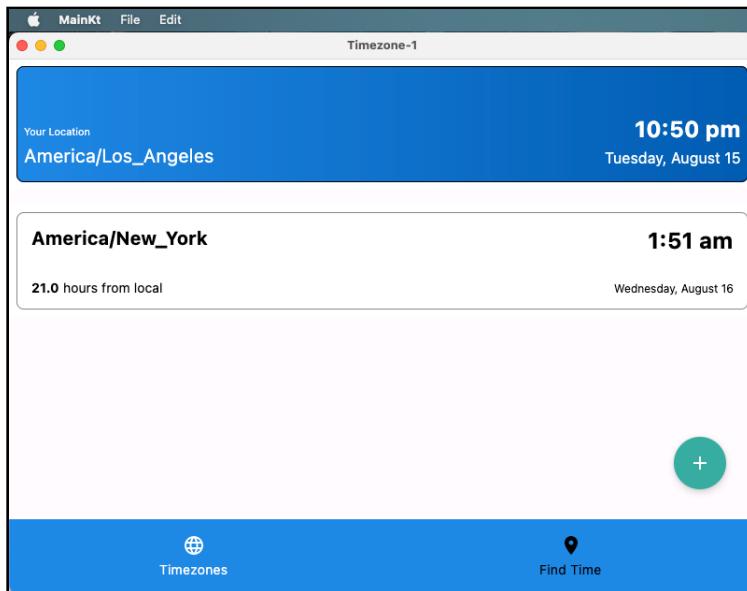


Fig. 5.16 – Desktop app running on macOS from packaged dmg

Running on Windows Platform

On Windows, the process is almost the same for running the app. To create a distributable package make sure you build the desktop package from Android Studio and then from the command line type:

```
.\gradlew.bat :desktop:package
```

Or, run the package task from the **compose desktop** task folder. Once this finishes, you'll find the **FindTime-1.0.0.msi** file in the **desktop/build/compose/binaries/main/msi** folder.

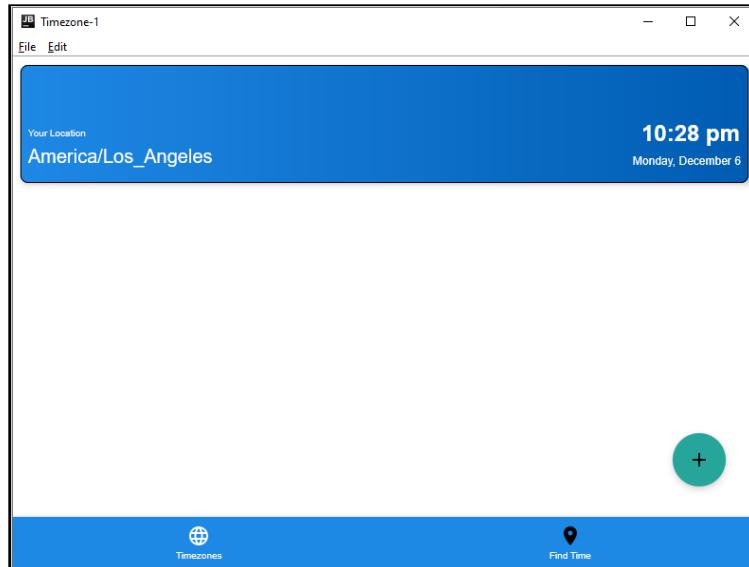


Fig. 5.17 – Desktop app running on Windows from packaged distribution

Congratulations! You were able to leverage your knowledge of Jetpack Compose to create your app on a whole new platform. If you have access to Windows, this will work there too.

Key Points

- **Compose Multiplatform** is a framework that uses most of the **Jetpack Compose** framework to display UIs.
- **Compose Multiplatform** works on Android, macOS, Windows and Linux desktops and the web.
- Desktop apps can have multiple windows.
- Desktop apps can use a menu system.
- Android can use **Compose Multiplatform** as well.

Where to Go From Here?

Compose for Desktop:

- Official site: <https://www.jetbrains.com/lp/compose-multiplatform/>
- Github Repository: <https://github.com/JetBrains/compose-multiplatform/#readme>
- Adoptium JVMs: <https://adoptium.net/>
- Material Surface: <https://m2.material.io/design/environment/surfaces.html#material-environment>

Congratulations! You've created a Compose Multiplatform app that uses a shared library for the business logic. Looks like you're on your way to mastering all the different platforms that KMP has to offer.

Section II: Kotlin Multiplatform: Intermediate

To effectively share code across apps, there are multiple things to keep in mind: access to platform-specific APIs, support for existing software engineering practices and persistence.

In this section, you'll learn how to use Kotlin features to access platform-specific APIs in your shared module and how Kotlin Multiplatform fits in with your current architecture. You'll also learn about dependency injection and how you can use it to test features present in your shared modules. Finally, you'll learn how to use a common codebase to handle persistence on different platforms.

6 Chapter 6: Connecting to Platform-Specific API

By Saeed Taheri

Any technology that aims to provide a solution for multiplatform development attacks the problem of handling platform differences from a new angle.

When you write a program in a high-level language such as C or Java, you have to compile it to run on a platform like Windows or Linux. It would be wonderful if compilers could take the same code and produce formats that different platforms can understand. However, this is easier said than done.

Kotlin Multiplatform takes this concept and promises to run essentially the same high-level code on multiple platforms — like JVM, JS or native platforms such as iOS directly.

Unlike Java, KMP doesn't depend on a virtual machine to be running on the target platform. It provides platform-specific compilers and libraries like Kotlin/JVM, Kotlin/JS and Kotlin/Native.

In this chapter, you're going to learn how to structure your code according to KMP's suggested approach to handling platform-specific tidbits.

Reusing Code Between Platforms

Kotlin Multiplatform doesn't compile the entire shared module for all platforms as a whole. Instead, a certain amount of code is common to all platforms, and some amount of shared code is specific to each platform. For this matter, it uses a mechanism called **expect/actual**.

In Chapter 1, you got acquainted with those two new keywords. Now, you're going to dive deeper into this concept.

Think of **expect** as a glorified **interface** in Kotlin or **protocol** in Swift. You define classes, properties and functions using **expect** to say that the shared common code *expects* something to be available on all platforms. Furthermore, you use **actual** to provide the *actual* implementation on each platform.

Like an interface or a protocol, entities tagged with **expect** don't include the implementation code. That's where the **actual** comes in.

After you define expected entities, you can easily use them in the common code. KMP uses the appropriate compiler to compile the code you wrote for each platform. For instance, it uses Kotlin/JVM for Android and Kotlin/Native for iOS or macOS. Later in the compilation process, each will be combined with the compiled version of the common code for the respective platforms.

You may ask why you need this in the first place. Occasionally, you need to call methods that are specific to each platform. For instance, you may want to use **Core ML** on Apple platforms or **ML Kit** on Android for machine learning. You could define certain **expect** classes, methods and properties in the common code and provide the **actual** implementation differently for each platform.

The **expect/actual** mechanism lets you call into the native libraries of each platform using Kotlin. How cool is that!

Say Hello to Organize

After you create a great app to find an appropriate time for setting up your international meetings, you'll need a way to make To-dos and reminders for those sessions. **Organize** will help you do exactly that.

As with many apps you use every day, **Organize** has a page that shows you the device information the app is running on. If you've ever faced a bug in your apps, you know how valuable this information can be when debugging.

As of writing this chapter, Android Studio and the KMM plugin do not use **Gradle Version Catalogs** by default. However, in order to adopt the most modern approach to managing dependencies with Android Studio, the Organize starter project employs them. It also includes additional platforms and pre-configured dependencies.

Furthermore, similar to the project you created in Section 1, you will be using the **Regular framework** instead of **CocoaPods** for iOS framework distribution. While **Swift Package Manager(SPM)** has become the standard method for managing dependencies when developing for Apple platforms, the KMP team has not yet made SPM an available option.

If you ever decide to create a new project yourself, you can change the iOS framework distribution option on the following page.

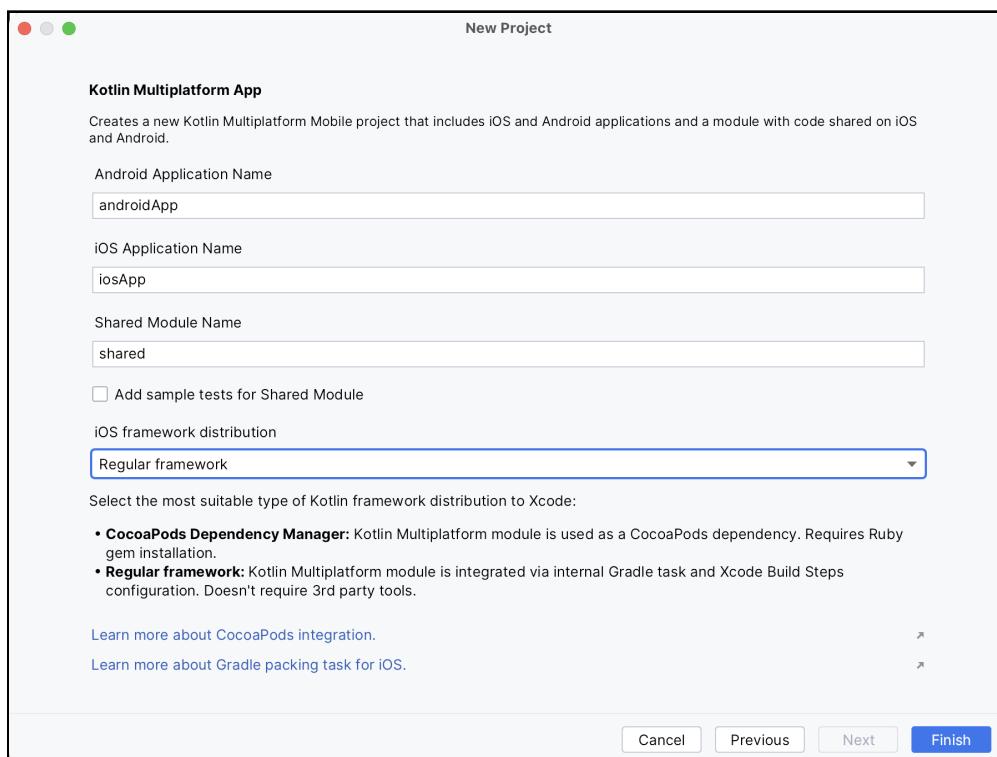


Fig. 6.1 – Select Regular framework option for iOS framework distribution

Updating the Platform Class

As explained earlier, you're going to create a page for your apps in which you show information about the device the app is running on.

For that matter, most of your work in this chapter relates to the `Platform` class.

Folder Structure

In Android Studio, choose the **Project** view in Project Navigator. Inside the **shared** module, browse through the directory structure.

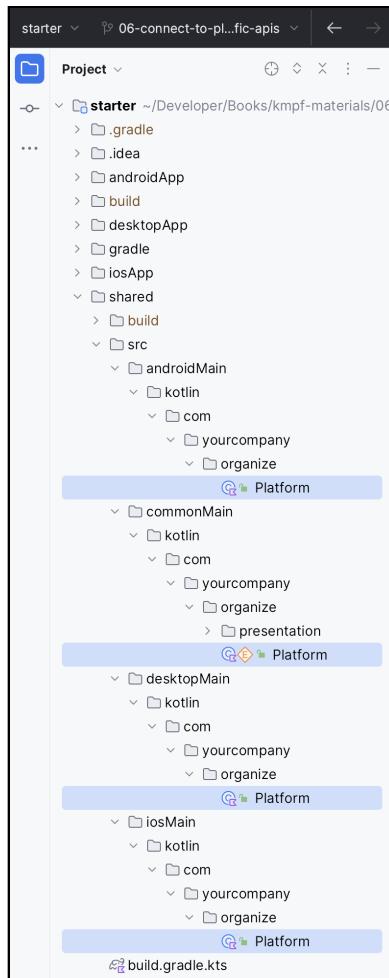


Fig. 6.2 – Folder structure in Android Studio

There's already a file named **Platform.kt** inside the project, or to be more exact, there are *four* of them — one for each platform you support, plus one. To make the expect/actual mechanism work, you'll need to define the `expect` and `actual` entities in exactly the same package in each platform.

In the image above, the `expect` class for **Platform** is inside the `com.yourcompany.organize` package under the `commonMain` directory. The actual implementations for iOS, Android and desktop are inside the same package under `iosMain`, `androidMain` and `desktopMain`, respectively.

Creating the Platform Class for the Common Module

Open **Platform.kt** inside the `commonMain` folder. Replace the `expect` class definition with the following:

```
expect class Platform() {
    val osName: String
    val osVersion: String

    val deviceModel: String
    val cpuType: String

    val screen: ScreenInfo

    fun logSystemInfo()
}

expect class ScreenInfo() {
    val width: Int
    val height: Int
    val density: Int?
}
```

By writing this, you're making a promise to KMP that you're going to provide this information. As you see, there's no implementation for anything here. You define what you want, just like an `interface` or `protocol`.

Note: You've used Kotlin's shorthand notation for constructor definition by using `Platform()`. This means KMP now expects you to provide an implementation for the constructor alongside the properties and methods.

You may be surprised that you didn't define `Platform` or `ScreenInfo` as a data class; after all, these classes seem a perfect fit for a data class, since they're essentially data holders.

The reason is that data classes in Kotlin automatically generate some implementations under the hood. Consequently, you can't use them here, as `expect` classes shouldn't have implementations.

You also can't define nested classes inside an `expect` class. Hence, you defined the `ScreenInfo` class outside the `Platform` definition. You can also create a new file if you desire. Doing it in the same file would work, too.

In the code gutter, click the yellow rhombus with the letter A in it. This lets you navigate to the actual implementation file for the platforms you defined in the project.

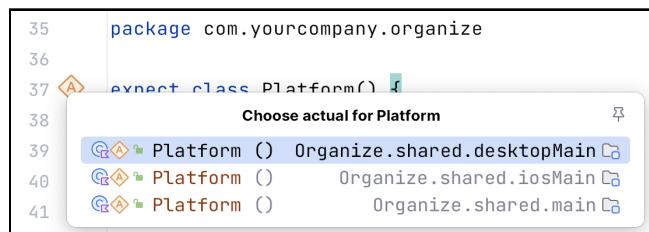


Fig. 6.3 – Navigate to actual implementation files

If the files aren't already in their respective places, or you haven't implemented the **actual** definition yet, you can put the cursor on the `expect` class name and press **Alt+Enter** on the keyboard. Android Studio will help ease the process. This is the case for `ScreenInfo`, for instance:

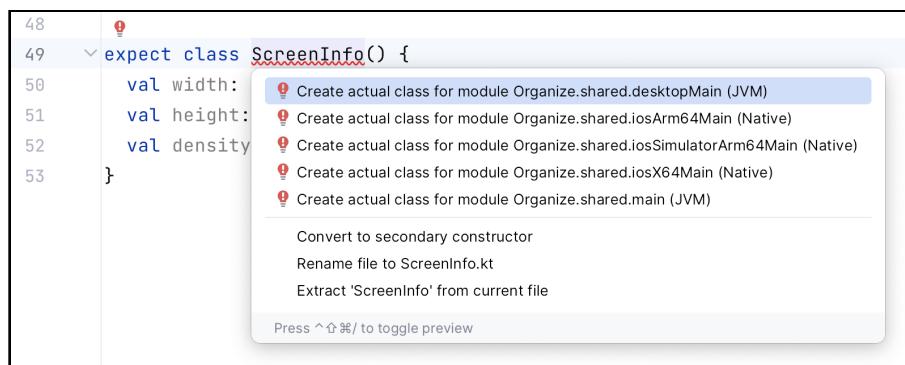


Fig. 6.4 – Alt+Enter on expect class name to create actual classes

Implementing Platform on Android

Go to the **Platform.kt** inside the **androidMain** folder.

You'll see that Android Studio has already started nagging you to fulfill the promise. After all, KMP is in its infancy, and you know how toddlers are!



Fig. 6.5 – Android Studio errors in actual class

Replace the entire class definition with this block of code:

```
//1
actual class Platform actual constructor() {
//2
    actual val osName = "Android"

//3
@androidx.annotation.ChecksSdkIntAtLeast(extension = 0)
actual val osVersion = "${Build.VERSION.SDK_INT}"

//4
actual val deviceModel = "${Build.MANUFACTURER} ${Build.MODEL}"

//5
actual val cpuType = Build.SUPPORTED_ABIS.firstOrNull() ?: "___"

//6
actual val screen = ScreenInfo()

//7
actual fun logSystemInfo() {
    Log.d(
        "Platform",
```

```
    "($osName; $osVersion; $deviceModel; ${screen.width}x$  
     ${screen.height}@${screen.density}x; $cpuType)"  
  )  
}  
  
// 8  
actual class ScreenInfo actual constructor() {  
  //9  
  private val metrics = Resources.getSystem().displayMetrics  
  
  //10  
  actual val width = metrics.widthPixels  
  actual val height = metrics.heightPixels  
  actual val density: Int? = round(metrics.density).toInt()  
}
```

This seems like a lot of code, but it's pretty straightforward:

1. You provide the actual implementation for the `Platform` as well as its default constructor. Here, you can't use the shorthand notation as you did in the `expect` file. You need to explicitly put an `actual` keyword before the `constructor`.
2. For the operating system name, you provided the value "Android" because you know this code will be compiled for the Android part of the shared module.
3. For the operating system version, you used the SDK version from the `Build` class in Android. Make sure to let the Android Studio import the needed package: `android.os.Build`. Since you're inside the Android part of the shared module, you can freely use any Android-specific API. Since you're checking the SDK version in this property, you must annotate the property with `@androidx.annotation.ChecksSdkIntAtLeast(extension = 0)`.
4. For the device model, you used static properties of `MANUFACTURER` and `MODEL` from `Build`.
5. Thankfully, `Build` can give you the CPU type of the device using the `SUPPORTED_ABIS` property. Since the result may be null on some older versions of Android, provide a default value as well.
6. You initialize an instance of `ScreenInfo` and store it in `screen` property.

7. Like in an interface, you provide function implementation here. For now, you'll use the `Log` class in Android to output all the properties to the console. Make sure to import `android.util.Log`. You can safely unwrap the nullable `screen` property since you initialized it with a non-null value in the previous part.
8. You provide the actual implementation for the `ScreenInfo` as well as its default constructor.
9. For fetching the screen properties, you'll need a `DisplayMetrics` object. You can get that using this block of code. As you see, you can have extra properties or functions inside the actual class. Make sure to import `android.content.res.Resources`.
10. You get the screen width, height and density using the `metrics` property you defined earlier. Import `kotlin.math.round` to be able to use the `round` function. For the `density` property, you'll need to explicitly write the type, since if you don't, its type would be non-nullable. Not only that, but you promised this property to be nullable in the expect file, and one should always stick to their promises. The reason this is of a nullable type will be clear when you implement the desktop part.

Next, you'll implement the iOS-specific code.

Implementing Platform on iOS

When you're inside an actual file, you can click the yellow rhombus with the letter **E** in the gutter to go to the expect definition. While inside `Platform.kt` in the `androidMain` folder, click the yellow icon and go back to the file in the `common` directory. From there, click the **A** icon and go to the iOS `actual` file.

The basics of the code you're going to add are the same as before. This time, though, you're calling into iOS-specific frameworks such as `UIKit`, `Foundation` and `CoreGraphics` to fetch the needed information.

Replace the actual implementation with the following block of code.

```
actual class Platform actual constructor() {  
    //1  
    actual val osName = when  
        (UIDevice.currentDevice.userInterfaceIdiom) {  
            UIUserInterfaceIdiomPhone -> "iOS"  
            UIUserInterfaceIdiomPad -> "iPadOS"  
            else -> kotlin.native.Platform.osFamily.name  
    }  
}
```

```
//2
actual val osVersion = UIDevice.currentDevice.systemVersion

//3
actual val deviceModel: String
    get() {
        memScoped {
            val systemInfo: utsname = alloc()
            uname(systemInfo.ptr)
            return NSString.stringWithCString(systemInfo.machine,
encoding = NSUTF8StringEncoding)
                ?: "___"
        }
    }

//4
actual val cpuType =
kotlin.native.Platform.cpuArchitecture.name

//5
actual val screen = ScreenInfo()

//6
actual fun logSystemInfo() {
    NSLog(
        "($osName; $osVersion; $deviceModel; ${screen.width}x${screen.height}@${screen.density}x; $cpuType)"
    )
}

actual class ScreenInfo actual constructor() {
    //7
    actual val width =
CGRectGetWidth(UIScreen.mainScreen.nativeBounds).toInt()
    actual val height =
CGRectGetHeight(UIScreen.mainScreen.nativeBounds).toInt()
    actual val density: Int? = UIScreen.mainScreen.scale.toInt()
}
```

1. There's a class in UIKit called `UIDevice` from which you can query information about the `currentDevice`. In this code, you're asking for the interface idiom to differentiate between iOS and iPadOS. The `UIUserInterfaceIdiom` enum has a few more cases. For brevity, you used the `Kotlin/Native Platform` class to find information in the `else` block.
2. You can also use `UIDevice` to get the OS version.

3. This is by far the clunkiest piece of code you'll encounter in this book. But don't worry: KMP isn't usually like this. It's here to demonstrate where things can become intricate. Objective-C at its core is C. In C, Structures (also called structs) are a way to group several related variables into one place. Practically, whenever you want to use a C struct, you'll need to utilize the `cinterop` (C Interoperability) package of Kotlin. Using that package has its perks and occasionally, it gets a bit challenging. Here, you're going to employ a C struct called `utsname`. Unix fans, rejoice! In this block of code, you're allocating memory using the `memScoped` block and the `alloc()` function call. Then, you pass a pointer to the allocated memory space to the `uname` function, which retrieves the operating system information and fills it inside `systemInfo`. Subsequently, you convert the C String filled with the machine name to `NSString` and return it. The cast from `NSString` to Kotlin String is automatic. Phew!
4. To obtain the CPU type, you can once again dig into C code, or like here, simply use the `Kotlin/Native Platform` class.
5. You initialize an instance of `ScreenInfo` and store it in `screen` property.
6. The function implementation is essentially the same as the Android implementation, except that you're passing the same string to `NSLog` function.
7. You need to combine your knowledge of `UIKit` and `CoreGraphics` to get the screen properties. First, you utilize the `UIScreen` class to fetch information about the `mainScreen` of the device. Then you use `CGRectGetWidth` and `CGRectGetHeight` functions of `CoreGraphics` to extract the width and height from the `nativeBounds` property, which is a `CGSize` Objective-C, or at its core, a C struct. Just like you did on Android, make sure to explicitly specify the type of density. Go ahead and import all missing packages if you haven't done so already.

Note: Whenever you use the `cinterop` package, or call into the `Kotlin/Native` package, you must opt-in, as those APIs are experimental. If you check out the `build.gradle.kts` file inside the `shared` module, at the end of the file, you'll see that these lines are already there for you:

```
tasks.withType<org.jetbrains.kotlin.gradle.tasks.KotlinNativeCom  
pile>().configureEach {  
    compilerOptions.freeCompilerArgs.addAll(  
        "-opt-in=kotlinx.cinterop.ExperimentalForeignApi",
```

```
        "-opt-in=kotlin.experimental.ExperimentalNativeApi"
    }
}
```

Also, you need to annotate the classes or functions in which you've used such API. Make sure to add these lines before each class definition:

```
@kotlinx.cinterop.ExperimentalForeignApi
@kotlin.experimental.ExperimentalNativeApi
actual class Platform actual constructor() {
    //...
}

@kotlinx.cinterop.ExperimentalForeignApi
actual class ScreenInfo actual constructor() {
    //...
}
```

These may look a bit weird for Kotlin and Swift developers. The reason is that you're using the Objective-C nomenclature for entities. This is how KMP works for Apple platforms. The interoperability is thereby creating a bridge between Kotlin and Objective-C.

The block of code in Section 3 is odd, even for Swift developers. If you were to write this section using Swift, there would also be some travels to the C world:

```
let deviceModel: String = {
    var systemInfo = utsname()
    uname(&systemInfo)
    let str = withUnsafePointer(to: &systemInfo.machine.0) { ptr
        in
            return String(cString: ptr)
    }
    return str
}()
```

Why Objective-C and not Swift, you may ask. Although Swift interoperability is in the works by KMP creators, they chose to go with Objective-C for a couple of reasons:

1. Many of the iOS frameworks themselves are built with Objective-C. Even when you write Swift code, you're using a bridge.

- Objective-C has a more flexible and dynamic runtime than Swift. Apparently, Swift's stricter type-safety features would have made the creation of KMP interoperability with Apple technologies more difficult.

Using Objective-C instead of Swift has some issues, though. For instance, you can't use some newly introduced frameworks such as AppIntents as they're Swift-only. Furthermore, you can't use Swift-only extension functions or properties — or even Swift enum cases — either. You have no choice other than to use the verbose naming of entities in Objective-C.

Implementing Platform on Desktop

Open **Platform.kt** inside the **desktopMain** folder.

Replace the actual implementation with this block:

```
actual class Platform actual constructor() {  
    //1  
    actual val osName = System.getProperty("os.name") ?: "Desktop"  
  
    //2  
    actual val osVersion = System.getProperty("os.version") ?:  
        "___"  
  
    //3  
    actual val deviceModel = "Desktop"  
  
    //4  
    actual val cpuType = System.getProperty("os.arch") ?: "___"  
  
    //5  
    actual val screen = ScreenInfo()  
  
    //6  
    actual fun logSystemInfo() {  
        print("$osName; $osVersion; $deviceModel; ${screen.width}x${screen.height}; $cpuType")  
    }  
}  
  
actual class ScreenInfo actual constructor() {  
    //7  
    private val toolkit = Toolkit.getDefaultToolkit()  
  
    actual val width = toolkit.screenSize.width  
    actual val height = toolkit.screenSize.height  
    actual val density: Int? = null  
}
```

1. The desktop app is based on JVM. As a result, you can use JDK classes and methods to get information about the device. There's a class in Java called `System`. You can get the operating system name by using the static `getProperty` method with the "`os.name`" parameter. Since this method may return null, you provided a default result.
2. You use the same method as before, but this time with the "`os.version`" parameter.
3. You hard-code the value "`Desktop`". JVM doesn't provide a way to know anything about the manufacturer and model.
4. Once again, `System` class to the rescue! Use "`os.arch`" as the parameter.
5. You create an instance of `ScreenInfo`, as you did on the other platforms.
6. Next, you use Kotlin's `print` function to output the usual info to the console.
7. You create an instance of `Toolkit` and query the screen size methods on that object. Unfortunately, this property doesn't give us the screen density.

Java doesn't have a UI toolkit in itself. If a platform owner wants to use JVM and provide developers with a way to develop user interfaces, it creates a UI toolkit or uses one already available.

You may have heard about Swing or Abstract Window Toolkit (AWT). Jetpack Compose for Desktop uses Swing internally to make window-based desktop applications. As of writing this book, Jetpack Compose for Desktop doesn't provide a way to query screen information outside its composable methods. However, you can use AWT methods outside the Compose world. The `Toolkit` class you used, is inside the `java.awt` package.

Sharing More Code

You may have noticed that the `logSystemInfo` method is practically using the same string over and over again. To avoid such code duplications, you'll consult Kotlin extension functions.

Open `Platform.kt` inside the `commonMain` folder. As you know, you can't add implementation to the properties or functions you defined here. However, no one said you can't use Kotlin extension functions.

At the end of the file, add this:

```
val Platform.deviceInfo: String
    get() {
        var result = "($osName; $osVersion; $deviceModel; ${screen.width}x${screen.height}")

        screen.density?.let {
            result += "@${it}x; "
        }

        result += "$cpuType"
        return result
    }
```

You're making the same string, this time, based on the fact that density may be null.

Now go back to the actual files and use this property inside `logSystemInfo` functions.

In `Platform.kt` inside `androidMain`:

```
actual fun logSystemInfo() {
    Log.d("Platform", deviceInfo)
}
```

In `Platform.kt` inside `iosMain`:

```
actual fun logSystemInfo() {
    NSLog(deviceInfo)
}
```

In `Platform.kt` inside `desktopMain`:

```
actual fun logSystemInfo() {
    print(deviceInfo)
}
```

With this technique, you're able to share code between the actual implementations.

Updating the UI

Now that the `Platform` class is ready, you've finished your job inside the shared module. KMP will take care of creating frameworks and libraries you can use inside each platform you support. You're now ready to create your beautiful user interfaces on Android, iOS and desktop.

Android

You'll do all of your tasks inside the **androidApp** module. The basic structure of the app is ready for you. Some important files need explaining. These will help you in the coming chapters as well. Here's what it looks like:

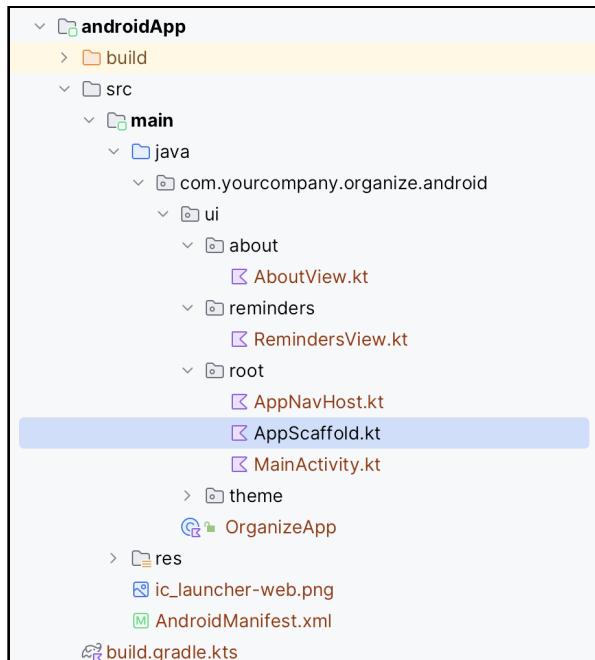


Fig. 6.6 – Folder structure for Android app

Inside the **root** folder, there are **AppScaffold.kt** and **AppNavHost.kt**. These two files set up the screens of the app and make the navigation between them work as intended. Please don't hesitate to take a look if you're interested.

The app has two main screens: **RemindersView**, which shows a simple “Hello World” for now, and the **AboutView**, which you're going to set up in this chapter. Go ahead and open it.

Here, the **ContentView** is where everything essential happens. Replace its implementation with the following snippet:

```

    @Composable
    private fun ContentView() {
        val items = makeItems()

        LazyColumn(
    
```

```
        modifier = Modifier.fillMaxSize(),
    ) {
    items(items) { row ->
        RowView(title = row.first, subtitle = row.second)
    }
}
```

Here, you get the items you'd like to show out of the function `makeItems` and put them inside a `LazyColumn`, which is basically a list view. Import `Modifier`, `LazyColumn` and `fillMaxSize` from the **Compose** library. Add the following import for the `items` method:

```
import androidx.compose.foundation.lazy.items
```

You'll implement `RowView` soon.

Add the `makeItems` method below `ContentView`:

```
private fun makeItems(): List<Pair<String, String>> {
    //1
    val platform = Platform()

    //2
    val items = mutableListOf(
        Pair("Operating System", "${platform.osName} ${platform.osVersion}"),
        Pair("Device", platform.deviceModel),
        Pair("CPU", platform.cpuType)
    )

    //3
    val max = max(platform.screen.width, platform.screen.height)
    val min = min(platform.screen.width, platform.screen.height)

    var displayInfo = "${max}x${min}"
    platform.screen.density?.let {
        displayInfo += " ${it}x"
    }

    items.add(Pair("Display", displayInfo))

    return items
}
```

1. First, you initialize an instance of the `Platform` class you created earlier. Import it.
2. Next, you create pairs of data with titles and info from the `platform` and store them in a mutable list.
3. You'll create a textual representation for the screen. Although you know that `density` property isn't null on Android, it's better to be safe than sorry when facing nullable properties.

Import `max` and `min` from `kotlin.math`. And for the final piece of this app, add the `RowView` composable function as follows:

```
@Composable
private fun RowView(
    title: String,
    subtitle: String,
) {
    Column(modifier = Modifier.fillMaxWidth()) {
        Column(Modifier.padding(8.dp)) {
            Text(
                text = title,
                style = MaterialTheme.typography.bodySmall,
                color = Color.Gray,
            )
            Text(
                text = subtitle,
                style = MaterialTheme.typography.bodyLarge,
            )
        }
        Divider()
    }
}
```

This is a simple vertical stack of text items that shows a title and subtitle. You can use predefined material typography values to polish things up. These are similar to the predefined text styles in the iOS Dynamic Type feature. Import the needed classes and methods from the Compose library.

That's the end of your journey on Android in this chapter. Build and run the app, and take a look at the result.

Tap the **i** button to take a look at the device properties.

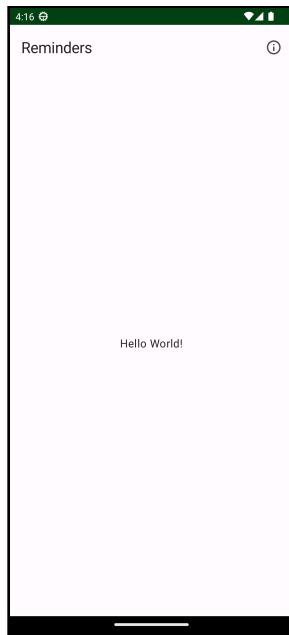


Fig. 6.7 – The first page of Organize on Android

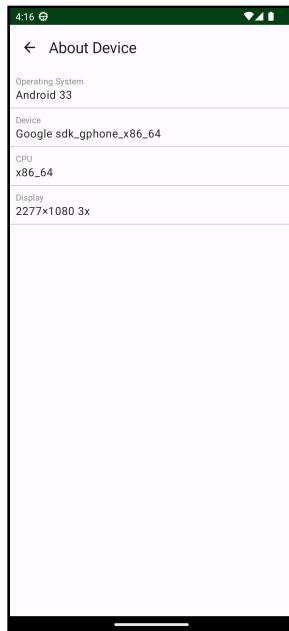


Fig. 6.8 – The About Device page of Organize on Android

Next, you're going to build the iOS app.

iOS

Although no one can stop you from using Android Studio for editing Swift files, it would be smarter to open Xcode.

Inside the **iosApp** folder in the project's root directory, open the Xcode project by double-clicking **iosApp.xcodeproj**.

The **ContentView.swift** file is the starting page of the application. It's already there for you. Take a look if you'd like.

Open **AboutView.swift** and replace the line where it has `Text("Hello World")` with this:

```
AboutListView()
```

Next, create a new SwiftUI view file called **AboutListView** by pressing **Command-N**.

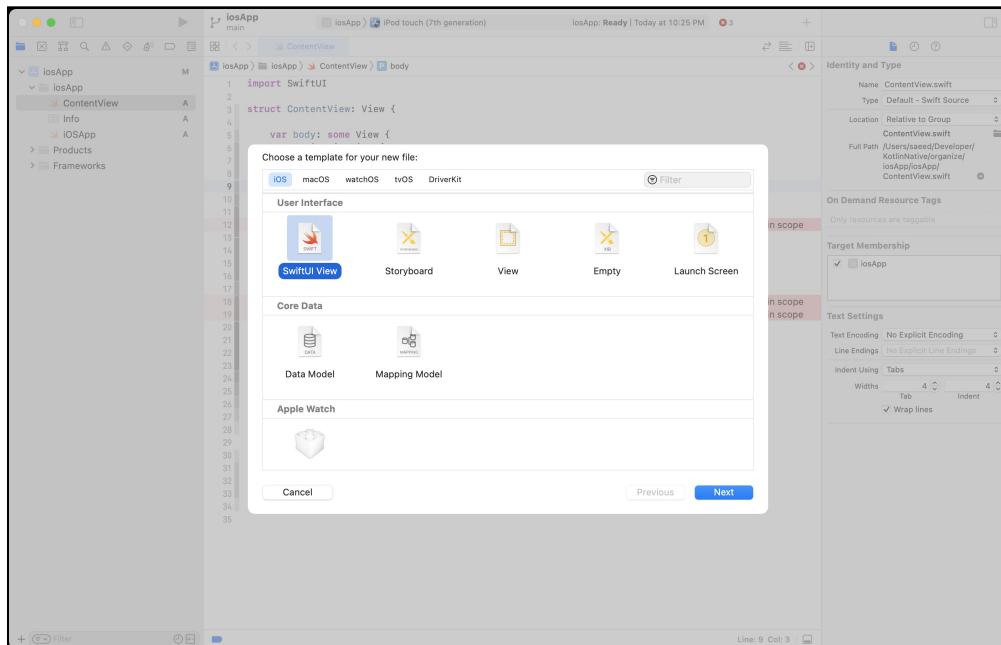


Fig. 6.9 – Xcode new file dialog

First, import the Shared module at the top of the file:

```
import Shared
```

This is the framework KMP created for you. If it gives you an error stating that it's not found, don't worry. Building the project will resolve the issue.

Second, add an inner struct in `AboutListView` to hold the data you're going to show:

```
private struct RowItem: Hashable {
    let title: String
    let subtitle: String
}
```

The conformance to the `Hashable` protocol is a necessity for the `ForEach` structure in SwiftUI.

Third, inside the `AboutListView` struct, add a property to hold a reference to the items you're going to show from the `Platform` class.

```
private let items: [RowItem] = {
    //1
    let platform = Platform()

    //2
    var result: [RowItem] = [
        .init(
            title: "Operating System",
            subtitle: "\u{2028}(platform.osName) \u{2028}(platform.osVersion)"
        ),
        .init(
            title: "Device",
            subtitle: platform.deviceModel
        ),
        .init(
            title: "CPU",
            subtitle: platform.cpuType
        )
    ]

    //3
    let width = min(platform.screen.width, platform.screen.height)
    let height = max(platform.screen.width,
                     platform.screen.height)

    var displayValue = "\u{2028}(width)\u{2028}\u{2028}(height)"

    if let density = platform.screen.density {
        displayValue += " @\u{2028}(density)\u{2028}x"
    }
}
```

```
result.append(
    .init(
        title: "Display",
        subtitle: displayValue
    )
)

//4
return result
}()
```

1. You create an instance of the `Platform` class.
2. Next, you create an array of `RowItem` instances, containing the info from the `platform` instance.
3. Then, you calculate the `width` and `height` and conditionally unwrap the `density` property and append the result to the array.
4. At the end, you return the array you'd like to show on the page.

Finally, replace the content of the `body` property with this:

```
var body: some View {
    List {
        ForEach(items, id: \.self) { item in
            VStack(alignment: .leading) {
                Text(item.title)
                    .font(.footnote)
                    .foregroundStyle(.secondary)
                Text(item.subtitle)
                    .font(.body)
                    .foregroundStyle(.primary)
            }
            .padding(.vertical, 4)
        }
    }
}
```

This is a very basic list in SwiftUI. For each item inside the `items` property, you show a vertical stack of text elements consisting of the title and the subtitle. You also apply a bunch of formatting modifiers such as `font` and `foregroundStyle` to make it appear more pleasing to the eye.

Build and run. Then, tap the **About** button to see the page you created.



Fig. 6.10 – The first page of Organize on iOS

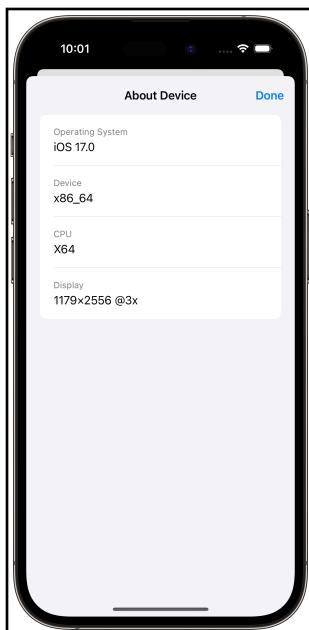


Fig. 6.11 – The About Device page of Organize on iOS

Desktop

In Section 1, you learned how to share your UI code between Android and desktop. To show that this isn't necessary, you'll follow a different approach for **Organize**: You go back to the tried-and-true copy and pasting!

The setup for the desktop app is a bit different from the Android app, though they both use Jetpack Compose. One difference is that you don't use the Jetpack Navigation Component on the desktop app. You also open the **About Device** page in a new window to be more in line with desktop conventions.

Except for a few nuances in the design for the About page, like showing each data item in a **Row** instead of a **Column**, the code is the same. It's there for you in the starter project. Back in Android Studio, Open `AboutView.kt` from the **desktopApp** module, locate `//2` and `//3` and uncomment the code below them.

There are multiple ways to run the desktop app. You can open the **Gradle** menu on the side under **desktopApp** ▶ **compose desktop** and click **run**.

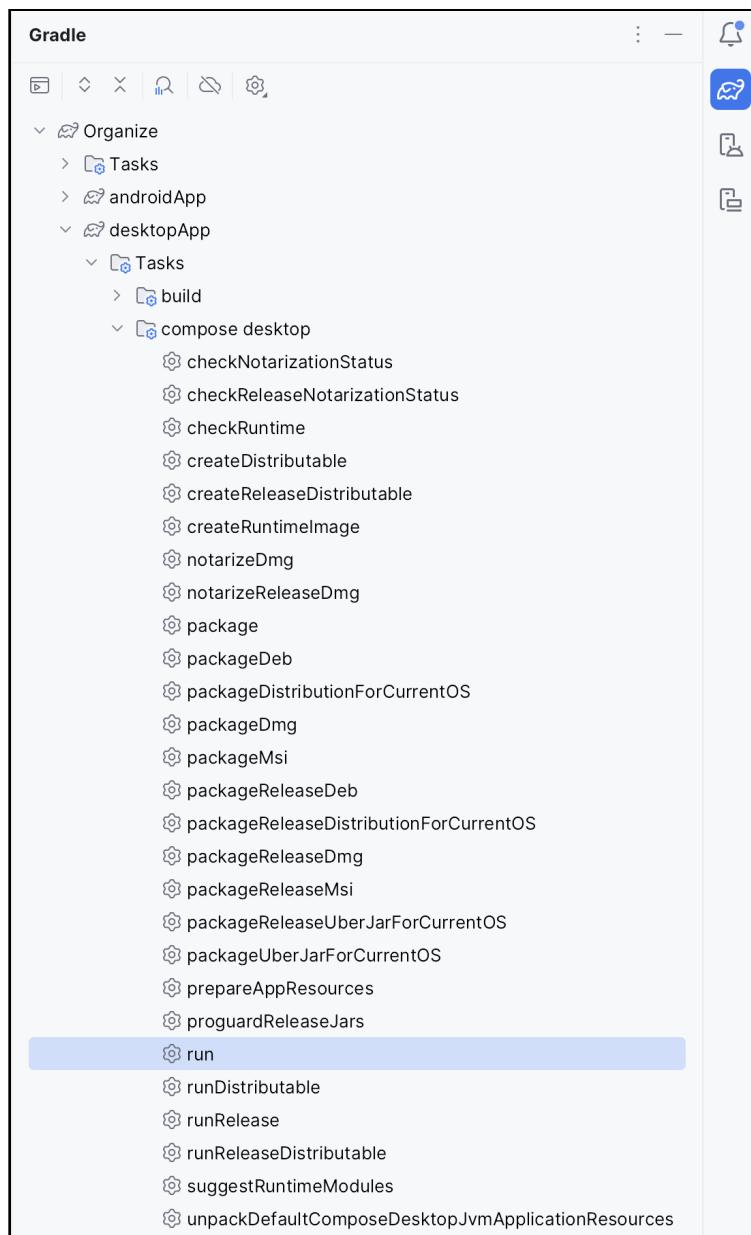


Fig. 6.12 – Gradle menu, run desktop app

The app runs, and it looks mostly like the Android app.

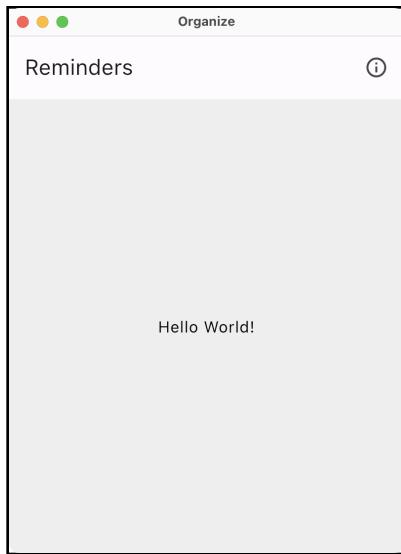


Fig. 6.13 – The first page of Organize on Desktop

About Device	
Operating System	Mac OS X 13.5.1
Device	Desktop
CPU	x86_64
Display	2560x1440

Fig. 6.14 – The About Device page of Organize on Desktop

Challenge

Here's a challenge for you to practice what you learned. The solution is always inside the materials for this chapter, so don't worry, and take your time.

Challenge: Create a Common Logger

You can call other expect functions inside your expect/actual implementations. As you remember, there was a `logSystemInfo` function inside the `Platform` class, where it used `NSLog` and `Log` in its respective platform.

Refactor these calls into a new class called `Logger`. As a bonus, you can add log levels to your implementation.

Key Points

- You can use the expect/actual mechanism to call into native libraries of each platform using Kotlin.
- Expect entities behave so much like an interface or protocol.
- On Apple platforms, Kotlin uses Objective-C for interoperability.
- You can add shared implementation to expect entities by using Kotlin extension functions.

Where to Go From Here?

- Interoperability with C: <https://kotlinlang.org/docs/native-c-interop.html>
- The expect/actual pattern: <https://kotlinlang.org/docs/multiplatform-connect-to-apis.html>

Congratulations! You've written multiplatform implementations for a class and then used them in native apps on three platforms.

Chapter 7: App Architecture

By Saeed Taheri

In the previous chapter, you started creating the **Organize** app. However, you didn't make it to the *organization* part. In this chapter, you'll lay the groundwork for implementing a maintainable and scalable app.

Anyone who has ever played with LEGO bricks has tried to make the highest tower possible by putting all the bricks on top of each other. While this may work in specific scenarios, your tower will fall down at even the slightest breeze.

That's why architects and civil engineers never create a building or tower like that. They plan extensively, so their creations stay stable for decades. The same applies to the software world.

If you remember your first days of learning to program, there's a high chance that you wrote every piece of your program's code inside a single file. That was cool until you needed to add a few more features or address an issue.

Although the term **software architecture** is relatively new in the industry, software engineers have applied the fundamental principles since the mid-1980s.

Design Patterns

The broad heading of software architecture consists of numerous subtopics. One of these is **architectural styles** — otherwise known as **software design patterns**. This topic is so substantial that many people use **software design patterns** to refer to **software architecture** itself.

Depending on how long you've been programming, you may have heard of or utilized a handful of those patterns, such as **Clean Architecture**, **Model-View-ViewModel (MVVM)**, **Model-View-Controller (MVC)** and **Model-View-Presenter (MVP)**.

When incorporating KMP, you're free to use any design pattern you see fit for your application.

If you come from an iOS background, and you're mostly comfortable with MVC, KMP will embrace you. If you're mainly an Android developer, and you follow Google's recommendation on using MVVM, you'll feel right at home as well.

There is no best or worst way to do it.

Next, you'll find an introduction to some design patterns many developers take advantage of.

Model-View-Controller

The MVC pattern's history goes back to the 1970s. Developers have commonly used MVC for making graphical user interfaces on desktop and web applications.

In the mobile world, Apple made MVC mainstream when it introduced the iPhone SDK in 2008. If you did iOS development before SwiftUI, you may have noticed that one of the base components was a `UIViewController`. It speaks for itself how Apple heavily invested in this pattern.

For long years before Google became opinionated about Android development patterns and architectures, developers used Model-View-Presenter, or **MVP**, which is a close deviation of MVC.

In MVC, you partition your code into three separate camps:

- **Model:** The central component of the pattern. It's completely independent of the UI and handles the logic and rules of the application.
- **View:** Any representation of information, such as lists, grids, etc. This section is usually platform- and framework-dependent. You can use UIKit and SwiftUI on iOS and Views or Jetpack Compose on Android.
- **Controller:** Accepts input and converts it to commands for model or view. It also receives feedback from the model and reflects the changes to the view. It's somehow the know-it-all of the pattern.

The diagram below shows the relationship between the different partitions:

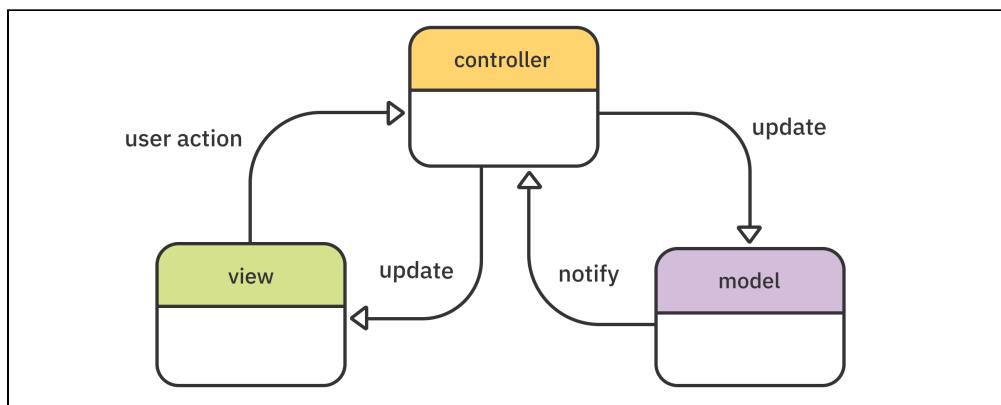


Fig. 7.1 – MVC diagram

Model-View-ViewModel

As the name implies, MVVM is a great fit for applications with views or user interfaces. Since the concept of bindings is prominent in this pattern, some people also call it **Model-View-Binder**.

It's much newer than MVC. John Gossman, one of Microsoft's engineers, announced MVVM in his blog in 2005. Microsoft embraced MVVM in .NET frameworks and made this pattern very popular.

Google introduced the Architecture Components at Google I/O 2017. This marked the first time Google recommended a design pattern for developing Android applications. Over the years, Google has also introduced various tools and components centered around the concept of MVVM. Today, MVVM is the preferred choice for most Android developers when creating an app.

The components of MVVM are as follows:

- **Model:** It's much like the model layer of MVC. It represents the app data and rules.
- **View:** Pretty much similar to the component with the same name in MVC. It represents the model, receives input from the user and forwards the handling of the input to the ViewModel via a link between View and ViewModel.
- **ViewModel:** ViewModel is basically the state of data in the model. It exposes some public methods and properties to which the View subscribes and receives the changes automatically. People call this mechanism **Data Binding**, or simply **Binding**.

Android developers are no strangers to using **LiveData**, or recently, **Kotlin Flow** or **StateFlow**, as the **Binder** inside ViewModel.

After Apple introduced the **Combine** framework and SwiftUI as a first-party solution to reactive programming and declarative UI, iOS developers began adopting the MVVM design pattern and the binding mechanism more and more. Additionally, Apple introduced the **Observation** framework in iOS 17 to simplify the utilization of this pattern.

Clean Architecture

In 2012, Robert C. Martin, also known as Uncle Bob, published a post in his blog at <https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html> explaining the details of a new design pattern he came up with based on Hexagonal Architecture, Onion Architecture and many more.

Clean Architecture has a steeper learning curve because it has more components. However, because of its extensibility and its ability to handle various problems in software development, it's very popular among professionals — especially when they want to create large applications.

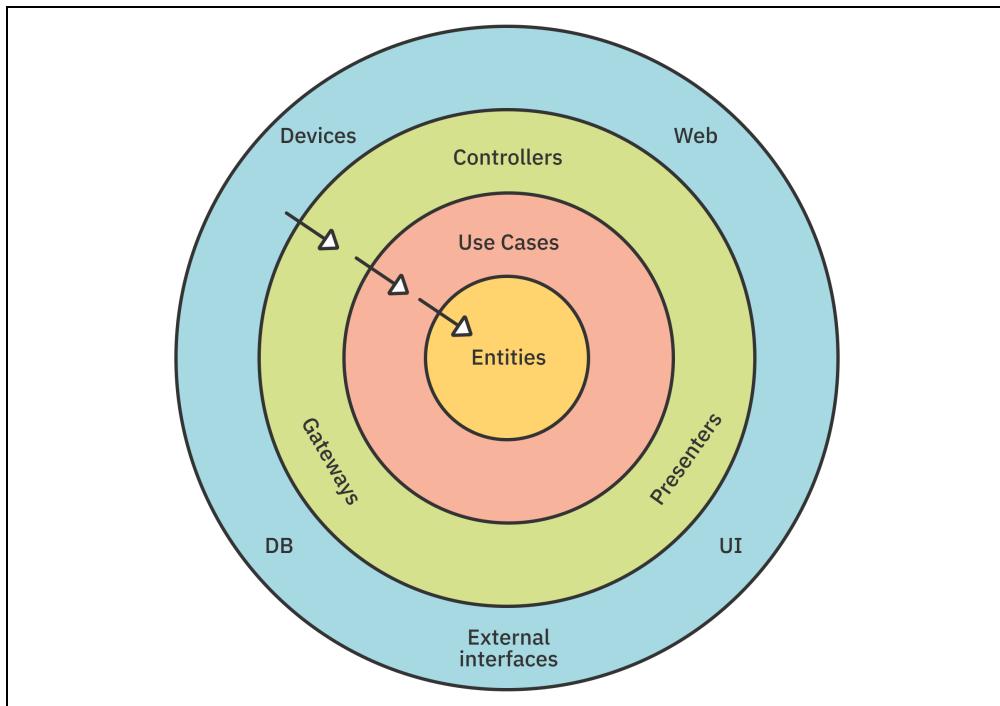


Fig. 7.2 — Clean Architecture Graph

The circles above represent different levels of software in an app. There are two principles to bear in mind about this graph:

1. The center circle is the most abstract, and the outer circle is the most concrete. This is called the **Abstraction Principle**. The Abstraction Principle specifies that inner circles should contain business logic, and outer circles should contain implementation details. In other words, the closer you are to the center, the less dependency on a specific platform you have.
2. Another principle of Clean Architecture is the **Dependency Rule**. This rule specifies that each circle can depend solely on the nearest inward circle — this is what makes the architecture work. This makes code based on Clean Architecture pretty decoupled and hence testable.

The basic components of Clean Architecture are as follows, explained from outer circles inward:

- **Presentation and Framework:** The outermost layer generally contains frameworks and tools specific to a platform. Using SwiftUI or Jetpack Compose for making interfaces? Here's the place. Using SwiftData or Room for database? They also belong here. You usually can't share code in this layer between platforms.
- **Controllers or Presenters:** This is the layer you used to have in MVC as Controller or ViewModel in MVVM. They receive input from the outer layer and pass them to the next layer. You can combine MVVM and MVC with Clean Architecture. It's also a good thing to do since the responsibilities of your controllers or ViewModels will decrease.
- **Use Cases or Interactors:** This layer defines the actions the user can trigger. The objects in the previous layer have access to use cases and can only call into the defined interactions. In the original definition of Clean Architecture, this is the layer you put your business logic in. As you're free to add your layers, you can delegate this responsibility to inner layers as well.
- **Entities:** Abstract definitions of all the data sources. It can contain some business logic.

While creating the **Organize** app, you're going to use the MVVM design pattern. You're free to choose any other pattern you like better for your applications.

Sharing Business Logic

KMP shines when you try to minimize the duplicated code you write. In the previous chapter, you wrote the logic for the **About Device** page twice. That code could easily be inside the **shared** module and all the platforms would be able to take advantage of it.

Creating ViewModels

Open the starter project in Android Studio. It's mostly the final project of the previous chapter.

Inside the **presentation** directory of the **commonMain** folder in the **shared** module, create a new file and name it **BaseViewModel.kt**.

Fill the file with this line:

```
expect abstract class BaseViewModel()
```

You're familiar with this line. This time, though, it's defining an abstract class, which all our app's ViewModels would extend. Next, you're going to implement the actual implementations of this class on all three platforms.

Put the cursor in the middle of the class name and press **Alt+Enter** and select **Add missing actual declarations**.

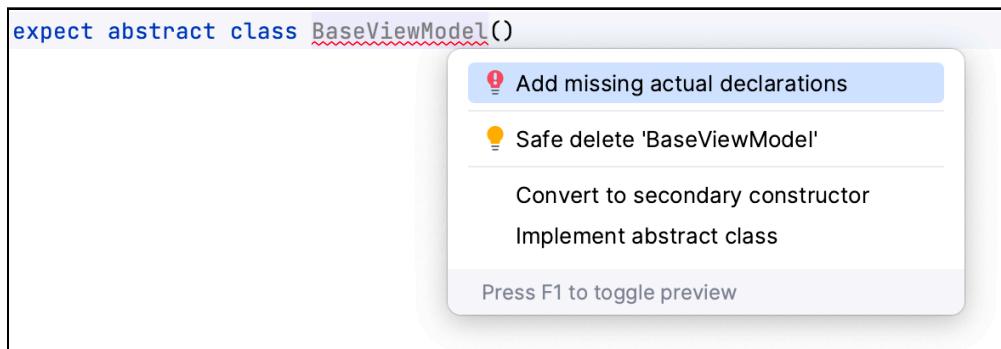


Fig. 7.3 – Alt+Enter on expect class name

Then, select **desktopMain**, **iosMain** and **androidMain**. Click **OK**. Android Studio will help you create all the needed actual files.

Note: If Android Studio fails to automatically create the needed actual files for you, don't worry. Create a file in the same package with the same name inside the missing platform's folder.

Open the Android version of **BaseViewModel.kt** and replace the content with this line:

```
actual abstract class BaseViewModel : ViewModel()
```

Don't forget to import the needed package:

```
import androidx.lifecycle.ViewModel
```

On Android, the ViewModels should extend the **Lifecycle** version of **ViewModel**, so they can survive the configuration changes on the devices.

If this is confusing to iOS developers, here's a small explanation:

On Android, when a configuration change occurs — for example, the device rotates or the user changes the system-wide theme or locale — the system recreates all the view components. However, the system will keep the same instance of the ViewModel extending from the Lifecycle package of AndroidX in memory. Hence, you can keep the view data inside the ViewModel and apply them to the newly created view components and the user won't notice anything.

For iOS and desktop, you don't need to extend anything. What Android Studio did for the actual files on those platforms is more than enough. They look like this:

```
actual abstract class BaseViewModel actual constructor()
```

Creating AboutViewModel

Now that you have a base.viewmodel, it's time to create the concrete versions. Start by creating a file named **AboutViewModel.kt** in the **commonMain** folder inside the **presentation** directory.

Define the class and subclass from the **BaseViewModel** you created earlier.

```
class AboutViewModel: BaseViewModel() {  
}
```

Inside the class, create an instance of the **Platform** class and press **Alt + Enter** to import it.

```
private val platform = Platform()
```

Define a data class inside the **AboutViewModel** class to hold the data you show in each row of the About page:

```
data class RowItem(  
    val title: String,  
    val subtitle: String,  
)
```

Next, create a function that generates the items for the About page. You wrote the same logic three times — once for each platform — in the previous chapter. You'll remove them all later.

```
private fun makeRowItems(platform: Platform): List<RowItem> {  
    val rowItems = mutableListOf(  
        RowItem("Operating System", "${platform.osName} $")
```

```
{platform.osVersion}),
    RowItem("Device", platform.deviceModel),
    RowItem("CPU", platform.cpuType),
)
platform.screen?.let {
    rowItems.add(
        RowItem(
            "Display",
            "${
                max(it.width, it.height)
            }x${
                min(it.width, it.height)
            } @${it.density}x"
        ),
    )
}
return rowItems
}
```

In the end, create an instance property to store the result of this function to avoid recreating the data. After all, these data would never change and are rather static.

```
val items: List<RowItem> = makeRowItems(platform)
```

This will be the public API of the ViewModel.

Using AboutViewModel in the View Layer

Android

Open **AboutView.kt** inside the **androidApp** module.

Remove the `makeItems()` method, as a similar implementation now exists inside **AboutViewModel**.

Next, edit the definition of the `AboutView` method as follows to account for the viewmodel:

```
@Composable
fun AboutView(
    viewModel: AboutViewModel = AboutViewModel(),
    onUpButtonClick: () -> Unit
)
```

For now, provide a default value for the `viewModel` parameter. In later chapters, you'll introduce dependency injection to your code and improve this instantiation.

Following that, update the `ContentView` method as follows:

```
@Composable
private fun ContentView(items: List<AboutViewModel.RowItem>) {
    LazyColumn(
        modifier = Modifier.fillMaxSize(),
    ) {
        items(items) { row ->
            RowView(title = row.title, subtitle = row.subtitle)
        }
    }
}
```

You injected the items this function needs for rendering, as a parameter. You also removed the now-unnecessary `makeItems` method invocation.

Now that the `ContentView` method requires a parameter, go back to the implementation of `AboutView` to pass the items in. Replace `ContentView()` with this:

```
ContentView(items = viewModel.items)
```

Build and run the Android app. It works just like before, but this time it uses a viewmodel.

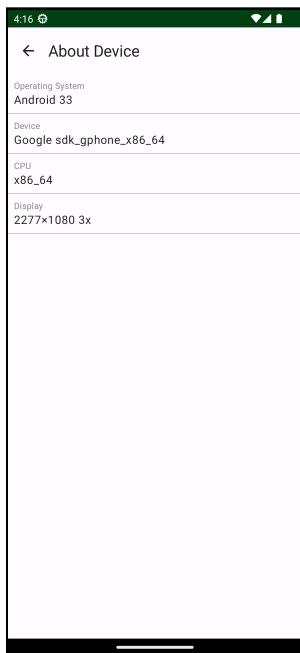


Fig. 7.4 – The About Device page of Organize on Android built using ViewModel.

iOS

Open the Xcode project and switch to **AboutView.swift**.

At the top of the file, make sure to import the **shared** module by adding this line:

```
import Shared
```

Next, inside the **AboutView** struct, add a property for the viewmodel:

```
@State private var viewModel = AboutViewModel()
```

You annotate the property with the **@State** directive to make SwiftUI create and hold an instance of **AboutViewModel** for the lifetime of **AboutView**.

Next, open **AboutListView.swift**. Remove the **RowItem** struct as well as the **items** property. Then, add a property to hold the items this view shows as follows:

```
let items: [AboutViewModel.RowItem]
```

Don't forget to import the shared module.

Inside the **AboutListView_Previews** struct, change **AboutListView()** invocation to the following:

```
AboutListView(items: [AboutViewModel.RowItem(title: "Title",  
subtitle: "Subtitle")])
```

In the code above, you are using a hardcoded row item to fix the UI preview inside Xcode.

Go back to **AboutView.swift** and pass the needed parameter to **AboutListView**:

```
AboutListView(items: viewModel.items)
```

Build and run to see the result of the refactoring you just did.

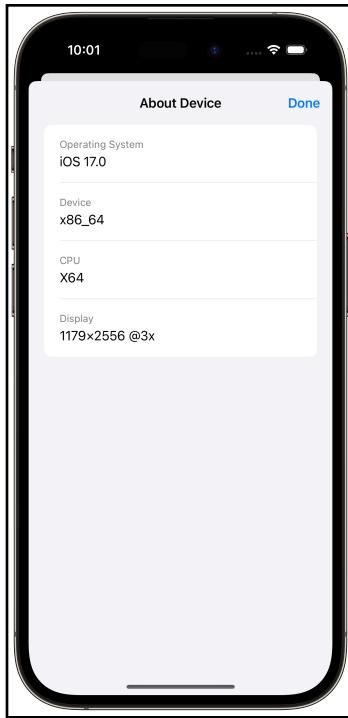


Fig. 7.5 – The About Device page of Organize on iOS built using ViewModel.

Desktop

You're now familiar with the process. Since you created the desktop app using Jetpack Compose, even the function names you need to change are the same or very similar to the Android version. Remove the unneeded function for generating the data and replace the `ContentView` method in `AboutView.kt` in the `desktopApp` module.

The result will look like this:

```
@Composable
fun AboutView(viewModel: AboutViewModel = AboutViewModel()) {
    ContentView(items = viewModel.items)
}

@Composable
private fun ContentView(items: List<AboutViewModel.RowItem>) {
    LazyColumn(
        modifier = Modifier.fillMaxSize(),
```

```
) {  
    items(items) { row ->  
        RowView(title = row.title, subtitle = row.subtitle)  
    }  
}
```

Build and run the desktop app and see the changes...or the lack thereof!

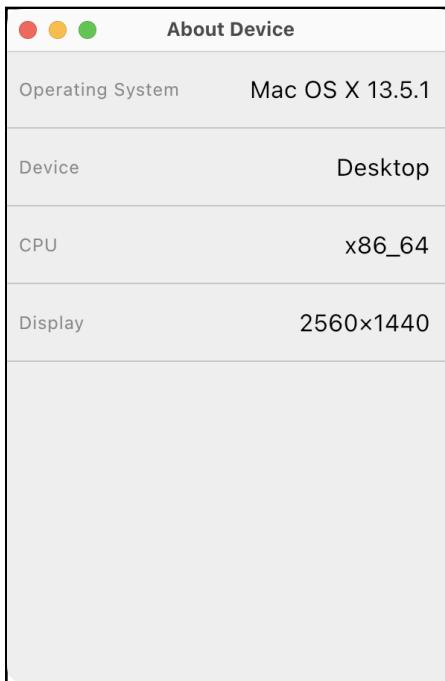


Fig. 7.6 – The About Device page of Organize on desktop built using ViewModel.

Creating Reminders Section

Until now, you were working on a supplementary page of the app. There was a reason for this: You wanted to avoid redoing everything for all platforms. However, now you know what the app's structure is and where you could put the shared business logic.

Repository Pattern

A first idea for implementing the **RemindersViewModel** might involve directly creating, updating and deleting reminders and exposing the data and the actions to **RemindersView**. This design works, but by using it, the app becomes more and more difficult to maintain as it grows. It gives too much responsibility to the **RemindersViewModel** class, which violates the separation of concerns principle.

For instance, when you start integrating a database into the app in later chapters, you would need to update many things in the viewmodel.

One way to mitigate this issue is to use a **Repository Pattern**. A repository is an object that sits in between the viewmodel and the source of your data, whether it's a remote server, a local database or even a cache in memory.

Create a new directory as a sibling to **presentation** deep inside the **commonMain** folder of the **shared** module and name it **data**. Then, create a new Kotlin class named **RemindersRepository** inside the **data** directory.

First, add a property to hold the **Reminders** objects internally:

```
private val _reminders: MutableList<Reminder> = mutableListOf()
```

You'll get a compiler error stating that the **Reminder** type is unresolved. **Reminder** will be a data model for our app. To keep things more organized, you're going to create the **Reminder** class inside a directory named **domain**, which is another sibling of **presentation** and **data**. If you pay close attention, you'll notice that there are some cues from the Clean Architecture here. But don't worry — you'll only use some naming conventions and won't dig deeper than that.

Create the **Reminder.kt** file and add this block of code:

```
data class Reminder(
    val id: String,
    val title: String,
    val isCompleted: Boolean = false,
)
```

Each reminder will have an identifier, a title and a value for whether it's completed or not.

Next, in `RemindersRepository`, add the import for the `Reminder` class.

Then, add this function to create a new reminder:

```
fun createReminder(title: String) {
    val newReminder = Reminder(
        id = UUID().toString(),
        title = title,
        isCompleted = false
    )
    _reminders.add(newReminder)
}
```

`UUID` is a class used to create random identifiers with the expect/actual mechanism. It's already there in the starter project. Take a look at its implementation if you're interested.

Next, add a function to update the `isCompleted` status of a reminder:

```
fun markReminder(id: String, isCompleted: Boolean) {
    val index = _reminders.indexOfFirst { it.id == id }
    if (index != -1) {
        _reminders[index] = _reminders[index].copy(isCompleted =
            isCompleted)
    }
}
```

It first checks if an item with the `id` exists. If the answer is yes, it updates the `isCompleted` value.

In the end, create a public getter property for all the reminders. Later, you'll change this to a **Kotlin Flow** to be able to propagate live changes to the viewmodel and view. Since using Flows on iOS is a bit tricky, you'll stick to plain properties for now.

```
val reminders: List<Reminder>
    get() = _reminders
```

You've created a nice-looking API for the repository. Good job!

Creating RemindersViewModel

Inside the **presentation** directory of **commonMain** module, create a new class named **RemindersViewModel**. Update it with the following:

```
class RemindersViewModel : BaseViewModel() {  
    //1  
    private val repository = RemindersRepository()  
  
    //2  
    private val reminders: List<Reminder>  
        get() = repository.reminders  
  
    //3  
    var onRemindersUpdated: ((List<Reminder>) -> Unit)? = null  
        set(value) {  
            field = value  
            onRemindersUpdated?.invoke(reminders)  
        }  
  
    //4  
    fun createReminder(title: String) {  
        val trimmed = title.trim()  
        if (trimmed.isNotEmpty()) {  
            repository.createReminder(title = trimmed)  
            onRemindersUpdated?.invoke(reminders)  
        }  
    }  
  
    //5  
    fun markReminder(id: String, isCompleted: Boolean) {  
        repository.markReminder(id = id, isCompleted = isCompleted)  
        onRemindersUpdated?.invoke(reminders)  
    }  
}
```

Here's what this class includes:

1. A property to keep a strong reference to the repository.
2. A property that accesses reminders from the repository.
3. Views can connect to this property to find out about changes in reminders. For now, it's the **link** or the **binding** component of MVVM. You make sure to call the lambda, or closure in Swift terms, with the current state of reminders at its setter block.

4. A method for creating a reminder after safeguarding against reminders with empty titles. When `viewModel` asks the repository to create a new reminder, it propagates the changes through `onRemindersUpdated`.
5. A method for changing the `isCompleted` property of a specific reminder.

Updating View on Android

Open **RemindersView.kt** inside the **androidApp** module. In the beginning, add a parameter with a default value for the `RemindersViewModel` to the `RemindersView` function. Then, pass `viewModel` into the `ContentView` method.

```
@Composable
fun RemindersView(
    viewModel: RemindersViewModel = RemindersViewModel(),
    onAboutButtonClick: () -> Unit,
) {
    Column {
        Toolbar(onAboutButtonClick = onAboutButtonClick)
        ContentView(viewModel = viewModel)
    }
}
```

You'll give the `ContentView` function a massive upgrade. First, remove the existing code in the function. Next, change the function signature to accept a parameter of type `RemindersViewModel`.

```
@Composable
private fun ContentView(viewModel: RemindersViewModel) {
```

Add a variable for remembering the state of reminders. Jetpack Compose re-renders, or recomposes the function whenever this state changes.

```
var reminders by remember {
    mutableStateOf(listOf<Reminder>(), policy =
neverEqualPolicy())
}
```

In the code above, `by` is a delegate syntax. Doing so delegates the `get()` and `set()` methods to the `remember` method.

Add the following imports to resolve the IDE warnings and errors:

```
import androidx.compose.runtime.getValue
import androidx.compose.runtime.setValue
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.neverEqualPolicy
import androidx.compose.runtime.remember
```

To connect the underlying changes of reminders in `viewModel` to this function, add this line after initializing `reminders`:

```
viewModel.onRemindersUpdated = {
    reminders = it
}
```

Whenever the viewmodel calls the `onRemindersUpdated` lambda, you set the new value of the `reminders` list to the `reminders` state variable. This makes the component react to changes. The policy you set in an earlier step will make sure the recomposition always happens, regardless of the equality status of new and old values.

Subsequently, create a `LazyColumn` to show the reminders in a list:

```
LazyColumn(modifier = Modifier.fillMaxSize()) {
    //1
    items(items = reminders) { item ->

        //2
        val onClick = {
            viewModel.markReminder(id = item.id, isCompleted = !item.isCompleted)
        }

        //3
        ReminderItem(
            title = item.title,
            isCompleted = item.isCompleted,
            modifier = Modifier
                .fillMaxWidth()
                .clickable(enabled = true, onClick = onClick)
                .padding(horizontal = 16.dp, vertical = 4.dp)
        )
    }
}
```

1. Using the `items` composable function, you provide the `reminders` state variable to the `LazyColumn` function. `LazyColumn` is an efficient version of `List` that renders only the subset of items that can be displayed on the screen.
2. Store a lambda, that calls into `viewModel` to update the `isCompleted` status of a particular reminder.
3. Use the already provided `ReminderItem` function for each row of the list. You are welcome to take a look at its implementation.

After the `items` block, you add an `item` that will contain the text field to add new reminders.

```
item {  
    //1  
    val onSubmit = {  
        viewModel.createReminder(title = textFieldValue)  
        textFieldValue = ""  
    }  
  
    //2  
    NewReminderTextField(  
        value = textFieldValue,  
        onValueChange = { textFieldValue = it },  
        onSubmit = onSubmit,  
        modifier = Modifier  
            .fillMaxWidth()  
            .padding(vertical = 8.dp, horizontal = 16.dp)  
    )  
}
```

1. An `onSubmit` lambda that creates a new reminder, and clears the text field when the user presses the **Return** or the **Done** key on their phone's keyboard.
2. A customized `NewReminderTextField` is inside the starter project. You bind the `value` and the `onValueChange` to the `textFieldValue` state variable.

Finally, add the `textFieldValue` state variable at the top of the `ContentView` function as follows:

```
var textFieldValue by remember { mutableStateOf("") }
```

Build and run the app. Add a couple of reminders and mark a few of them as done.

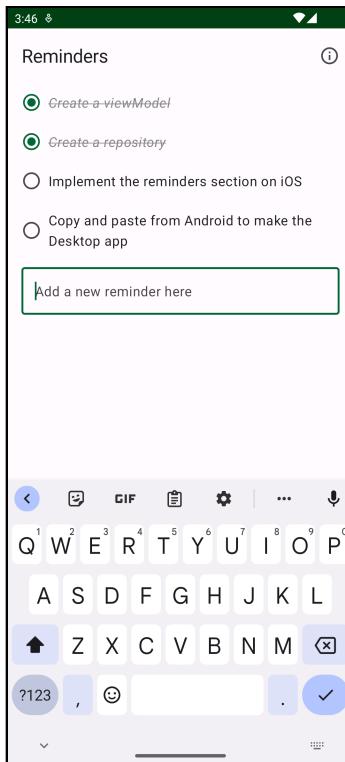


Fig. 7.7 – The Reminders first page on Android

Updating the View on iOS

For the reactive nature of data binding to function effectively, SwiftUI heavily relies on the **Combine** framework or, as of iOS 17, the **Observation** framework. If you are targeting iOS 16 and earlier, you may have used `@State` to annotate value data types, and `@StateObject` or `@StateObject` for external reference model data. With the advent of iOS 17, Apple has simplified state management. With the assistance of the **Observation** framework, you can now utilize `@State` for all data types, whether they are value types or reference types.

However, there's a catch in using **RemindersViewModel**. Since you defined the viewmodel inside the KMP **Shared** module, you weren't able to use Combine or Observation there, as they're Swift-only.

One way to address the issue is to create a wrapper around the viewmodel and expose the properties for SwiftUI to use.

Open the `iosApp.xcodeproj` and create a new Swift file by pressing **Command-N**. Name it `RemindersViewModelWrapper.swift` and place it in the `Reminders` directory.

Add the following code to the file:

```
//1
import Observation
import Shared

//2
@Observable
final class RemindersViewModelWrapper {
    //3
    let viewModel = RemindersViewModel()

    //4
    private(set) var reminders: [Reminder] = []

    init() {
        //5
        viewModel.onRemindersUpdated = { [weak self] items in
            self?.reminders = items
        }
    }
}
```

1. You should import `Observation` as well as the `Shared` framework.
2. By annotating your class with the `@Observable` macro, the class becomes observable by SwiftUI view.
3. Here, you hold a strong reference to the *real* viewmodel.
4. You expose a property out of this class. SwiftUI will re-render the body of the view when this property changes.
5. At initialize, you subscribe to `onRemindersUpdated` closure of `viewModel` and update your published property accordingly. By using `[weak self]`, you break a potential memory cycle.

Open `RemindersView.swift` and replace the struct with the following code. It's rather long, but it looks and behaves a lot like what you did with Jetpack Compose:

```
struct RemindersView: View {
    //1
    @State private var viewModelWrapper =
        RemindersViewModelWrapper()
```

```
//2
@State private var textFieldValue = ""

var body: some View {
    //3
    List {
        //4
        if !viewModelWrapper.reminders.isEmpty {
            Section {
                ForEach(viewModelWrapper.reminders, id: \.id) { item
                    in
                        //5
                        ReminderItem(title: item.title, isCompleted:
                            item.isCompleted)
                            .onTapGesture {
                                //6
                                withAnimation {
                                    viewModelWrapper.viewModel.markReminder(
                                        id: item.id,
                                        isCompleted: !item.isCompleted
                                    )
                                }
                            }
            }
        }
    }
}

//7
Section {
    NewReminderTextField(text: $textFieldValue) {
        withAnimation {
            viewModelWrapper.viewModel.createReminder(title:
                textFieldValue)
            textFieldValue = ""
        }
    }
}.navigationTitle("Reminders")
}
```

1. Using the `@State` annotation, you create an instance of the wrapper you defined in the previous step.
2. Using the `@State` annotation, you create a property to hold the text field's text value. You even used the same variable name in Jetpack Compose.
3. `List` in SwiftUI is essentially the equivalent of `LazyColumn` in Jetpack Compose.

4. If the `reminders` property contains values, you create a section with reminder items.
5. For each row of the list in the first section, you use an instance of `ReminderItem` that's inside the starter project.
6. When the user taps on each row, you call into `viewModel` to mark the reminder as completed or uncompleted. The `withAnimation` function makes the transition look smooth.
7. This section is there to create a text field for adding new items. You bind it to the `textFieldValue` property.

Build and run, and take a look at the Reminders page in all its glory!

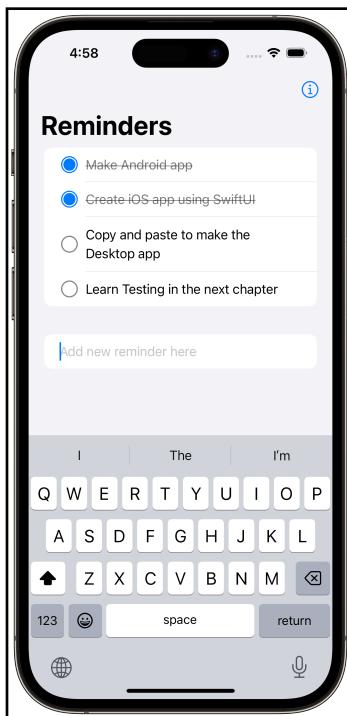


Fig. 7.8 – The Reminders first page on iOS

Updating View on Desktop

Since the desktop app is using Jetpack Compose, you can literally copy and paste the code from `RemindersView.kt` in the `androidApp` module to the same file in the `desktopApp` module.

After doing so, build and run the desktop app.

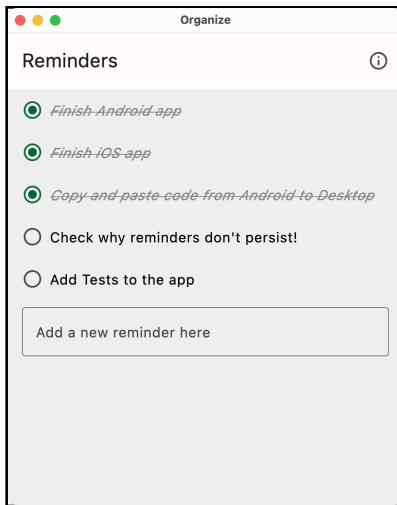


Fig. 7.9 – The Reminders first page on desktop

One point to mention is that the apps forget your reminders whenever you relaunch them or navigate to another page and come back. This is because you're storing the reminders in a property inside the repository. In later chapters, when you integrate a database, you'll fix this.

In the final project, there are a couple of touches for improving keyboard support — such as focus switch. For brevity's sake, they weren't in this chapter.

Sharing Tests and UI

By sharing business logic, you reduce the code you need to write for each platform to their respective UI code.

In the next chapter, you're going to add tests to the project. Since all the business logic now resides in a single place, you'll write a single set of tests. Hence, you'll write fewer test codes — which means you're secretly rejoicing!

You might have thought it was a little weird to copy and paste code between Android and desktop. And, you might be thinking of a way to share these pretty similar pieces of code. Since you've been using Jetpack Compose for both of these platforms, there are a couple of ways to share these codes. You'll learn more about one option in [Appendix C](#).

One thing to notice is that sharing UI code may not always be a good decision for a couple of reasons:

- You may have noticed that the desktop app looks a bit unusual aesthetically. It's adhering to Material Design guidelines, which isn't a typical approach on the desktop. It doesn't look like other native apps on Windows or macOS, either. Many would prefer to stick to the native UI toolkit of each platform instead of using Jetpack Compose, which uses Java Swing under the hood. For that matter, many developers wouldn't create their desktop app using the approach you saw in this book. If you don't do that, you won't have Jetpack Compose for Desktop, and therefore, no code to share between Android and desktop.
- Each platform has its differences. A desktop app would usually need a different design than the Android app. For instance, it doesn't need to have large touch targets, it doesn't have multitouch, it mostly uses a mouse and keyboard instead of touch as input, etc. If you want to create a great app for each platform, you need to take these into consideration.

All these explanations apply to UI testing as well. If you somehow share UI code, you can have shared UI tests. If you don't, you'll need to create UI tests for each platform separately.

Challenge

Here's a challenge for you to see if you mastered this chapter. The solution is waiting for you inside the materials for this chapter.

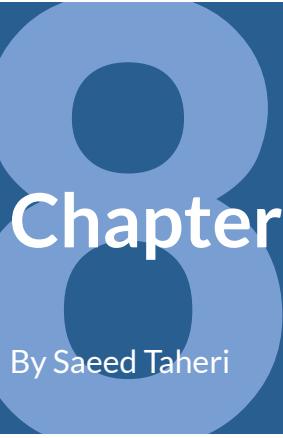
Challenge: Moving Page Titles to Viewmodels

As viewmodels are responsible for making everything ready for views to show, you can make the viewmodels provide the page title to their respective views. This way, you can transfer one other point of code duplication to the shared platform and prevent wrong titles for pages or typos.

You can add the `title` property to both `AboutViewModel` and `ReminderViewModel` and then utilize it in the respective views across all platforms.

Key Points

- You can use any design pattern you see fit with Kotlin Multiplatform.
- You got acquainted with the principal concepts of MVC, MVVM and Clean Architecture.
- Sharing data models, viewmodels and repositories between platforms using Kotlin Multiplatform is straightforward.
- You can share business logic tests using Kotlin Multiplatform.
- Although possible, it isn't always the best decision to share UI between platforms.



Chapter 8: Testing

By Saeed Taheri

Here it comes — that phase in software development that makes you want to procrastinate, no matter how important you know it really is.

Whether you like it or not, having a good set of tests — both automated and manual — ensures the quality of your software. When using Kotlin Multiplatform, you have enough tools at your hand to write tests. So if you're thinking of letting it slide this time, you'll have to come up with another excuse. :]

Setting Up the Dependencies

Testing your code in the KMP world follows the same pattern you're now familiar with. You test the code in the **common** module. You may also need to use the **expect/actual** mechanism as well. With this in mind, setting up the dependencies is structurally the same as it is with non-test code.

From the starter project, using Android Studio, open the **build.gradle.kts** file inside the **shared** module. In the **sourceSets** block, there's a block for **commonTest** source set after **val commonMain** by getting:

```
val commonTest by getting {
    dependencies {
        implementation(kotlin("test"))
    }
}
```

This is a dependency on the **kotlin.test** library. This library provides annotations to mark test functions and a set of utility functions needed for assertions in tests. Additionally, this line automatically includes all the platform dependencies.

Do a Gradle sync if required.

As you declared above, your test codes will be inside the **commonTest** folder. Create it as a sibling directory to **commonMain** by right-clicking the **src** folder inside the **shared** module and choosing **New > Directory**. Once you start typing **commonTest**, Android Studio will provide you with autocompletion. Choose **commonTest/kotlin**.

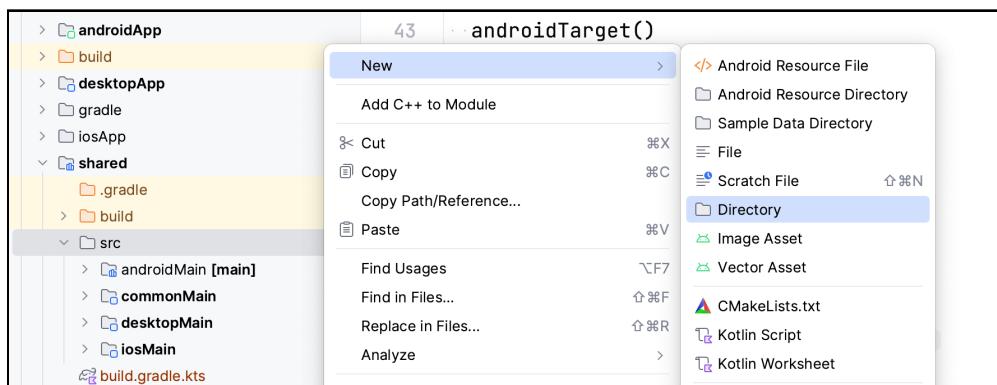


Fig. 8.1 – Create a new directory in Android Studio

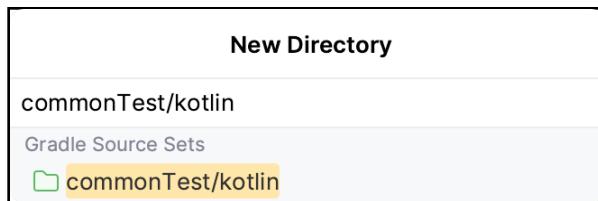


Fig. 8.2 – Android Studio suggests naming the test directory

Note: Although not necessary, it's a good practice to have your test files in the same package structure as your main code. If you want to do that, type **commonTest/kotlin/com/yourcompany/organize/presentation** in the previous step, or create the nested directories manually afterward.

Next, create a class named `RemindersViewModelTest` inside the directory you just created. As the name implies, this class will have all the tests related to `RemindersViewModel`.

Now it's time to create the very first test function for the app. Add this inside the newly created class:

```
@Test  
fun testCreatingReminder() {  
}
```

You'll implement the function body later. The point to notice is the `@Test` annotation. It comes from the `kotlin.test` library. Make sure to import the needed package at the top of the file if Android Studio didn't do it automatically for you:
`import kotlin.test.Test.`

As soon as you add a function with `@Test` annotation to the class, Android Studio shows run buttons in the code gutter to make it easier for you to run the tests.

```
39 ►  class RemindersViewModelTest {  
40    ...  
41 ►    @Test  
42    ...  
43    fun testCreatingReminder() {  
44      ...  
45    }  
46  }
```

Fig. 8.3 – Run button for tests in code gutter

You can run the tests by clicking on those buttons, using commands in the terminal, or by pressing the keyboard shortcut **Control-Shift-R** on Mac or **Control-Shift-F10** on Windows and Linux.

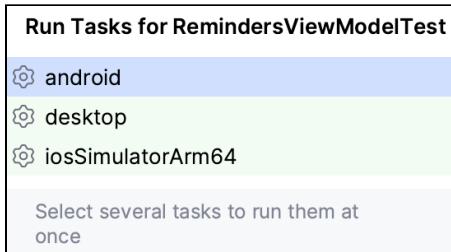


Fig. 8.4 – Choosing test platform

As an example, choose **android** to run the test on Android.

Congratulations! You ran your first test successfully.

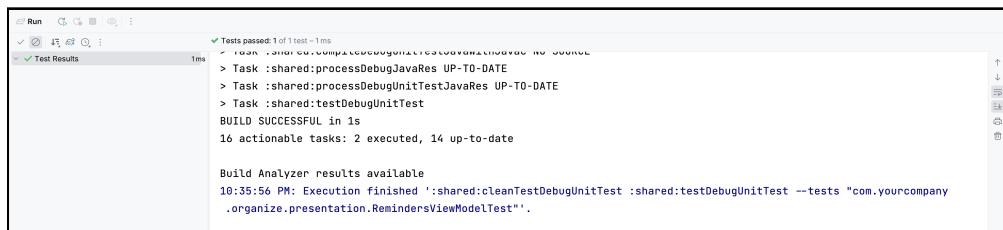


Fig. 8.5 – First test successful

When you ask the system to run the test on any platform, it needs to find a test library on that platform to run your tests on. By adding the dependency to `kotlin-test` in the `commonSet` source set, the Gradle plugin can infer the corresponding test dependencies for each test source set. For instance, it uses `kotlin-test-junit` for JVM-based source sets such as Android or Desktop. Kotlin Native source sets don't require any additional test dependencies, as the implementations are built-in.

Writing Tests for RemindersViewModel

With the dependencies for unit testing all in place, it's time to create some useful test functions.

Since you're testing the `viewModel`, you require an instance of `RemindersViewModel` at hand. Add a `lateinit` property in the class for this matter as follows:

```
private lateinit var viewModel: RemindersViewModel
```

Next, you need to somehow initialize this property. When writing tests, you can tag a function with `@BeforeTest` annotation. This will make sure that the specific function runs before every test in the class. That seems a good place to *set up* the `viewModel`.

Add this function to the class:

```
@BeforeTest
fun setup() {
    viewModel = RemindersViewModel()
```

Note: There's a `@AfterTest` annotation as well. As the name implies, it runs after each test in the class. You can use functions tagged with this annotation to do any needed cleanup.

As for the body of `testCreatingReminder()`, update it with:

```
@Test
fun testCreatingReminder() {
    //1
    val title = "New Title"

    //2
    viewModel.createReminder(title)

    //3
    val count = viewModel.reminders.count {
        it.title == title
    }

    //4
    assertTrue(
        actual = count == 1,
        message = "Reminder with title: $title wasn't created.",
    )
}
```

1. First, you create a title constant.
2. You use the `createReminder` method of the `viewModel` to create a new reminder.
3. Next, you check the number of items in `reminders` property of the `viewModel` having the title you used. If you faced an error about the visibility of `reminders`, don't worry. You'll fix it soon.

4. `kotlin.test` library includes several **assert** functions, which you can take advantage of. Here, you're using `assertTrue` to check if `count` equals 1. If that's true, it means the creation process was successful. If not, you show a message in the console.

The `reminders` property in **RemindersViewModel** was **private** when you wrote it. Since **commonTest** is in the same module as **commonMain**, you can change the visibility modifier for that property to **internal**. This way, outsiders using the **shared** module such as **androidApp** and **iosApp** won't see any changes and the property would be visible to your test functions.

Open **RemindersViewModel.kt** and change the aforementioned property to this:

```
internal val reminders: List<Reminder>
    get() = repository.reminders
```

Now it's time to run the test. To run the tests on all platforms at once, you can try either of these actions:

- Choose **allTests** from the list of tasks in the Gradle pane in Android Studio.

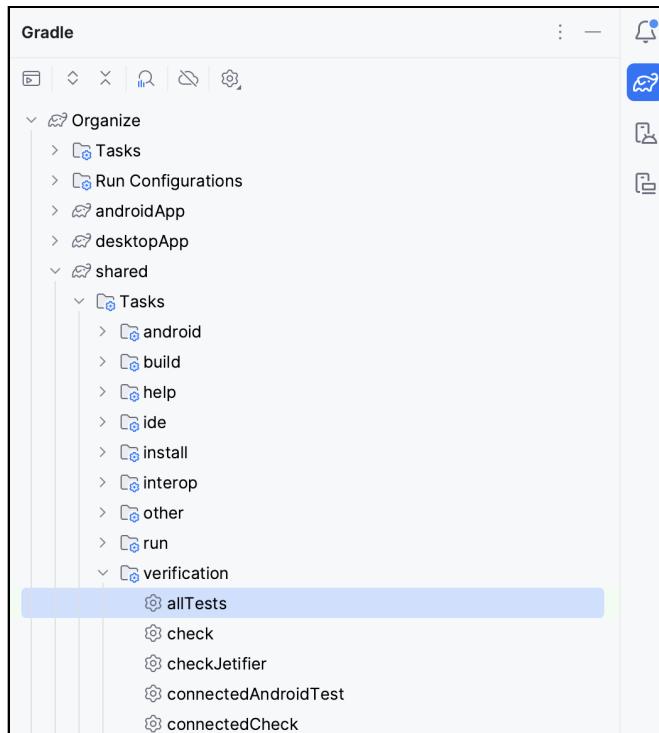
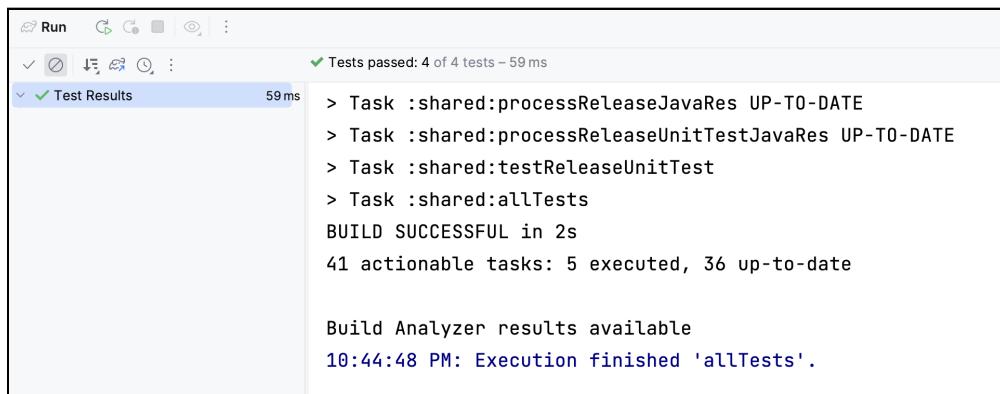


Fig. 8.6 — Choosing `allTests` from Gradle pane

- Run the command `./gradlew :shared:allTests` in Terminal while you're in the working directory of the project.

Whatever option you pick, you will have a successful test for all platforms. Hooray!



The screenshot shows the 'Test Results' tab in the Android Studio interface. The status bar at the top indicates 'Tests passed: 4 of 4 tests – 59 ms'. The main pane displays the build log:

```
> Task :shared:processReleaseJavaRes UP-TO-DATE
> Task :shared:processReleaseUnitTestJavaRes UP-TO-DATE
> Task :shared:testReleaseUnitTest
> Task :shared:allTests
BUILD SUCCESSFUL in 2s
41 actionable tasks: 5 executed, 36 up-to-date

Build Analyzer results available
10:44:48 PM: Execution finished 'allTests'.
```

Fig. 8.7 – Successful test for creating a reminder

Writing Tests for Platform

All implementation details of the **RemindersViewModel** class were inside the **commonMain** source set. However, the **Platform** class is a bit different. As you remember, **Platform** uses the expect/actual mechanism. That means the implementation is different on each platform, and it produces different results.

To address this matter, you need to have multiple test suites. Those would follow the source sets pattern you saw in previous steps. Create **androidUnitTest**, **iosTest** and **desktopTest** directories in the **shared** module. Don't forget to add **com/yourcompany/organize** directories.

You have two choices: Either you use the same expect/actual mechanism for your test class, or you create the test classes independent of each other in each source set. In both methods, the system will run all the functions annotated with `@Test`. However, since expect/actual will force you to fulfill the expected test functions, it's a safer choice from a structural standpoint.

In **commonTest** folder, create a class named **PlatformTest** under the **com.yourcompany.organize** package and define the class like this:

```
expect class PlatformTest {
    @Test
```

```
    fun testOperatingSystemName()  
}
```

Here, you're promising to implement a test function named `testOperatingSystemName`. You can add any test function, but for the sake of brevity, this is the only Platform test function you'll see in this chapter.

You've heard a lot about how to create actual classes. If you aren't yet comfortable enough with the process, go back and take a look at Chapter 6.

Android

Create `PlatformTest.kt` inside the directories you created earlier in `androidTest` and update as follows:

```
import kotlin.test.DefaultAssert.assertEquals  
  
actual class PlatformTest {  
    private val platform = Platform()  
  
    @Test  
    actual fun testOperatingSystemName() {  
        assertEquals(  
            expected = "Android",  
            actual = platform.osName,  
            message = "The OS name should be Android."  
    }  
}
```

Pretty straightforward, isn't it? You assert that the operating system name should be "Android".

However, this test will probably fail. As of writing this chapter, there's an open issue at <https://issuetracker.google.com/issues/191287536> where the tests on Android run as instrument tests instead of JUnit tests. This causes some problems. For instance, you'll get some errors telling you that `Build.SUPPORTED_ABIS` shouldn't be null, or you need to mock `Resources.getSystem()`. As a hacky workaround, you can edit the `Platform` implementation on Android to not instantiate problematic properties right away.

Open **Platform.kt** in **androidMain** module and change these values:

```
actual val cpuType =  
    Build.SUPPORTED_ABIS?.firstOrNull() ?: "___"  
  
actual val screen: ScreenInfo  
    get() = ScreenInfo()
```

Now run the tests for Android again, and this particular test you wrote will pass successfully. However, it's better to hope unit test issues go away soon.

iOS

Open **build.gradle.kts** for the **shared** module and add these lines to the **sourceSets** block.

```
val iosX64Test by getting  
val iosArm64Test by getting  
val iosSimulatorArm64Test by getting  
val iosTest by creating {  
    dependsOn(commonTest)  
    iosX64Test.dependsOn(this)  
    iosArm64Test.dependsOn(this)  
    iosSimulatorArm64Test.dependsOn(this)  
}
```

This helps you compile the tests written inside the **iosTest** source set.

Create **PlatformTest.kt** inside the directories you created earlier in **iosTest** and update as follows:

```
@kotlinx.cinterop.ExperimentalForeignApi  
@kotlin.experimental.ExperimentalNativeApi  
actual class PlatformTest {  
    private val platform = Platform()  
  
    @Test  
    actual fun testOperatingSystemName() {  
        assertTrue(  
            actual = platform.osName.equals("iOS", ignoreCase = true)  
            || platform.osName == "iPadOS",  
            message = "The OS name should either be iOS or iPadOS."  
        )  
    }  
}
```

You check if the OS name is either **iOS** or **iPadOS**.

Desktop

Create **PlatformTest.kt** inside the directories you created earlier in **desktopMain** and update as follows:

```
actual class PlatformTest {
    private val platform = Platform()

    @Test
    actual fun testOperatingSystemName() {
        assertTrue(
            actual = platform.osName.contains("Mac", ignoreCase =
true) || platform.osName.contains("Windows", ignoreCase =
true) || platform.osName.contains("Linux", ignoreCase = true)
            || platform.osName == "Desktop",
            message = "Non-supported operating system"
        )
    }
}
```

This is a bit difficult to test properly. For now, you can check if the reported OS name contains the app's supported platforms. If not, let the test fail. If you run the **allTests** Gradle task as before, the system will run these tests as well. Try it to see a new batch of successful tests.

UI Tests

Until now, the approach you've followed in this book is to share the business logic in the **shared** module using Kotlin Multiplatform and create the UI in each platform using the available native toolkit. Consequently, you've been able to share the tests for the business logic inside the **shared** module as well.

For testing UI, you can safely assume that there's no KMP in place. You test Android and desktop UIs using **Compose Tests**, and iOS UI using **XCUITest**.

Android

You created the UI for **Organize** entirely using Jetpack Compose. Testing Compose layouts is different from testing a View-based UI. The View-based UI toolkit defines what properties a View has, such as the rectangle it's occupying, its properties and so forth. In Compose, some composables may emit UI into the hierarchy. Hence, you need a new matching mechanism for UI elements.

Fortunately, the creators of Jetpack Compose had this in mind and provided the necessary tools to test layouts.

Open **build.gradle.kts** within **androidApp** and add these inside the dependencies block:

```
androidTestImplementation(platform(libs.androidx.compose.bom))
debugImplementation(libs.androidx.ui.test.manifest)
androidTestImplementation(libs.junit)
androidTestImplementation(libs.androidx.ui.test.junit4)
androidTestImplementation(libs.androidx.fragment.testing)
androidTestImplementation(libs.androidx.test.runner)
```

In the **defaultConfig** section of android block, add this to tell the system how to run the tests:

```
testInstrumentationRunner =
"androidx.test.runner.AndroidJUnitRunner"
```

Go ahead and sync your project now. Next, it's time to create packages and files. Inside the **src** directory, create these nested folders: **androidTest/java/com/yourcompany/organize/android/**

You're going to replicate the same package structure of the **main** directory.

Next, create a Kotlin file called **AppUITest.kt** and add the following code:

```
import androidx.compose.ui.test.junit4.createAndroidComposeRule
import com.yourcompany.organize.android.ui.root.MainActivity
import org.junit.Rule

class AppUITest {
    @get:Rule
    val composeTestRule = createAndroidComposeRule<MainActivity>()
}
```

In the code above, you add a property of type **AndroidComposeTestRule**. The **createAndroidComposeRule** creates a test rule for the activity you provide. It brings up the activity, so you can run your tests.

The very first test you'll write is to check for the existence of the **About** button. As you remember, the button is in the top right corner of the app and has an **i** icon. The way you can match that element in your tests is through a mechanism named **Semantics**.

Semantics

Semantics give meaning to a piece of UI — whether it's a simple button or a whole set of composables. The semantics framework is primarily there for accessibility purposes. However, tests can take advantage of the information exposed by semantics about the UI hierarchy.

You attach semantics to the composables through a **Modifier**.

Open **RemindersView.kt** in the **androidApp** module and attach a **semantics** modifier to the **IconButton** in the **Toolbar** compasable. The **IconButton** will look like this:

```
IconButton(  
    onClick = onAboutButtonClick,  
    modifier = Modifier.semantics { contentDescription =  
        "aboutButton" },  
    ) {  
    Icon(  
        imageVector = Icons.Outlined.Info,  
        contentDescription = "About Device Button",  
    )  
}
```

You're attaching semantics to the **IconButton** with the content description **aboutButton**.

Go back to **AppUITest.kt** and add the following functon:

```
@Test  
fun testAboutButtonExistence() {  
    composeTestRule  
        .onNodeWithContentDescription("aboutButton")  
        .assertIsDisplayed()  
}
```

Using the **composeTestRule** you defined, you query a node with the content description you set, and assert if it's displayed.

Run the test using the run button in the code gutter. You'll see that the emulator or the connected device runs your app for an instant and then closes it.

If everything goes well, you should see the report for a passed test.

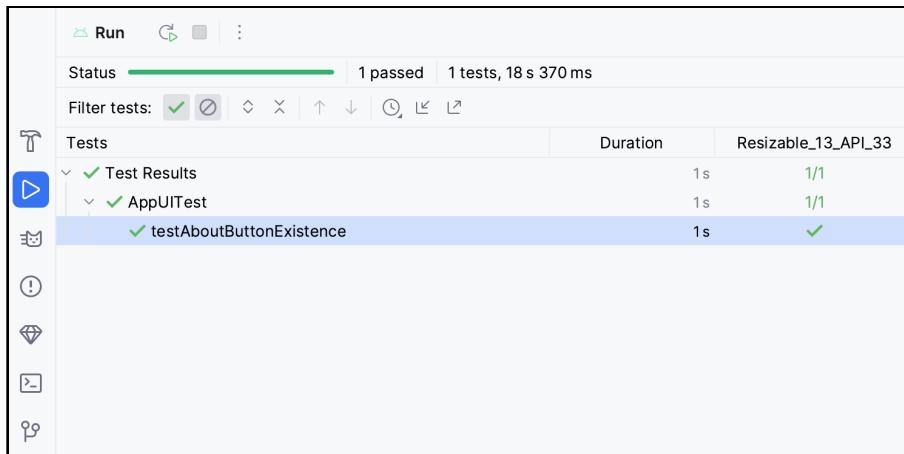


Fig. 8.8 – Successful test for about button existence

Next up is testing whether the **About** page opens and closes successfully. Add the following function to test the same:

```
@Test
fun testOpeningAndClosingAboutPage() {
    //1
    composeTestRule
        .onNodeWithContentDescription("aboutButton")
        .performClick()

    //2
    composeTestRule
        .onNodeWithText("About Device")
        .assertIsDisplayed()

    //3
    composeTestRule
        .onNodeWithContentDescription("Up Button")
        .performClick()

    //4
    composeTestRule
        .onNodeWithText("Reminders")
        .assertIsDisplayed()
}
```

In this code:

1. You find the **About** button using the semantics you defined and simulate performing a click on it.
2. Check if there's a text on the screen with **About Device** content. The **About** page has this title, and it's only there if that page is onscreen. This is not a good way to do it, though. This test will fail if you localize your app in another language. Using semantics is always a better choice.
3. If you'd set content description on buttons as you did with the **Up Button** in the toolbar, you can use that without setting and querying semantics. You find the button and perform a click on it.
4. When you close the **About** page, the app should be in the **Reminders** page. Check for the page title if this is the case.

Run the test, and it will pass.

Desktop

As the UI code for Android and desktop are essentially the same, the tests will be very similar. The setup is a bit different, though. The code is already there for you in the starter project. These are the differences you should consider:

- Test dependencies are different. Take a look at **build.gradle.kts** in **desktopApp** module.

```
val jvmTest by getting {
    dependencies {
        implementation(compose.desktop.uiTestJUnit4)
        implementation(compose.desktop.currentOs)
    }
}
```

```
@Before
fun setUp() {
    composeTestRule.setContent {
        var screenState by remember
        { mutableStateOf(Screen.Reminders) }

        when (screenState) {
            Screen.Reminders ->
            RemindersView(
                onAboutButtonClick = { screenState =
Screen.AboutDevice }
        )
    }
}
```

```
        Screen.AboutDevice -> AboutView()
    }
}
```

- Since there are no windows in this test suite, the second test function will be like this:

```
@Test
fun testOpeningAboutPage() {
    //1
    composeTestRule
        .onNodeWithText("Reminders")
        .assertIsDisplayed()

    //2
    composeTestRule
        .onNodeWithContentDescription("aboutButton")
        .performClick()

    //3
    composeTestRule.waitForIdle()

    //4
    composeTestRule
        .onNodeWithContentDescription("aboutView")
        .assertIsDisplayed()
}
```

1. First, you check if you're in the **Reminders** page by asserting the existence of the Reminders title.
2. You simulate a click on the **About** button.
3. Next, wait for the recomposition to finish. When the Compose test rule was an **Activity**, it did this automatically. Now, it's your job to make your tests wait.
4. Lastly, check if an element with the semantics **aboutView** exists in the hierarchy.

Run your test suite to see them all pass.

iOS

To make the UI code testable in Xcode, you need to add a UI Test target to your project. While the iOS app project is open in Xcode, choose **File > New > Target...** from the menu bar.

Scroll down until you find **UI Testing Bundle**.

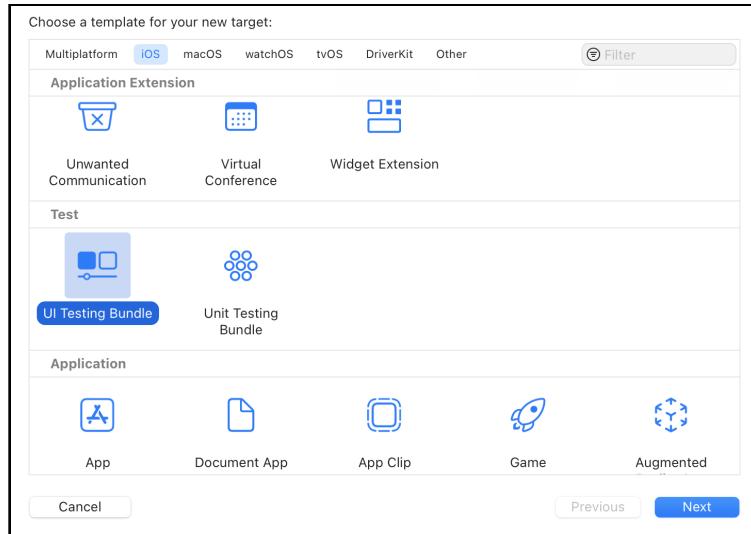


Fig. 8.9 – Xcode New Target Template

Click **Next**. While the default values for the target name and other options are usually fine, check that the information matches what's in line with the Organize app. Set the **Organization Identifier** to **com.yourcompany**. For instance the bundle identifier suggested may be different. Click **Finish** to let Xcode create the UI test target for you.

Take a look at the file navigator. Xcode has created a folder with two test classes for you.

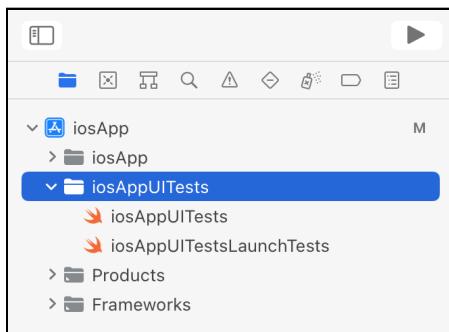


Fig. 8.10 – Xcode UI Test Target files

You can safely delete **iosAppUITestsLaunchTests.swift**. Open **iosAppUITests.swift** and delete all the contents of the class. You're going to write a couple of test functions here.

First, store an instance of the app as a property in the test class.

```
private let app = XCUIApplication()
```

Second, override the `setUp` function. The system calls this method before running each test. It's similar to when you tag a test function in Kotlin using `@BeforeTest`.

```
override func setUp() {
    continueAfterFailure = false
    app.launch()
}
```

This function will prevent the continuation of tests should any errors occur. Then launch the app, so you can run your tests.

Next, write a test to check the existence of the **About** button.

```
func testAboutButtonExistence() {
    XCTAssert(app.buttons["About"].exists)
}
```

The assert functions in Xcode test frameworks usually start with `XCTAssert`. This is the simplest one you could use, and it needs a Boolean parameter. Query all the buttons of the app and look for one with **About** title.

Note: Xcode test functions should start with `test....`, otherwise Xcode won't identify them as test functions, and so won't run them.

As with Android Studio, you can run the tests using the button in the code gutter. You can also choose the **Test** button from the **Product** menu or press **Command-U**.

You could improve this code a bit. Imagine you've localized your app in French. When you run the test above in French, the button title won't be **About**, so the test will fail. You can easily resolve this.

Go to **ContentView.swift** and attach the below modifier to the **Button** element:

```
.accessibilityIdentifier("aboutButton")
```

The **Button** element will now be as follows:

```
Button {
    shouldOpenAbout = true
} label: {
```

```
    Label("About", systemImage: "info.circle")
        .labelStyle(.titleAndIcon)
    }
    .accessibilityIdentifier("aboutButton")
    .popover(isPresented: $shouldOpenAbout) {
        AboutView()
        .frame(
            idealWidth: 350,
            idealHeight: 450
        )
    }
}
```

From now on, you can refer to this specific button using `aboutButton` regardless of what its title is.

Next, you can change the test body to this:

```
XCTAssert(app.buttons["aboutButton"].exists)
```

This is similar to the `semantics` modifier in Jetpack Compose. Run your test again to confirm nothing has changed in behavior and result.

Recording UI tests

Xcode has a cool feature that you can take advantage of to make the process of creating UI tests easier.

Create a new test function and put the cursor in the empty body.

```
func testOpeningAndClosingAboutPage() {
    // Put the cursor here
}
```

At the bottom of the page, a **Record** button would appear. Click on it. The app will run on the simulator, and Xcode will turn whatever action you do in the app into code.



Fig. 8.11 – Xcode UI Test Record button

Do these actions in order:

1. Tap the **About** button.
2. When the **About** page comes up, tap the **Done** button.
3. Stop recording using the same button with which you started recording.

Take a look at the test function. Xcode has added code for you. It will be something like this.

```
func testOpeningAndClosingAboutPage() {  
    let app = XCUIApplication()  
    app.navigationBars["Reminders"].buttons["aboutButton"].tap()  
    app.navigationBars["About Device"].buttons["Done"].tap()  
}
```

If that's all you had in mind, you're good to go! Otherwise, this gives you a starting point for writing your tests. You can also learn from this feature how to find elements on screen and act on them.

Another thing to take note of is that Xcode automatically goes for the `accessibilityIdentifier` if you'd set any. If not, it uses the static title to query elements. It's a great practice to always set this modifier on elements.

That said, you can take cues from Xcode's automatic test recording system and have this test function:

```
func testOpeningAndClosingAboutPage() {  
    //1  
    app.buttons["aboutButton"].tap()  
  
    //2  
    let aboutPageTitle = app.staticTexts["About Device"]  
    XCTAssertTrue(aboutPageTitle.exists)  
  
    //3  
    app.navigationBars["About Device"].buttons["Done"].tap()  
  
    //4  
    let remindersPageTitle = app.staticTexts["Reminders"]  
    XCTAssertTrue(remindersPageTitle.exists)  
}
```

1. Simulate tapping the **About** button when the app launches.
2. Check if there's a text on screen with **About Device** content. The **About** page has this title, and it's only there if that page is onscreen.

3. Find the **Done** button in one of the app's navigation bars with an **About Device** title and try tapping on it.
4. When you close the **About** page, the app should be in the **Reminders** page. Check for the page title if this is the case.

Run all the tests in **iosAppUITests** class by putting your cursor in the middle of its name and pressing **Command-U**.

Browse through the results in the Xcode console, or see the green checkmarks in the code gutter and the **Test Navigator** and rejoice!

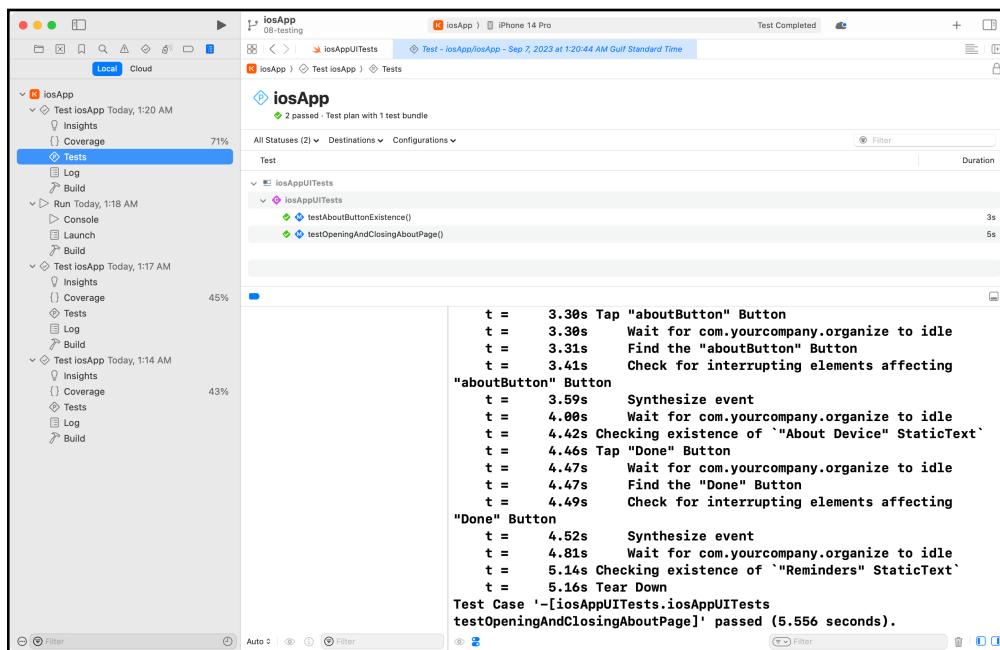


Fig. 8.12 – Xcode UI Test Success

Challenge

Here is a challenge for you to see if you've got the idea. The solution is inside the materials for this chapter.

Challenge: Writing Tests for RemindersRepository

Going one level deeper into the app's architectural monument, it's essential to have a bulletproof repository. After all, repositories are the backbone of the viewModels in **Organize**. Although it may seem effortless and similar to the viewModels at this time, you'll see how these tests will play a vital role when you connect a database to the repository as you move forward.

With this explanation in mind, try to create a test suite for `RemindersRepository`.

Key Points

- KMP will help you write less test code in the same way that it helped you write less business logic code.
- You can write tests for your common code as well as for platform-specific code – all in Kotlin.
- Declaring dependencies to a testing library for each platform is necessary. KMP will run your tests via the provided environment – such as JUnit on JVM.
- Using expect/actual mechanisms in test codes is possible.
- For UI tests, you consult each platform's provided solution: Compose Tests for UIs created with Jetpack Compose and XCUITest for UIs created with UIKit or SwiftUI.

Where to Go From Here?

This chapter barely scratched the surface of testing. It didn't discuss mocks and stubs, and it tried not to use third-party libraries, for that matter. There are a few libraries worth mentioning, though:

- Kotest (<https://kotest.io>): A multiplatform Kotlin testing library with extended assertions and support for property testing. It can generate values for edge cases and random values.
- Turbine (<https://github.com/cashapp/turbine>): A small multiplatform library geared toward testing Koltin Flows. This chapter didn't talk about Coroutines and Flows. However, if you ever wanted to test those, take a look at Turbine as `kotlinx-coroutines-test` library doesn't support Kotlin/Native yet.
- MockK (<https://mockk.io/>): This is the most famous library for mocking in Kotlin. Although there's a multiplatform version available, it lacks support for iOS.

If you are eager to learn more about testing in general, there are great resources out there, such as screencasts and articles, as well as these two books from kodeco.com:

- Android Test-Driven Development by Tutorials (<https://www.kodeco.com/books/android-test-driven-development-by-tutorials>)
- iOS Test-Driven Development by Tutorials (<https://www.kodeco.com/books/ios-test-driven-development-by-tutorials>)

Chapter 9: Dependency Injection

By Saeed Taheri

Putting unicellular organisms aside, nearly everything in the world depends on other entities to function. Whether it's something in nature or something mankind has created, it usually takes multiple things to create a working instance of anything.

Imagine an assembly line in a car factory. They don't create the engines and the wheels on the assembly line. Car manufacturers outsource many of the parts to other companies. In the end, they bring them all to the assembly line, *inject* each part into the making-in-progress and a shiny new car appears. The car is *dependent* on other objects. The same applies to the software world.

If you were to model the Car into a class, one of its dependencies would be the Engine. The car object shouldn't be responsible for creating the engine. You should inject the engine from outside into the assembly line — or in programming nomenclature, constructor, or initializer.

Advantages of Dependency Injection

Dependency injection, or **DI**, has many advantages.

- **Maintainability:** DI makes your code maintainable. If your classes are loosely coupled, you can catch bugs more easily and address a possible issue faster than you would with a convoluted class that doesn't adhere to the single-responsibility principle.
- **Reusability:** Going back to the car factory example, you're able to reuse the same model of wheels for many cars the factory manufactures. Loosely coupled code will let you reuse many parts of your code in different ways.
- **Ease of refactoring:** There may come a time in the lifetime of your app when you need to apply a change to your codebase. The less coupled your classes are, the easier the process will be. Imagine you needed to change the engine if you wanted to have new headlights!
- **Testability:** Everything comes back to the code being loosely coupled. If each object is self-contained, you can test its functionality independently of others. No one would like a car whose engine wouldn't work when a windshield wiper is broken! This way, each team responsible for each module will test their product and hand it over to other teams.
- **Ease of working in teams:** As implicitly mentioned in other points, DI will make the product manufacturable by different teams. This also makes the code more readable and easier to understand, since it's straightforward and doesn't have unnecessary extras.

Automated DI vs. Manual DI

Now that you're on the same page with those who favor using dependency injection in their apps, you need to actually provide the dependencies where needed.

Open the starter project in Android Studio. Next, open **RemindersViewModel.kt** from the **presentation** directory in **commonMain**. You'll take on the responsibility of creating the repository instance outside **RemindersViewModel**.

Remove the repository definition and pass it in via the constructor like this:

```
class RemindersViewModel(  
    private val repository: RemindersRepository  
) : BaseViewModel() { // ...  
}
```

Build the project by going to the **Build** menu and clicking **Make Project**. You'll immediately see there are compile issues in both **RemindersView.kt** files on Android and desktop. The same error is also there for **RemindersView.swift**, which Android Studio can't catch.

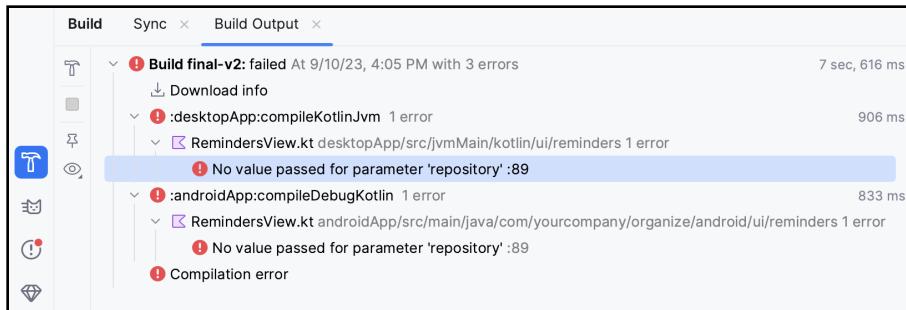


Fig. 9.1 – No value passed for repository

Note: One implementation which won't show up in the output above, but still needs to be updated, is the `viewModel` definition in **RemindersViewModelTest.kt**. Pass `RemindersRepository()` where you initialize an instance of `RemindersViewModel` class.

You'll have to go to each of these files and provide an instance of `RemindersRepository`. What if the `repository` has its own dependencies? And what if those dependencies have their dependencies as well? This is a rabbit hole you want to avoid getting into!

You can provide all the dependencies yourself and no one can prevent you from doing so. Off the record, iOS developers usually do all this and write all the boilerplates by themselves, since there's not a popular library or methodology that everyone agrees on.

However, in the Android world, some libraries solve this problem by automating the process of creating and providing dependencies. They fall into two categories:

- Static solutions that generate the dependency graph at compile time.
- Solutions that connect the dependencies at runtime.

The most famous library for the first category is **Hilt**. Google recommends Hilt as part of their app architecture suggestions.

The catch is that neither Hilt nor its biological parent Dagger is available for KMP. Thus, the approach you can take is to do manual DI or use the most famous library of the second category: **Koin**.

Many would call libraries like Koin — which resolve dependencies at runtime — **Service Locators**. Those who favor static DI libraries will seriously object if you call Koin a DI library. However, here you're free to call it whatever you like.

Setting Up Koin

Setting up Koin is similar to how you've set up other multiplatform libraries in the previous chapters – a shared part and some specific libraries to use for each platform.

Open **libs.versions.toml** inside the **gradle** directory in the root of your project and add the Koin version in the **[versions]** section. As of writing this chapter, the latest version of Koin is 3.4.3.

```
koin = "3.4.3"
```

Next, in the **[libraries]** section, add these entries:

```
koin-core = { group = "io.insert-koin", name = "koin-core",
version.ref = "koin" }
koin-test = { group = "io.insert-koin", name = "koin-test",
version.ref = "koin" }
koin-android = { group = "io.insert-koin", name = "koin-
android", version.ref = "koin" }
koin-androidx-compose = { group = "io.insert-koin", name =
"koin-androidx-compose", version = "3.4.6" }
```

Thereafter, open **build.gradle.kts** for the **shared** module.

Add a dependency for **commonMain** source set as follows:

```
implementation(libs.koin.core)
```

While you're here, add a test dependency to **commonTest** as well. You're going to need it later in the chapter.

```
implementation(libs.koin.test)
```

Next, open **build.gradle.kts** for the **androidApp** and add these two dependencies. The second one is necessary because the app is using Jetpack Compose.

```
implementation(libs.koin.android)
implementation(libs.koin.androidx.compose)
```

Last but not least, open **build.gradle.kts** for the **desktopApp** and add this dependency to **jvmMain**:

```
implementation(libs.koin.core)
```

Make sure to sync Gradle after adding all these dependencies.

Declaring Your App Dependencies for Koin

Koin uses a special Kotlin Domain Specific Language — or **DSL** — to let you describe your application and its dependency graph.

There are three steps to start using Koin:

1. **Declare your modules:** Modules are entities that Koin later injects into different parts of your app as needed. You can have as many modules as you want.
2. **Start Koin:** A single call to `startKoin` function, passing in the modules in your app, will make a Koin instance ready to do the injection job in your app.
3. **Perform the injection:** Using some special keywords provided by Koin lets you inject object instances at will.

Inside the **shared** module, in the **commonMain** directory, create a sibling file to **Platform.kt** named **KoinCommon.kt**. You're going to write Koin setup codes there.

First, create an object in which you can hold a reference to the modules.

```
package com.yourcompany.organize

object Modules {
    val repositories = module {
        factory { RemindersRepository() }
    }
}
```

Define a module using the `module` block. A `factory` is a definition that will give you a new instance each time you ask for this object type. If you want to have a single instance or a singleton across the lifetime of your app, use the `single` keyword. It's most suited for things like databases and network managers.

Second, add a constant for the ViewModel's module inside the `Modules` object.

```
val viewModels = module {
    factory { RemindersViewModel(get()) }
}
```

The new kid in town is the `get()` function. It's a generic function that will resolve a component dependency. When you use this function, Koin looks up the declaration you provided and finds a matching call. As you remember, `RemindersViewModel` needs an instance of a `RemindersRepository` in its constructor, and you just defined it as a module.

So, Koin is good to go! Bear in mind that Koin resolves this dependency at runtime. Hence, if you use `get()` without a matching declaration, your app will most likely crash.

Finally, create a global function below `Modules`, which you'll call from each platform.

```
fun initKoin(
    appModule: Module = module { },
    repositoriesModule: Module = Modules.repositories,
    viewModelsModule: Module = Modules.viewModels,
): KoinApplication = startKoin {
    modules(
        appModule,
        repositoriesModule,
        viewModelsModule
    )
}
```

This function takes three parameters.

The first one is the `appModule`. You can use this in later chapters for injecting app-level dependencies. Since those dependencies come from each platform, you're making the ability to pass them from outside.

The second and third parameters are for repositories and viewModels with default values. You'll see later on that you need to pass those in certain scenarios.

The return type of `initKoin` is an instance of `KoinApplication`. You get an instance of this type by calling `startKoin`, passing in all the modules you defined. This is the Koin starting point and the glue that keeps everything together.

Import all the missing dependencies as follows:

```
import com.yourcompany.organize.data.RemindersRepository
import com.yourcompany.organize.presentation.RemindersViewModel
import org.koin.core.KoinApplication
import org.koin.core.context.startKoin
import org.koin.core.module.Module
import org.koin.dsl.module
```

Using Koin on Each Platform

Now that you've got all the pieces, it's time to use Koin for real.

Android

Open `OrganizeApp.kt` in the `androidApp` module. Add the `onCreate` function below inside the class and start Koin there.

```
override fun onCreate() {
    super.onCreate()

    initKoin(
        viewModelsModule = module {
            viewModel {
                RemindersViewModel(get())
            }
        }
    )
}
```

Call the `initKoin` method you defined earlier. You're using the `viewModel` block, which comes from `org.koin.androidx.viewmodel.dsl.viewModel` package, to declare an Android `viewModel`. The difference between Android `viewModels` and others is that they will live through the Android configuration changes, such as device rotation. This needs a special kind of initialization, which Koin for Android does for you.

Open `RemindersView.kt` in the `androidApp` module and replace the `RemindersView` function definition as follows:

```
@Composable
fun RemindersView(
    viewModel: RemindersViewModel = getViewModel(),
    onAboutButtonClick: () -> Unit,
) {
    // ...
}
```

Calling the `getViewModel()` extension function, which Koin provides, does all the creation and injection process.

Build and run the Android app. The app should behave as you're familiar with — this time with DI, though.



Fig. 9.2 — The Android app after integrating Koin.

iOS

Koin is a Kotlin library. Lots of bridging occurs, should you want to use it with Swift and Objective-C files and classes. To make things easier, you'd better create some helper classes and functions.

Create **KoinIOS.kt** inside the **iosMain** directory as a sibling to **Platform.kt**.

Create a function inside an object for initializing Koin on iOS. Swift doesn't bridge Kotlin functions with default parameters. This function is to compensate for that limitation.

```
package com.yourcompany.organize

object KoinIOS {
    fun initialize(): KoinApplication = initKoin()
}
```

Next, create an extension function on Koin for getting instances of a specific Objective-C class. Remember that extension functions need to be in the top-level file. So make sure this function is outside the KoinIOS object you just created. Unfortunately, there's no easy way to write them as generic functions, and some type-casting will be necessary at the call site.

```
@kotlinx.cinterop.BetaInteropApi
fun Koin.get(objCClass: ObjCCClass): Any {
    val kClazz = getOriginalKotlinClass(objCClass)!!
    return get(kClazz, null, null)
}
```

Here, you're passing `null` for `qualifier` and `parameter`. If you find yourself in need of passing parameters when asking for a dependency, you could add this extension function as well:

```
@kotlinx.cinterop.BetaInteropApi
fun Koin.get(objCClass: ObjCCClass, qualifier: Qualifier?,
            parameter: Any): Any {
    val kClazz = getOriginalKotlinClass(objCClass)!!
    return get(kClazz, qualifier) { parametersOf(parameter) }
}
```

As some types and functions you used in these extensions are in beta, you need to opt in by annotating the functions with `@kotlinx.cinterop.BetaInteropApi`.

Next, open Xcode, create **Koin.swift** inside the **Supporting Files** directory and write the class as follows:

```
import Shared

final class Koin {
    //1
    private var core: Koin_coreKoin?

    //2
    static let instance = Koin()

    //3
    static func start() {
        if instance.core == nil {
            let app = KoinIOS.shared.initialize()
            instance.core = app.koin
        }
        if instance.core == nil {
            fatalError("Can't initialize Koin.")
        }
    }

    //4
    private init() {}

    //5
    func get<T: AnyObject>() -> T {
        guard let core else {
            fatalError("You should call `start()` before using `"
                + "#function`")
        }

        guard let result = core.get(objCClass: T.self) as? T else {
            fatalError("Koin can't provide an instance of type: `"
                + "(T.self)`")
        }

        return result
    }
}
```

1. Store a reference to the Koin core type. This will make it possible to ask for objects.
2. Create a static property for the newly created class to use it as a singleton.
3. Call this function when the app starts. Here, you're calling into Kotlin to initialize Koin. `KoinIOS.shared` is the way Kotlin exposes the object you created earlier. If for any reason this procedure fails, you'll make the app crash.

4. Mark the initializer for this class as private. This will prevent people from accidentally initializing the Swift Koin class apart from the way you intended.
5. This method uses the get extension methods you wrote on Koin in Kotlin. It first checks if `core` isn't `nil`. Then it tries casting from `Any` to the generic type `T`. This will make this function type-safe at the call site.

Two steps remain for using Koin on iOS.

First, open **iOSApp.swift** and start Koin at initialization time.

```
@main
struct iOSApp: App {
    init() {
        Koin.start()
    }
    // ...
}
```

Finally, open **RemindersViewModelWrapper.swift** and initialize `viewModel` as follows:

```
let viewModel: RemindersViewModel = Koin.instance.get()
```

Build and run. The app will work as before.



Fig. 9.3 – The iOS app after integrating Koin.

Desktop

This is the easiest of all platforms. First, open **Main.kt** and add a reference to the **Koin** object. Initialize it in the **main** function.

```
lateinit var koin: Koin
private set

fun main() {
    koin = initKoin().koin

    return application { // ...
    //
```

Next, open **RemindersView.kt** in the **desktopApp** module, and change the **RemindersView** composable function definition as follows:

```
@Composable
fun RemindersView(
    viewModel: RemindersViewModel = koin.get(),
    onAboutButtonClick: () -> Unit,
) {
    // ...
}
```

Use the **koin** instance you created and take advantage of the **get()** function.

Build and run — enjoy!

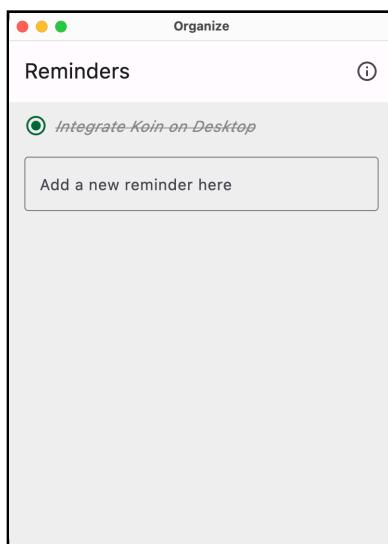


Fig. 9.4 – The Desktop app after integrating Koin.

Updating AboutViewModel

You're now familiar with the process, it's time to update `AboutViewModel` to use DI. Put the book down and see if you can do it all by yourself.

Here's what you should do:

Open `AboutViewModel.kt` from the `commonMain/presentation` directory and move the `platform` definition to the constructor as follows:

```
class AboutViewModel(  
    platform: Platform  
) : BaseViewModel() {  
    // ...  
}
```

Next, open `KoinCommon.kt` in the `commonMain` directory and add another factory block to the `viewModels` module.

```
val viewModels = module {  
    factory { RemindersViewModel(get()) }  
    factory { AboutViewModel(get()) }  
}
```

Declare to Koin how to resolve the dependency of the `AboutViewModel` class, which is of type `Platform`. Create a constant named `core` inside the `Modules` object to hold this and some other dependencies, which will come in a later chapter.

```
object Modules {  
    val core = module {  
        factory { Platform() }  
    }  
    // ...  
}
```

Last but not least in this file, update the definition of `initKoin` to accept the new modules you defined:

```
fun initKoin(  
    appModule: Module = module { },  
    coreModule: Module = Modules.core,  
    repositoriesModule: Module = Modules.repositories,  
    viewModelsModule: Module = Modules.viewModels,  
) : KoinApplication = startKoin {  
    modules(  
        appModule,  
        coreModule,
```

```
        repositoriesModule,  
        viewModelsModule,  
    )  
}
```

Next, to follow the usual approach, you'll update the codes for the platforms in order.

Android

Open **AboutView.kt** in the **androidApp** module and change the **AboutView** composable definition to this:

```
fun AboutView(  
    viewModel: AboutViewModel = getViewModel(),  
    onUpButtonClick: () -> Unit  
) {  
    // ...  
}
```

You're once again using the `getViewModel()` method from the Koin Android library.

Finally, open **OrganizeApp.kt** for the Android app and declare the module for **AboutViewModel** as well:

```
initKoin(  
    viewModelsModule = module {  
        viewModel {  
            RemindersViewModel(get())  
        }  
        viewModel {  
            AboutViewModel(get())  
        }  
    }  
)
```

Build and run the app to make sure everything is still working as expected.

iOS

It's pretty straightforward. Open **AboutView.swift** and change the definition of `viewModel` as follows:

```
@State private var viewModel: AboutViewModel =  
    Koin.instance.get()
```

And that's it! Build and run the iOS app. Open the About page to ensure DI is working correctly.

Desktop

This is also a piece of cake. Open **AboutView.kt** in **desktopApp** module and change the **AboutView** composable function definition:

```
fun AboutView(  
    viewModel: AboutViewModel = koin.get()  
) {  
    ContentView(items = viewModel.items)  
}
```

That concludes integrating Koin in the viewModels of all three apps.

Build and run and confirm the app is behaving as before.

Testing

Because of the changes you made in this chapter, the tests you wrote in the previous chapter wouldn't compile anymore. Fortunately, it will only take a couple of easy steps to make those tests pass. You'll also learn a few more tricks for testing your code along the way.

Checking Koin Integration

As you already know, Koin resolves the dependency graph at runtime. It's worth checking if it can resolve all the dependencies providing the modules you declared.

In the **commonTest** directory, create a file called **DITest.kt** as a sibling to **PlatformTest.kt**.

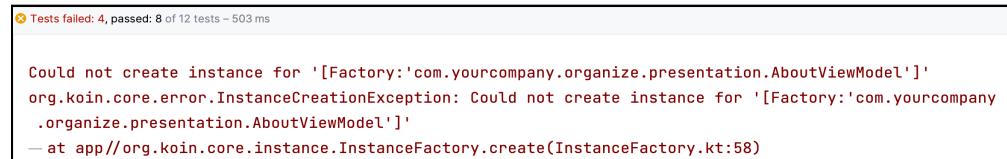
Create a class named **DITest**, and add this test function inside it:

```
package com.yourcompany.organize  
  
class DITest {  
    @Test  
    fun testAllModules() {  
        koinApplication {  
            modules(  
                Modules.viewModels,  
            )  
        }  
    }  
}
```

```
    }.checkModules()  
}
```

Here, you're creating an instance of `KoinApplication` and providing a list of modules. For now, add the `viewModels` module. `checkModules()` is a function that does the integration check you read about earlier. It verifies all definitions' dependencies, starts all modules and then checks if definitions can run.

Run the test and check out the result.



```
Tests failed: 4, passed: 8 of 12 tests – 503 ms  
  
Could not create instance for '[Factory:'com.yourcompany.organize.presentation.RemindersViewModel']'  
org.koin.core.error.InstanceCreationException: Could not create instance for '[Factory:'com.yourcompany  
.organize.presentation.RemindersViewModel']'  
— at app//org.koin.core.instance.InstanceFactory.create(InstanceFactory.kt:58)
```

Fig. 9.5 – Koin `checkModules` test failed

The test failed, and the reason is pretty obvious. By only providing the `viewModels` module, Koin can't create an instance of `RemindersViewModel` or `AboutViewModel`. To fix this, just add `Modules.repositories` and `Modules.core` to the list of modules in the test function above. Therefore, you'll be passing these modules:

```
modules(  
    Modules.core,  
    Modules.repositories,  
    Modules.viewModels,  
)
```

Run the test again, and it will pass successfully.

Note: The test only passes on desktop and iOS. It fails on Android. The reason for that lies in the creation process of `ScreenInfo`. If you take a look at the actual Android implementation of that class, you'll see that it calls `Resources.getSystem()`. When running tests, this call would fail since the Android system isn't available while unit testing. Resolving this issue needs mocking the `Resources` class, which is beyond the scope of this chapter. You saw a similar issue in the previous chapter as well.

A good citizen doesn't litter, and neither does a good developer when testing Koin. Whenever you create an instance of `KoinApplication` or call `startKoin`, make sure to stop it after you don't need it anymore. As you know, a good place to do so is to create a function with the `@AfterTest` annotation.

Add this function to `DITest` class, which uses the Koin-provided `stopKoin` method to do the cleanup.

```
@AfterTest
fun tearDown() {
    stopKoin()
}
```

Updating RemindersViewModelTest

Open `RemindersViewModelTest.kt`. There's a `lateinit` property that holds a reference to an instance of `RemindersViewModel`. In the `setup` method, you're initializing this property like this:

```
viewModel = RemindersViewModel(RemindersRepository())
```

Yuck! No one likes this anymore. It's better to summon Koin to do the job!

There are a few steps you need to take to integrate Koin into tests.

First, make `RemindersViewModelTest` extend `KoinTest`. This conformance will make the test class a `KoinComponent`. The `KoinComponent` interface is here to help you retrieve instances directly from Koin using some special keywords.

Next, change the `viewModel` property definition to this:

```
class RemindersViewModelTest: KoinTest {
    private val viewModel: RemindersViewModel by inject()

    //...
}
```

The `by` keyword in Kotlin delegates the implementation of the accessors for a property to another object. By using `by inject()`, Koin will lazily retrieve your instances from the dependency graph. The `inject()` function is an extension to `KoinTest`.

The last piece is to initialize Koin before the test and stop it after the test — pretty much like when you tested the Koin integrity.

Implement these two functions in the test class:

```
@BeforeTest
fun setup() {
    initKoin()
}
```

```
@AfterTest  
fun tearDown() {  
    stopKoin()  
}
```

In the `setup` method, you don't need to initialize the `viewModel` property yourself anymore. You just call the `initKoin` method, which you called at the application's launch. It provides the default values for the modules. The `tearDown` function is exactly the same as in the `DITest` class.

Run the tests for the `RemindersViewModelTest` class, and they will pass as they used to do.

Key Points

- Classes should respect the single-responsibility principle and shouldn't create their own dependencies.
- Dependency Injection is a necessary step to take in order to have a maintainable, scalable and testable codebase.
- You can inject dependencies into classes manually, or use a library to do all the boilerplate codes for you.
- Koin is a popular and declarative library for DI, and it supports Kotlin Multiplatform.
- You declare the dependencies as modules, and Koin resolves them at runtime when needed.
- When testing your code, you can take advantage of Koin to do the injections.
- You can test if Koin can resolve the dependencies at runtime using the `checkModules` method.
- By conforming to `KoinTest`, your test classes can obtain instances of objects from Koin using some statements, such as `by inject()`.

Where to Go From Here?

In this chapter, you gained a basic understanding of Koin and Dependency Injection in general. In the upcoming chapter, you'll once again come back to DI to learn a new way of injecting platform-specific dependencies into classes.

To learn more about Koin in particular, the best source may be the official documentation at <https://insert-koin.io/docs/reference/introduction>, which is rather concise and self-explanatory, while covering numerous scenarios.

As mentioned in the chapter, there are other DI libraries as well. However, either those don't work in Multiplatform scenarios, or they're not as popular as Koin.

If you aren't targeting Multiplatform, you can consult **Hilt**: <https://developer.android.com/training/dependency-injection/hilt-android> and **Dagger**: <https://dagger.dev> on Android. In the iOS world, there isn't a go-to library.

For testing in particular, Koin also provides you with a way to mock or stub different objects.

Chapter 10: Data Persistence

By Saeed Taheri

The big elephant in the room of the **Organize** app is that it doesn't remember anything you add into it. As soon as you close the app or stop the debugger, every TODO item disappears for good.

The reason for this issue is that it's storing everything in memory and — surprisingly enough — computer memory, or to be more exact the RAM, may remind people of Dory the fish!

Apps can persist their data if they store them on non-volatile storage. Examples of this type of storage are **HDD**, or Hard Disk Drive, **SSD**, or Solid-State Storage, and Flash Storage.

Putting aside the details of how computers work and going more high level, you can mostly persist data using three different mechanisms:

1. Key-Value Storage
2. Database
3. File system

In this chapter, you'll learn about the first two options, which are more structured and more straightforward than working with file systems directly.

Key-Value Storage

One of the most common use cases when persisting data is to store bits of information in a dictionary or map style.

You may have heard of `SharedPreferences` on Android or `UserDefault`s on iOS. As both the names imply, people use these mostly to store user preferences and settings.

Since the setup process for using each of these classes is platform-specific, you could use the old and sweet **expect/actual** mechanism to create a single interface for accessing key-value storage on each platform. Although you completely know how to do this manually, it's a lot of boilerplate code to write.

Fortunately, there's a library named **Multiplatform Settings** (<https://github.com/russhwolf/multiplatform-settings#multiplatform-settings>) that does most of the heavy lifting for you.

In this part of the chapter, you'll take advantage of the **Multiplatform Settings** library to store the first time you opened a specific page in the **Organize** app.

Setting Up Multiplatform Settings

There are two ways of setting up the library.

One way is passing in instances of each platform's storage mechanism — such as `SharedPreferences` for Android, `UserDefault`s for iOS, or a similar implementation for JVM or desktop. This way, you could customize the instances at your will. For example, you could use an instance of `SharedPreferences` besides the one created using `PreferenceManager.getDefaultSharedPreferences()` or pass in a container for iOS apart from `UserDefault`s.`.standard`.

The other way is using the *no-arg* module. By using this, you'll have a faster and easier setup at the expense of customizability. For educational purposes, you'll use the long way here.

Open `build.gradle.kts` in the **shared** module and add this dependency for the **commonMain** source set:

```
implementation(libs.multiplatform.settings)
```

The entry for this library is already available in `libs.versions.toml` of the project.

Next, open `build.gradle.kts` of the `androidApp` module and add the same line to its dependencies as well.

Make sure to sync Gradle.

Then, open `KoinCommon.kt` from `commonMain` within the `shared` module and add an `expect` constant for platform modules to the file:

```
expect val platformModule: Module
```

As it's now second nature, you should provide the actual implementation for this module on all platforms. However, before doing that, make sure to pass this module when starting Koin in `initKoin` function.

```
startKoin {
    modules(
        appModule,
        coreModule,
        repositoriesModule,
        viewModelsModule,
        platformModule, // Don't forget to add this module
    )
}
```

Android

Still in the `shared` module, create `KoinAndroid.kt` inside `androidMain` as a sibling to `Platform.kt` and add this block of code:

```
actual val platformModule = module {
    single<Settings> {
        SharedPreferencesSettings(get())
    }
}
```

Make sure to import the `Settings` from the `com.russhwolf.settings.Settings` package while adding the needed imports.

`SharedPreferencesSettings` is the Android implementation of `Settings`, which uses `SharedPreferences` as its internal key-value store.

You're using Koin's `single` keyword, so it provides this dependency as a singleton.

SharedPreferencesSettings needs an instance of SharedPreferences in its constructor. You're asking Koin to fetch that dependency at runtime. Don't worry! To prevent a crash, you'll provide that dependency soon.

Open **OrganizeApp.kt** in **androidApp** module. As you may remember, the **initKoin** function had a parameter named **appModule**. Now it's time to use it.

Pass this block of code to the **appModule** parameter:

```
module {
    //1
    single<Context> { this@OrganizeApp }

    //2
    single<SharedPreferences> {
        get<Context>().getSharedPreferences(
            "OrganizeApp",
            Context.MODE_PRIVATE
        )
    }
}
```

Import the requested libraries. Here's what's going on in the code above:

1. Setting up SharedPreferences requires an instance of the Android Context. You're declaring to Koin that it can use the Application instance as the singleton Context.
2. You use the `get()` function to get an instance of Context and create a private SharedPreferences named **OrganizeApp**.

iOS

Open **KoinIOS.kt** from **iosMain** inside the **shared** module and add the actual implementation of **platformModule** constant:

```
actual val platformModule = module { }
```

An empty module will silence the compiler.

You initialized Koin on iOS through the `initialize` method on `KoinIOS` object. You can add a parameter to that method, so you can inject an instance of `UserDefaults` — or, in Objective-C nomenclature, `NSUserDefaults`.

Make the following changes to the `initialize` function:

```
fun initialize(  
    userDefaults: UserDefaults,  
) : KoinApplication = initKoin(  
    appModule = module {  
        single<Settings> {  
            UserDefaultsSettings(userDefaults)  
        }  
    }  
)
```

As always don't forget to import the requested libraries. This is similar to the Android counterpart, but you're using `UserDefaultsSettings`, which is an implementation of `Settings` on Apple platforms. `UserDefaultsSettings` needs an instance of `UserDefaults` in its constructor.

Next, open the starter project in Xcode and go to **Koin.swift**. Inside the `Koin` class, change the line in `start` where you initialized `KoinIOS` to account for the changes you made to `initialize`:

```
let app = KoinIOS.shared.initialize(  
    userDefaults: UserDefaults.standard  
)
```

Desktop

Create **KoinDesktop.kt** inside the `desktopMain` directory as a sibling to `Platform.kt` and add the actual implementation for `platformModule`.

```
actual val platformModule = module {  
    //1  
    single {  
        Preferences.userRoot()  
    }  
  
    //2  
    single<Settings> {  
        PreferencesSettings(get())  
    }  
}
```

Here's what's happening in this code:

1. As you turned over to JVM when constructing the `Platform` class in earlier chapters, you need to do the same here as well. JVM has a `Preferences` class, and you can take advantage of it for storing key-value pairs. There are two predefined containers for `Preferences`: one for user values and one for system values. You need to use the `userRoot`.
2. Having an instance of JVM's `Preferences` object, you can declare your need for `Settings` instance to Koin and instruct it to use `PreferencesSettings` to create one.

Finally, add all the requested imports. There's nothing else to do for the desktop app.

Build and run all the apps to make sure there aren't any compile-time or runtime issues.

Storing Values Using Multiplatform Settings

In this part, you'll store the first time you open the **About Device** page.

Open `AboutViewModel.kt` and add a constructor parameter of type `Settings` to `AboutViewModel`'s definition.

```
class AboutViewModel(  
    platform: Platform,  
    settings: Settings,  
) : BaseViewModel() {  
    // ...  
}
```

Add a property to store the formatted timestamp of the first time this page is opened:

```
val firstOpening: String
```

Next, add the `init` block to initialize this property as follows:

```
init {  
    //1  
    val timestampKey = "FIRST_OPENING_TIMESTAMP"  
  
    //2  
    val savedValue = settings.getLongOrNull(timestampKey)  
  
    //3
```

```

firstOpening = if (savedValue == null) {
    val time = Clock.System.now().epochSeconds - 1
    settings.putLong(timestampKey, time)

    DateFormatter.formatEpoch(time)
} else {
    DateFormatter.formatEpoch(savedValue)
}
}

```

Here's the explanation of the above code:

1. This is the key with which you'll store the timestamp in `settings`.
2. You fetch the `Long` value using the key.
3. If the fetched value is `null`, you get the current time using the `Clock` object in the **kotlinx-datetime** library and store it in epoch second format (time value measured in seconds since the Unix Epoch) in `settings`. If the value isn't `null`, you use the `savedValue`. In either case, you format the saved date and store the user-facing string in the property. The `DateFormatter` object is already available for you in this chapter's materials.

Now it's time to show this value in the UI.

Android

First, open **OrganizeApp.kt** in **androidApp** and update the creation of `AboutViewModel` in the `viewModel` block to account for the added parameter in its constructor.

```

viewModel {
    AboutViewModel(get(), get())
}

```

Do the same for its factory in **KoinCommon.kt** in the **shared** module:

```

factory { AboutViewModel(get(), get()) }

```

Next, open **AboutView.kt** in the **androidApp** module. Change the `ContentView` composable function to accept a footer and then show it at the bottom of row items:

```

@Composable
private fun ContentView(
    items: List<AboutViewModel.RowItem>,
    footer: String?,
) {
}

```

```
LazyColumn(  
    modifier = Modifier  
        .fillMaxSize()  
        .semantics { contentDescription = "aboutView" },  
) {  
    items(items) { row ->  
        RowView(title = row.title, subtitle = row.subtitle)  
    }  
    footer?.let {  
        item {  
            Text(  
                text = it,  
                style = MaterialTheme.typography.labelSmall,  
                textAlign = TextAlign.Center,  
                modifier = Modifier  
                    .fillMaxWidth()  
                    .padding(8.dp),  
            )  
        }  
    }  
}
```

This code is pretty straightforward. It looks rather long, but it's only adding a new row to the `LazyColumn` to display the timestamp.

Finally, update the `AboutView` composable function where you used `ContentView`.

```
@Composable  
fun AboutView(  
    viewModel: AboutViewModel = getViewModel(),  
    onUpButtonClick: () -> Unit  
) {  
    Column {  
        Toolbar(onUpButtonClick = onUpButtonClick)  
        ContentView(  
            items = viewModel.items,  
            footer = "This page was first opened:\n$  
{viewModel.firstOpening}"  
        )  
    }  
}
```

Build and run. Open the **About Device** page by tapping on the **i** button.

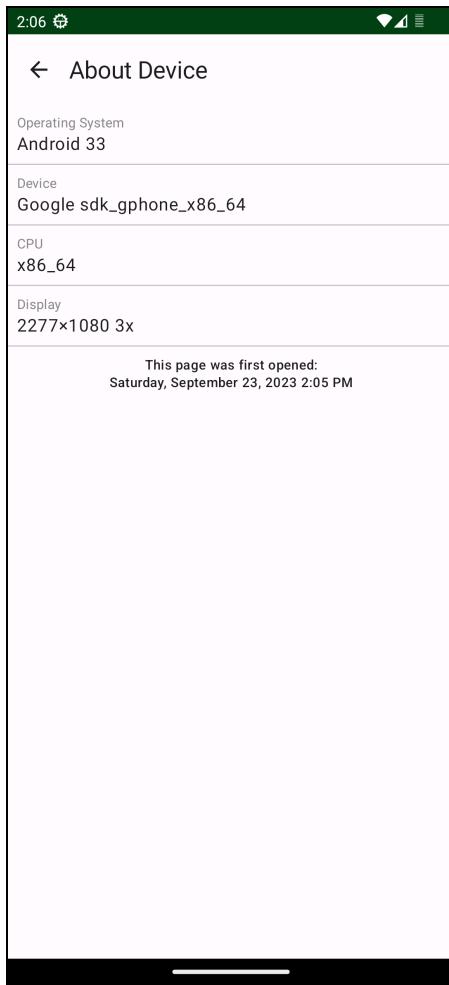


Fig. 10.1 – The About page on Android

You can see the timestamp of the first time the app was opened. Since this is the first time you're running the app with the new code, the timestamp corresponds to the time you opened the app in the current session.

iOS

Open **iosApp.xcodeproj** and go to **AboutListView.swift**. First, add a property for the footer as you did for the Android counterpart:

```
let footer: String
```

Next, update the body computed property to show the footer in the Section.

```
var body: some View {
    List {
        Section {
            ForEach(items, id: \.self) { item in
                // ...
            }
            footer: {
                Text(footer)
                    .font(.caption2)
            }
        }
    }
}
```

While you're in this file, update the `#Preview` macro to silence Xcode's woes.

```
#Preview {
    AboutListView(
        items: [
            AboutViewModel.RowItem(
                title: "Title",
                subtitle: "Subtitle"
            )
        ],
        footer: "Section Footer"
    )
}
```

Next, open **AboutView.swift** and update the usage of **AboutListView** to account for the footer.

```
AboutListView(
    items: viewModel.items,
    footer: "This page was first opened on \
(viewModel.firstOpening)"
)
```

Build and run the app. Open the **About Device** page by tapping on the **About** button in the bottom toolbar.

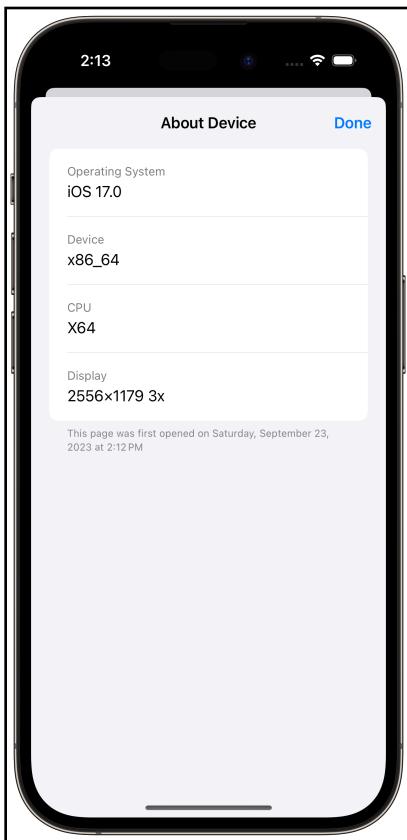


Fig. 10.2 – The About page on iOS

Desktop

Open **AboutView.kt** in the **desktopApp** module. Change the **ContentView** composable function to accept a footer, and then show it at the bottom of row items. It's the same definition of the **ContentView** in the **androidApp** module. You can look back at the implementation above.

Finally, update the **AboutView** composable function like this:

```
@Composable
fun AboutView(viewModel: AboutViewModel = koin.get()) {
    ContentView(
        items = viewModel.items,
        footer = "This page was first opened:\n$
```

```
{viewModel.firstOpening}"  
}  
}
```

Build and run, then check out the **About Device** window.

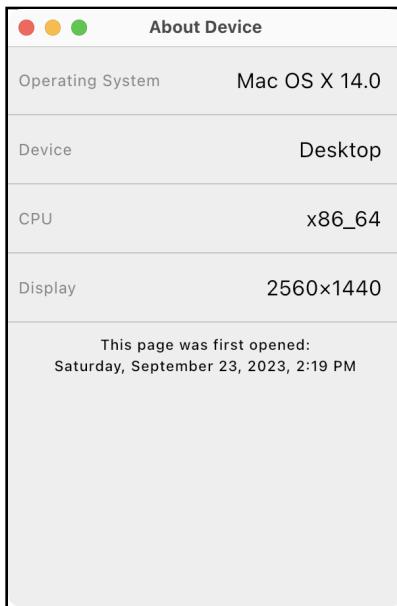


Fig. 10.3 – The About page on Desktop

That concludes integrating the **Multiplatform Settings** library into **Organize**.

Database

A database is an organized collection of data. Whenever you're dealing with a structured set of data that you need to access in a certain way, it's a good choice to use a database over directly messing with the file system.

Although databases have many kinds and models, relationally based ones such as **SQLite** have been the most popular among mobile developers.

For instance, **Core Data**, which is a framework for managing an object graph on iOS, uses SQLite as its persistent store.

Also in the Android world, Google's recommendation for a database has been **Room** for a while, and it's a typesafe wrapper over SQLite with numerous extra features.

Unfortunately, none of those two popular options are available for KMP. However, there's a great library named **SQLDelight**, which generates typesafe Kotlin APIs from your **SQL** statements and works with KMP with a fairly easy setup.

SQL

SQL is a database querying language, and you shouldn't mistake it for the database itself. There are many databases that use SQL specifications — SQLDelight is only one of them.

SQLDelight can work with SQLite, MySQL, or even PostgreSQL. However, the version that works in KMP uses SQLite under the hood.

You've already defined some actions in the **Organize** app, such as showing all reminders, creating a new reminder and marking reminders as done. You're going to define these actions using SQL so SQLDelight can understand them.

In the **shared** module, create nested directories as follows inside the **commonMain** directory — either in the file manager of the operating system or in Android Studio.

```
sqldelight/com/yourcompany/organize/db
```

Inside the **db** directory, which is short for database, create a file named **Table.sq**.

Note: sq is usually the file extension for SQLDelight. Android Studio may suggest installing a plugin for this matter. It'll help you with autocompletion when writing SQL statements. If it didn't recommend you install the plugin, you can manually search for it in the Plugins Marketplace of Android Studio.

Relational databases represent data in **Tables**. A table will help you structure your data in the way you want. Start by adding this block to the file you created:

```
CREATE TABLE ReminderDb (
    id TEXT NOT NULL PRIMARY KEY,
    title TEXT NOT NULL UNIQUE,
    isCompleted INTEGER NOT NULL DEFAULT 0
);
```

This snippet creates a table named `ReminderDb` with these **Columns**:

- `id`, the **Primary Key** of this table
- `title`
- `isCompleted`

You can consider columns to be like fields or properties in data classes.

As is clear from the code, you specify `TEXT` or `INTEGER` for the types and `NOT NULL` to specify non-nullability. The `DEFAULT` keyword will let you provide a default value for an entity. The `UNIQUE` keyword prevents you from adding a new item with the same title as an existing item. Finally, `PRIMARY KEY` keyword is used to uniquely identify each record in the table.

One thing to keep in mind is that in many variations of SQL-based databases — such as SQLite — a Boolean type doesn't exist, and you should represent that type in some other way. Here, you're using `INTEGER`.

After you define the table and the specifications of data you store in it, it's time to define actions you want to do on the data. Add the following in the same file:

```
selectAll:  
SELECT * FROM ReminderDb;
```

You're defining an action named `selectAll`, which runs the next line when you call it. It selects all items in the `ReminderDb` table you defined earlier. The asterisk means *all* in SQL.

Next, an action to add a reminder.

```
insertReminder:  
INSERT OR IGNORE INTO ReminderDb(id, title)  
VALUES (?,?);
```

This statement will let you insert a new item in `ReminderDb` table. The `IGNORE` keyword will make the database ignore values that cause any potential errors. For example, you defined the `title` to be unique, so the system will ignore a duplicated value should you try to insert one.

Question marks are placeholders, meaning that real values will be available later.

Last but not least, you need an action for marking a reminder as done.

```
updateIsCompleted:  
UPDATE ReminderDb SET isCompleted = ? WHERE id = ?;
```

Using the `WHERE` keyword, you can find an item in the table with a certain `id` and set its `isCompleted` field to a certain value.

Setting Up SQLDelight

You need to apply the SQLDelight Gradle plugin in your project.

First, open `build.gradle.kts` of the project and add a line in the `plugins` block:

```
id("app.cash.sqlDelight").version(sqlDelightVersion).apply(false)
```

Next, open `build.gradle.kts` of the `shared` module and apply the plugin in `plugins` block:

```
id("app.cash.sqlDelight")
```

To make it possible for the SQLDelight Gradle plugin to read the `Table.sq` file, you need to define the database. In the same file, add this block at the bottom:

```
sqlDelight {  
    databases {  
        create("OrganizeDb") {  
            packageName.set("com.yourcompany.organize")  
            schemaOutputDirectory.set(  
                file("src/commonMain/sqlDelight/com/yourcompany/  
organize/db"))  
        }  
    }  
}
```

The code above creates a database named `OrganizeDb`, sets the package name you will use in this database and sets a schema output directory — which is necessary for database migrations.

SQLDelight requires something called a **Driver** to run your statements. A driver is a glue between the database schema you defined and the platform specific needs. For example, it requires an instance of the `Context` object on Android.

You should add dependencies for drivers on all platforms in `build.gradle.kts` of the `shared` module.

```
val androidMain by getting {  
    dependencies {  
        implementation(libs.sqlDelight.driver.android)
```

```
// ...
}
// ...
```

```
val iosMain by creating {
    dependencies {
        implementation(libs.sqlDelight.driver.native)
    }
    // ...
}
```

```
val desktopMain by getting {
    dependencies {
        implementation(libs.sqlDelight.driver.sqlite)
        // ...
    }
    // ...
}
```

The modules and versions are already in the **libs.versions.toml**.

Make sure to sync Gradle.

Database Helper

To make executing database actions easier, it's a good practice to create a common interface that abstracts the database you're using. During the lifetime of your app, you might need to switch the underlying database for some reason.

If usages of SQLDelight or any other third-party library aren't scattered throughout your app, you can replace it in one single place, and you won't need to touch anywhere else.

In the **data** folder inside the **commonMain** directory, create the file **DatabaseHelper.kt** and define the class as follows:

```
class DatabaseHelper(
    sqlDriver: SqlDriver,
) {
```

It accepts an instance of **SqlDriver**, which you'll inject via Koin on each platform. As you read earlier, you'll need a driver to run SQL statements.

Next, create a property inside the `DatabaseHelper` class to hold a reference to the `OrganizeDb`. The Gradle plugin generates this class based on what you defined earlier in `build.gradle.kts`.

```
private val dbRef: OrganizeDb = OrganizeDb(sqlDriver)
```

If Android Studio fails to resolve `OrganizeDb`, try building the project once so the code generation happens. Once done, you can import the class.

Add a method to fetch all reminders from the database just below `dbRef` property:

```
fun fetchAllItems(): List<ReminderDb> =  
    dbRef.tableQueries  
        .selectAll()  
        .executeAsList()
```

Use the `tableQueries` property on `OrganizeApp`, which contains all SQL statements you defined. As you named one of your statements `selectAll`, you use the same naming. Then, call `executeAsList` to get the results in a list. Note here that we are using a model class `ReminderDb` here. This is autogenerated using along with `OrganizeDb` based on the table that you created just above.

Next, add a method to insert a new reminder into the database:

```
fun insertReminder(id: String, title: String) {  
    dbRef.tableQueries.insertReminder(id, title)  
}
```

Finally, add a method to update the `isCompleted` status of each reminder:

```
fun updateIsCompleted(id: String, isCompleted: Boolean) {  
    dbRef.tableQueries  
        .updateIsCompleted(isCompleted.toLong(), id)  
}
```

You will also add an extension function that returns the `isCompleted` status of each reminder as `Boolean`. It'll help you later. Add it outside the class:

```
fun ReminderDb.isCompleted() = this.isCompleted != 0L
```

Furthermore, add this extension function, so you can convert `Boolean` to `Long` easily:

```
internal fun Boolean.toLong(): Long = if (this) 1L else 0L
```

Using the Database in the App

You should inject an instance of the `DatabaseHelper` class you created to wherever you want to use the database. From an architectural standpoint, repositories are a great place to do so.

Open `RemindersRepository.kt` and change the class entirely as follows:

```
//1
class RemindersRepository(
    private val databaseHelper: DatabaseHelper
) {
//2
    val reminders: List<Reminder>
        get() = databaseHelper.fetchAllItems().map(ReminderDb::map)

//3
    fun createReminder(title: String) {
        databaseHelper.insertReminder(
            id = UUID().toString(),
            title = title,
        )
    }

//4
    fun markReminder(id: String, isCompleted: Boolean) {
        databaseHelper.updateIsCompleted(id, isCompleted)
    }
}
```

Here's what the following code does:

1. Add a constructor property of type `DatabaseHelper`. It will let you inject an instance later.
2. Next, make `reminders`, a computed property that reflects what's in the database. You use the `map` function to map instances of `ReminderDb` to `Reminder`. You'll write `ReminderDb::map` soon.
3. This method will call `insertReminder` of `DatabaseHelper`.
4. Like the previous method, this method is calling into `DatabaseHelper` to mark reminders as completed or vice versa.

By applying the changes above, you made the database the single source of truth for reminders. You don't need to store reminders in properties and sync properties manually.

At the end of this file outside the class, add the extension function for mapping from `ReminderDb` to `Reminder`.

```
fun ReminderDb.map() = Reminder(
    id = this.id,
    title = this.title,
    isCompleted = this.isCompleted(),
)
```

Since you changed the constructor signature of `RemindersRepository`, the next step to take is to update those initialization calls. You'll only need to do it once because you used Koin to do the creation process.

Open `KoinCommon.kt` and update the `repositories` property inside the `Modules` object:

```
val repositories = module {
    factory { RemindersRepository(get()) }
    factory { AboutViewModel(get(), get()) }
}
```

By adding a simple `get()` call, you can silence the errors of Android Studio. However, you shouldn't forget to provide an instance of `DatabaseHelper` through Koin.

Since the database is one of the core functionalities of the app, add a module to the `core` property inside the `Modules` object:

```
val core = module {
    factory { Platform() }
    factory { DatabaseHelper(get()) }
}
```

A single dependency remains to be declared — the `SqlDriver`, which `DatabaseHelper` needs. Since `SqlDriver` is platform-dependent, you can declare it inside the `platformModule` you already defined through the `expect/actual` mechanism.

Android

Open **KoinAndroid.kt** in **androidMain** and add a singleton definition underneath the **Settings** declaration as follows:

```
single<SqlDriver> {
    AndroidSqliteDriver(OrganizeDb.Schema, get(), "OrganizeDb")
}
```

Creating an instance of **AndroidSqliteDriver** requires at least a database scheme, which you'll get from the generated **OrganizeDb** class, and an instance of **Context**, which you get through Koin by taking advantage of the **get()** function. Optionally, you could specify a name.

Build and run the app, add a few reminders and check some of them off the list. Then kill the app and launch it again. Everything is there because it should have always been this way. :]

iOS

Open **KoinIOS.kt** and set this as the **platformModule** actual property:

```
actual val platformModule: Module = module {
    single<SqlDriver> {
        NativeSqliteDriver(OrganizeDb.Schema, "OrganizeDb")
    }
}
```

This time, you're using the native implementation of **SqlDriver**.

Add the following code to import **NativeSqliteDriver** if Android Studio fails to do so:

```
import app.cash.sqldelight.driver.native.NativeSqliteDriver
```

Build and run the app. Verify that the reminders are persisted across app sessions.

Desktop

Open **KoinDesktop.kt**, and add the **SqlDriver** module definition as follows:

```
single<SqlDriver> {
    val driver = JdbcSqliteDriver(JdbcSqliteDriver.IN_MEMORY)
    OrganizeDb.Schema.create(driver)
    driver
}
```

However, you'll see that it still doesn't persist data between launches. This is because of the `IN_MEMORY` flag you're passing in. You then create a schema using the driver. When you use the `create` function, it assumes there is no data available.

Fortunately, `JdbcSqliteDriver` has another constructor, which takes a path to the database in the form of `jdbc:sqlite:PATH`. The PATH can either be relative or absolute. You can use that initializer; however, you should pay attention to the `create` method. You should call it only once. If you try to call `create` on an existing database, the app will crash.

For you to see this in action, you can set up the driver as follows for the first launch:

```
single<SqlDriver> {  
    val driver = JdbcSqliteDriver("jdbc:sqlite:OrganizeDb.db")  
    OrganizeDb.Schema.create(driver)  
    driver  
}
```

You're creating a file named **OrganizeDb.db** inside the directory where the app's codes are. You then create the schema using the driver. After doing this, try running the app. It'll most likely crash. Don't worry and continue.

Next, remove the line where you create the schema and run the app again. This time, the app uses the database file and persists everything.

In a production app, you can write custom logic to handle this dance.

Keep this in mind — in-memory databases are a good choice when writing tests.

Migration

Imagine one day you decide to add a new feature to the app: setting due dates on each reminder. This means you need to update many things throughout your code. One of the most important parts is the database schema. Although delicate, it's pretty straightforward to do.

First, run the `generateCommonMainOrganizeDbSchema` task from the Gradle pane in Android Studio or in the command line. This will make a file called **1.db** and put it in the same folder where **Table.sq** exists.

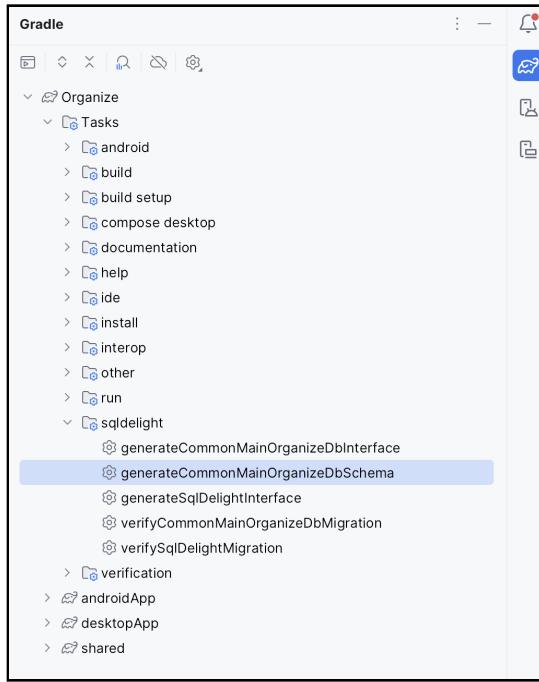


Fig. 10.4 – Generate Database Schema Gradle Task

Second, open **Table.sq** and update the table-creation statements, as well as any other actions you desire. This file should always reflect the current state of the database.

```
CREATE TABLE ReminderDb (
    id TEXT NOT NULL PRIMARY KEY,
    title TEXT NOT NULL UNIQUE,
    isCompleted INTEGER NOT NULL DEFAULT 0,
    dueDate INTEGER
);

setDueDate:
UPDATE ReminderDb SET dueDate = ? WHERE id = ?;
```

Here's the explanation of the above code:

1. You add a new column named `dueDate` to the table. It can be null, so you don't add the `NOT NULL` keyword. Since there's no `Date` type in SQLite, you'll store the timestamp as `INTEGER`.

2. Next, you write an update statement that will let you set a due date on a reminder.

Third, create a file called **1.sqm** in the same directory to write the migration statements. You must always name this file using this pattern: <version_to_upgrade_from>.sqm.

```
ALTER TABLE ReminderDb ADD COLUMN dueDate INTEGER;
```

You're telling the system to alter the `ReminderDb` table and add a new column for `dueDate`.

To check that the migration can happen without any errors, run `verifySqlDelightMigration` task from the Gradle pane in Android Studio or in the command line.

This will consider **1.db**, **1.sqm** and **Table.sq** to check the validity of the SQL statements you wrote.

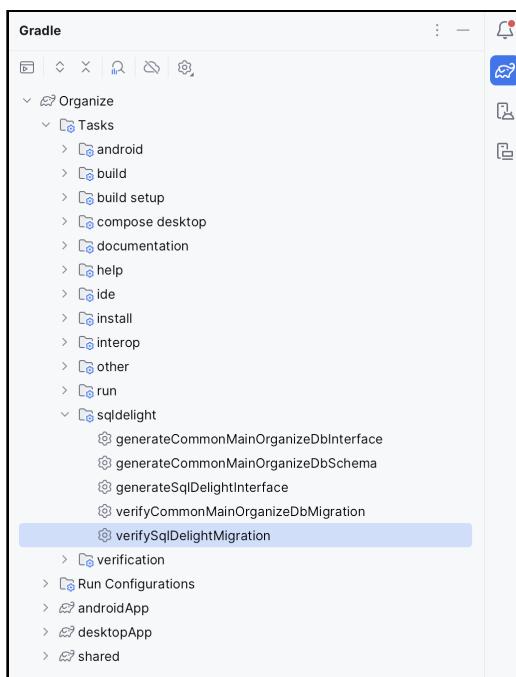


Fig. 10.5 – Verify SqlDelight Migration Gradle Task

If this test passed successfully, run the `generateCommonMainOrganizeDbSchema` task once again to save the current schema as **2.db**. You can safely check these schema files out into your git repository as well.

Build and run the app on all platforms to ensure it works everywhere. When building for desktop, you might get an error - `java.sql.SQLException: column 4 out of bounds [1,3] android after migration.`

To resolve the above error, open **KoinDesktop.kt** in **desktopMain** and update the **SqlDriver** singleton as follows:

```
single<SqlDriver> {
    val driver = JdbcSqliteDriver("jdbc:sqlite:OrganizeDb.db")
    OrganizeDb.Schema.migrate(driver, 1, 2)
    driver
}
```

Using `OrganizeDb.Schema.migrate`, you're migrating your schema from older version (1.db) to the newer version (2.db). Now, build and run the desktop app once again. It should run without any issues. After the first build, you can remove the `migrate` command for consecutive builds as you did with the `create` command earlier.

That's it. Now you know how to migrate your database.

This chapter doesn't help you with adding the UI for setting due dates on reminders. Set a due date for yourself to add due date support to Organize! :]

Adding Coroutines

Take a look at how you set up `RemindersViewModel`, and you'll remember that you needed to invoke the `onRemindersUpdated` lambda to notify users of the `ViewModel` of potential changes.

This gets the job done; however, you can achieve the same result as well as many additional features by using a more robust solution, such as a Kotlin Flow.

Kotlin Flow lets you observe streams of data. They're sequential and can emit individual values for an observer to process.

Kotlin Coroutines are the building blocks of Flows. You can't collect values out of a Flow without using Coroutines. In other words, you use Flows when you want to observe multiple asynchronously computed values. The `asynchronous` keyword in Kotlin will immediately bring up the `suspend` functions concept, for which you need to be acquainted with Coroutines to work on.

SQLDelight will let you consume a database query as a Flow. For this to work, you need to use some extension methods defined in the Coroutines Extensions library of SQLDelight. You should set up your app to work with Coroutines in the first place.

Multithreaded programming is difficult. Coroutines have come to simplify it for developers. However, working with Coroutines in an environment besides JVM, such as on iOS, has always been a hassle.

Recently, JetBrains has introduced a new memory model for Kotlin Native, which promises to simplify working with Coroutines on native platforms as well. You're going to get acquainted with that in the coming chapters.

Hence, for brevity, this chapter doesn't talk about using SQLDelight with Kotlin Flows.

Challenge

Databases have four basic operations: Create, Read, Update and Delete, a.k.a. **CRUD**. In Organize, you used three of those operations. Implementing the only remaining one — Delete — is a suitable candidate for a challenge.

Challenge: Adding Support for Deleting Reminders

Add a feature to Organize that lets the user delete reminders individually. For the UI part, you may take advantage of swipe gestures on Android and iOS. On desktop, you can use a context menu that's displayed when the user right-clicks on any reminder.

Key Points

- There are three major ways of persisting data on a device: Key-Value storage, database and working directly with the file system.
- Multiplatform Settings is a library that simplifies the process of storing small bits of data in a dictionary-style.
- You can use databases to store structured data and access them in a certain way.
- SQLDelight is a relationally based database that generates typesafe Kotlin API based on the SQL statements you write. When used in KMP, it uses SQLite under the hood.
- Migrating databases is a delicate and important step when you want to change your database schema.
- SQLDelight has an extension library that lets you observe database changes using Kotlin Flows.

Where to Go From Here?

This has been a long chapter. However, there remains lots of ground to cover.

Here are a couple of suggestions for you if you are eager to learn more:

- Getting acquainted with SQL will let you write more performant queries.
- Consulting the SQLDelight documentation, which is available here (<https://cashapp.github.io/sqldelight/>), will let you explore more of its features.
- Testing is an essential aspect of development. As mentioned earlier, you can take advantage of in-memory databases in your tests. Both Multiplatform Settings and SQLDelight offer testing artifacts which you can exploit.

Section III: Kotlin Multiplatform: Advanced

Networking is crucial to most modern apps, and it usually involves implementing similar logic using different frameworks and languages. Under the hood, it also involves concepts like serialization and concurrency. Fortunately, Kotlin Multiplatform has dedicated libraries for each of these.

In this section, you'll learn how to use serialization to decode JSON data to Kotlin objects. You'll then learn how to use a common networking library that leverages this common serialization to fetch data from the internet. To make the networking performant, you'll also learn about concurrency in Kotlin using coroutines and the considerations for different platforms. Finally, you'll learn how to extract an existing feature to a Kotlin Multiplatform library and also different ways of publishing this library.

Chapter 11: Serialization

By Carlos Mota

Great job on completing the first two sections of the book! Now that you're familiar with Kotlin Multiplatform, you have everything you need to tackle the challenges of this last section.

Here, you'll start making a new app named **learn**. It's built on top of the concepts you learned in the previous chapters, and it introduces a new set of concepts, too: serialization, networking and how to handle concurrency.

learn uses the Kodeco RSS feeds to show the latest tutorials for Android, iOS, Unity and Flutter. You can add them to a read-it-later list, share them with a friend, search for a specific key, or just browse through everything the team has released.

The Need for Serialization

Your application can send or receive data from a third party, either a remote server or another application in the device. Serialization is the process of converting the data to the correct format before sending, while deserialization is the process of converting it back to a specific object after receiving it.

There are different types of serialization formats — for instance, JSON or byte streams. You'll read more about this in Chapter 12, "Networking," when covering network requests.

Android uses this concept to share data across activities, services or receivers — either in the same application or to third-party apps. The difference is that instead of relying on `Serializable` to send data from custom types, the OS requires you to implement `Parcelable` to send these objects.

Project Overview

To follow along with the code examples throughout this section, download the **starter** project and open **11-serialization/projects/starter** with Android Studio.

Once the project synchronizes, you'll see a set of subfolders and other important files:

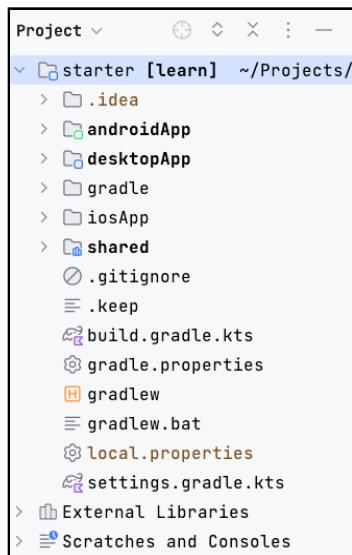


Fig. 11.1 — Project view hierarchy

Android App

The **androidApp** module follows the same structure that you're already used to from your Android apps, and you can use any library or component as you typically do:

- **components**: generic **Composable** functions that are used in different screens.
- **ui**: contains all the UI. The sub packages correspond to specific screens (**bookmark**, **home**, **latest** or **search**), the navigation bar (**main**), or the application design system (**theme**).
- **utils**: utility class to format a date.
- **KodecoApplication.kt**: the **Application** class that initializes the **Context** needed by **SQLDelight**.

Build and run the app.

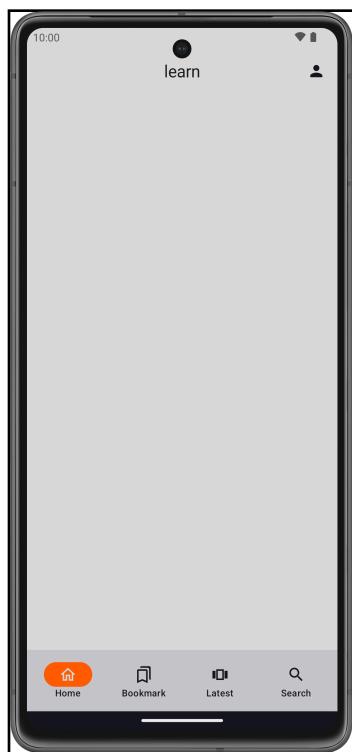


Fig. 11.2 – Android starter project running. Empty screen with no data.

This is your app skeleton. It doesn't look much, but that's because there's no data to show. You'll load the app data in the next sections.

iOS App

Your iOS app is inside the **iosApp** folder. It uses the **Swift Package Manager** to handle external libraries, which is available with Xcode.

Use Xcode or AppCode to open the file **iosApp.xcodeproj** located inside the **iosApp** folder.

Open the **iosApp** target, navigate to the **Build Phases** tab, and then select the **Compile Kotlin** dropdown. Here you have:

```
cd "$SRCROOT/.."  
./gradlew :shared:embedAndSignAppleFrameworkForXcode
```

This task compiles the **SharedKit** framework and adds it to your project when necessary.

Build and run the app. You'll see a screen like this:

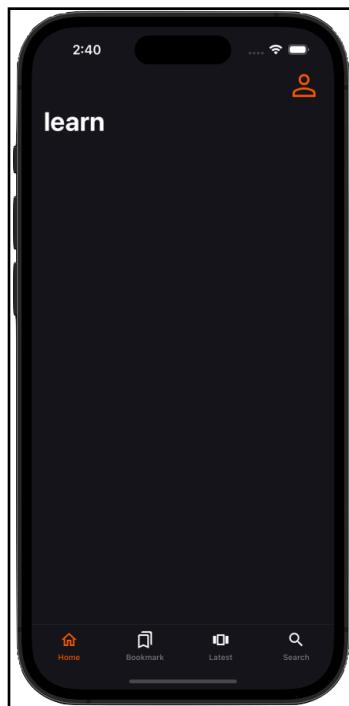


Fig. 11.3 – iOS starter project running. Empty screen with no data.

Desktop Application

The desktop application is similar to the Android app with just a couple of small changes — namely, on the libraries used that weren't available for the **JVM** target:

- **precompose** (<https://github.com/Tlaster/PreCompose>): A community library that lets you use **Jetpack Lifecycle**, **ViewModel**, **LiveData** and **Navigation** in a desktop application.

To run the desktop application, go to the command line and in the project root folder, enter:

```
./gradlew desktopApp:run
```

After it finishes, a new window will open with the app. You'll see a screen like this:

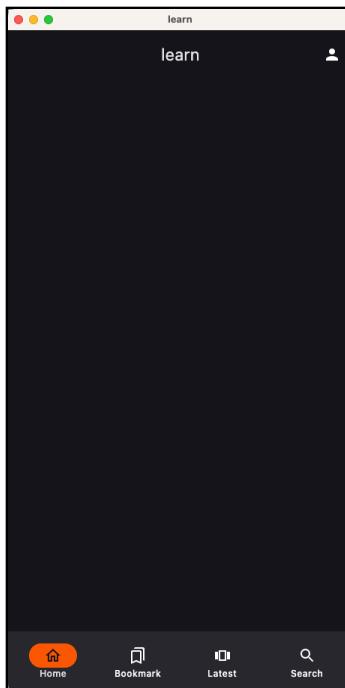


Fig. 11.4 – Desktop starter project running. Empty screen with no data.

Shared Module

This contains the entire business logic of **learn**. It's the multiplatform code that's shared across Android, iOS and desktop.

Common Code

When you open the shared module, you'll see two things inside **commonMain**:

- **kotlin**: Where the application business logic is.
- **sqldelight**: Contains the SQL file that's going to be used by **SQLDelight** to create the app's database, along with the corresponding CRUD methods to interact with it.

In the next section, you'll see what **learn** will look like after you implement all the required features.

Application Features

Before starting to write code, have a look first at the app concept and its features:

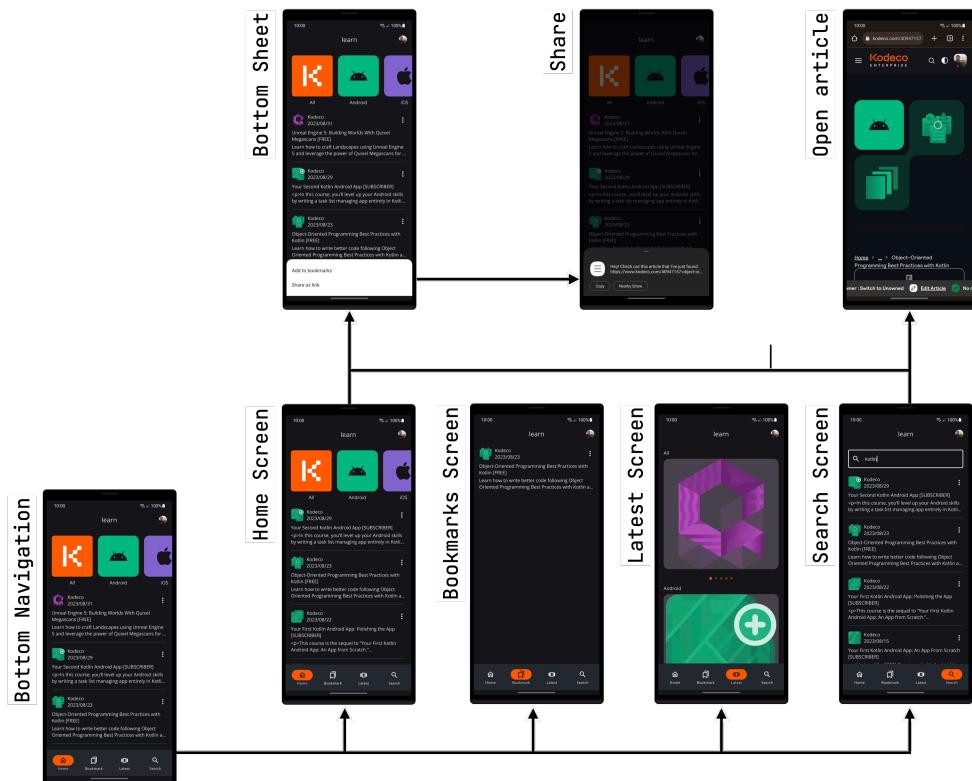


Fig. 11.5 – Application screens overview and navigation.

learn has four different screens that you can navigate to from the app's bottom bar:

Home

This is the app's default screen. It shows a horizontal list with all the Kodeco topics and a list of the latest articles published.

Clicking any topic redirects the user to the **latest** screen where they can see the most recent articles written, read or shared, then add them to the bookmarks list or remove them once they're done.

To open an article, click on the card and you'll be automatically redirected to the browser. If you click on the three dots inside the card instead, the app will show a bottom sheet. From there, you can add it to your **bookmarks**, or send it to a friend.

Bookmarks

This screen shows all the articles that you've saved. Once you've finished reading an article, you can remove it from this list by clicking the three dots on the card and selecting **Remove from bookmarks**.

Latest

This features a more graphical interface with the latest articles' sections and covers.

Search

Here, you can filter by a specific keyword and finally find that article you've been looking for.

Now that you're familiar with the app and the project, it's time to start learning how to implement these features.

Adding Serialization to Your Gradle Configuration

Kotlin doesn't support serialization and deserialization directly at the language level. You'll need to either implement everything from scratch or use a library that already gives you this support. Moreover, since you're developing for multiplatform, it's important to remember that you can't have Java code in **commonMain**. Otherwise, the project won't compile. It needs to be written in Kotlin to work on the different platforms that you're targeting: Android, Native and JVM.

So, do we need to implement serialization from scratch then? No, this is already available on **kotlinx.serialization**, a library created and maintained by JetBrains.

Time to import this library into the app. Open Android Studio and wait for the project to finish synchronizing. **learn** is using Gradle version catalogs to add and maintain its dependencies. Open the **libs.versions.toml** file located in the **gradle** folder and inside the [versions] section define the library version that you're going to use:

```
kotlinx-serialization-json = "1.6.0"
```

To use **kotlinx.serialization** you need to define the plugin and the library that's going to be added to the **build.gradle.kts** files. To add the first one, scroll down to [plugins] and add:

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

The **version.ref** attribute indicates the version that the app should use, which in this case is the same as **kotlin**.

Locate the [libraries] section and add:

```
kotlinx-serialization-json = { module =
    "org.jetbrains.kotlinx:kotlinx-serialization-json", version.ref
    = "kotlinx-serialization-json" }
```

Click **Sync Now** to start using these libraries and plugins.

Open the **build.gradle.kts** file located in the in root folder and in the **plugins** section add:

```
alias(libs.plugins.jetbrains.kotlin.serialization) apply false
```

Now, open the **build.gradle.kts** file inside **shared** and load the serialization plugin by adding the following code inside the **plugins** block:

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

Click **Sync Now** and wait for this process to finish.

There are four different **build.gradle.kts** files in the project:

- **build.gradle.kts**: This is located in the project root folder and configures the Android app, desktop app, and the shared module. It contains the repositories from which dependencies will be downloaded.
- **shared/build.gradle.kts**: This is the shared module configuration file. It contains the plugins and libraries that you're going to use, as well as the platforms you're going to target.
- **androidApp/build.gradle.kts**: The configuration file for the Android app. Defines a set of parameters used to compile the project, namely the SDK version, dependencies and compilations flags.
- **desktopApp/build.gradle.kts**: This is the configuration file for the desktop application. It defines the same configuration as the Android **build.gradle.kts** file.

Different Serialization Formats

kotlinx.serialization supports a set of serialization formats outside the box:

- **JSON** (<https://github.com/Kotlin/kotlinx.serialization/blob/master/formats/README.md#json>): A lightweight human-readable data-interchange format. **learn** uses JSON to load the Kodeco RSS feed links from a local file and later to deserialize the data received from the server (**kotlinx-serialization-json**).
- **JSON-Okio** (<https://github.com/Kotlin/kotlinx.serialization/blob/master/formats/README.md#json-okio>): A set of extensions for JSON that allow to integrate with Okio.
- **Protocol buffers** (<https://github.com/Kotlin/kotlinx.serialization/blob/master/formats/README.md#protobuf>): A cross-platform mechanism for serialized structured data (**kotlinx-serialization-protobuf**).
- **CBOR** (<https://github.com/Kotlin/kotlinx.serialization/blob/master/formats/README.md#cbor>): A binary data serialization format (**kotlinx-serialization-cbor**).

- **Properties** (<https://github.com/Kotlin/kotlinx.serialization/blob/master/formats/README.md#properties>): A key-value file that saves the configuration parameters of an application used in Java-related technologies (**kotlinx-serialization-properties**).
- **HOCON** (<https://github.com/lightbend/config/blob/master/HOCON.md>): A superset of JSON that's more human-readable and typically used in configuration files (**kotlinx-serialization-hocon**).

Note: There are a set of community-maintained libraries that support additional formats (<https://github.com/Kotlin/kotlinx.serialization/blob/master/formats/README.md>).

Except for **kotlinx-serialization-json**, all of these libraries are still experimental. Although they all seem robust and are used in a set of applications, it's worth mentioning that the API can (drastically) change in future releases — meaning that you might end up refactoring its usage.

During the scope of this book, you'll only need to add **kotlinx-serialization-json**. Navigate to **shared**, open the **build.gradle.kts** file and search for the **commonMain** field. There's already a set of dependencies on the project.

After `kotlinx.datetime`, add:

```
implementation(libs.kotlinx.serialization.json)
```

Synchronize the project, and now you're ready to use serialization for JSON format.

Creating a Custom Serializer

If you're using custom types on your objects, there might be no serializer available. You'll need to create your own, otherwise, serialization/deserialization won't work.

A good example of this is on the **KodecoContent.kt** data class, inside the **shared** module. The `platform` field is of type `PLATFORM`, an **enum** created to identify which section the article belongs to.

If you don't create a custom serializer, **kotlinx-serialization-json** won't be able to identify what the keywords "All," "Android," "iOS," "Unity" or "Flutter" are. This happens because the attribute on the JSON is type `String`, and it should be `PLATFORM` instead. You need to implement a custom serializer/deserializer to provide this support.

In the `commonMain` package inside the `shared` module, go over to `data` and create a `KodecoSerializer.kt` file.

Start by adding the `findByKey` method:

```
private fun findByKey(key: String, default: PLATFORM =  
    PLATFORM.ALL): PLATFORM {  
    return PLATFORM.values().find { it.value == key } ?: default  
}
```

This will receive a key and return the corresponding value in the enum `PLATFORM`.

Now that you've added the mapping function, create the `KodecoSerializer` object. In the same file, add the following code and import the required classes:

```
//1  
object KodecoSerializer : KSerializer<PLATFORM> {  
  
    //2  
    override val descriptor: SerialDescriptor =  
        PrimitiveSerialDescriptor("PLATFORM", PrimitiveKind.STRING)  
  
    //3  
    override fun serialize(encoder: Encoder, value: PLATFORM) {  
        encoder.encodeString(value.value.lowercase())  
    }  
  
    //4  
    override fun deserialize(decoder: Decoder): PLATFORM {  
        return try {  
            val key = decoder.decodeString()  
            findByKey(key)  
        } catch (e: IllegalArgumentException) {  
            PLATFORM.ALL  
        }  
    }  
}
```

Here's a step-by-step breakdown of this logic:

1. You extend the `KSerializer` class and define the type of object that you are going to serialize/deserialize.
2. It's necessary to define the `PrimitiveSerialDescriptor` that contains the class name — `PLATFORM` — and how the parameter should be read. In this case, it's going to be from a `String`.
3. Now that everything is ready, it's time to define the `serialize` method. Since, the content type of `PLATFORM` is `String`, you just need to call `encodeString` and send the value of the received object.
4. Finally, to deserialize, you're going to call the opposite method, which is `decodeString`, and with the raw value (the key), you'll call `findByKey` to see which value of the enum it corresponds to.

That's it! `KodecoSerializer` is ready. You just need to add it to the class. Open the `KodecoContent.kt` file again, and above the declaration of `PLATFORM` add:

```
@Serializable(with = KodecoSerializer::class)
```

This associates the new device serializer to the `PLATFORM` class. Remember to import from `kotlinx.serialization.Serializable`.

Serializing/Deserializing New Data

Navigate to `FeedPresenter.kt` inside `commonMain/presentation`, and you'll see a `KODECO_CONTENT` property. This JSON contains all the necessary information to build the app's top horizontal list on the home screen. Its structure has the following attributes:

- **platform**: The different areas covered by Kodeco articles: Android, iOS, Unity and Flutter. There's a fifth value — all. Once set, it removes this filter and shows everything published.
- **url**: Contains the RSS feed URL from where the articles should be fetched.
- **image**: A cover image that corresponds to the `platform` that was selected.

These three attributes are already mapped into the data class `KodecoContent`, which is inside `data/model`. Open it and add to the top of its declaration:

```
@Serializable
```

The `KodecoContent` data class uses `@Serializable`, so when decoding the `KODECO_CONTENT` property, it easily generates a list of `KodecoContent` that maps the attributes of the JSON string into the fields of the data class.

With everything defined, navigate to `FeedPresenter.kt` and first add to the class:

```
private val json = Json { ignoreUnknownKeys = true }
```

This will create the `Json` object that you'll use to decode the file content. It's important to set `ignoreUnknownKeys` to `true` to avoid any exceptions that might be thrown in case one of the fields inside `KodecoContent.kt` doesn't have a direct attribute in the JSON file. Remember to import `kotlinx.serialization.json.Json`.

Now, update the `content` property to decode the `KODECO_CONTENT` instead of returning an `emptyList`:

```
val content: List<KodecoContent> by lazy {
    json.decodeFromString(KODECO_CONTENT)
}
```

`content` is lazily initialized. In other words, it will open the file and read its content only when accessed. When done, it calls `decodeFromString` to generate a list of `KodecoContent` objects.

Build and run the project and see what's new in `learn`. You'll see screens similar to the following ones on different platforms:

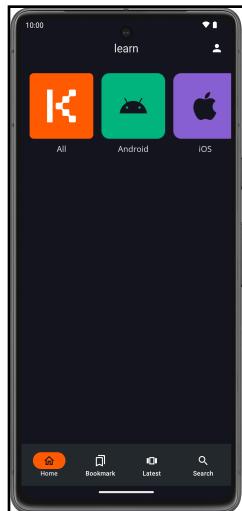


Fig. 11.6 – Android app with different platforms

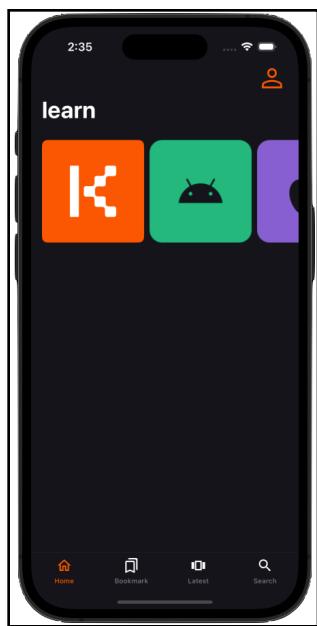


Fig. 11.7 – iOS app with different platforms

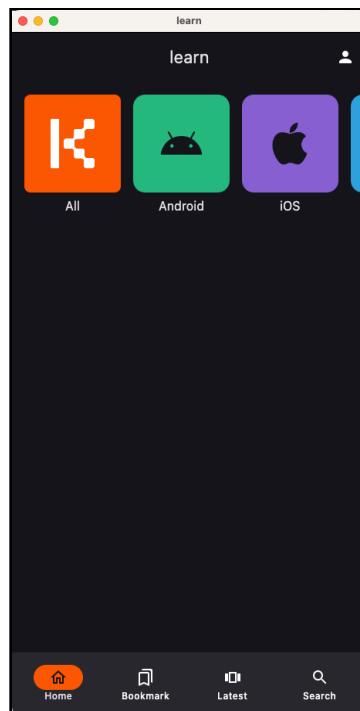


Fig. 11.8 – Desktop app with different platforms

Serializable vs. Parcelable

Java has a `Serializable` interface located in the `java.io` package. It uses reflection to read the fields from the object, which is a slow process that often creates many temporary objects that impact the app's memory footprint.

On the other hand `Parcelable`, an Android specific equivalent for `Serializable`, requires all the object types to be declared. This makes it a faster solution, since there's no need to use reflection to understand the object type.

One might argue that `Parcelable` is more complex to implement. This was true some years ago, since it was necessary to override a couple of methods and create the read/write methods according to the object fields. But, you don't have to do all that now, as the `kotlin-parcelize` plugin generates this code automatically. So the only effort here is to add an annotation to the top of the class — `@Parcelize` — and extend `Parcelable`.

Implementing Parcelize in KMP

`Parcelable` and `Parcelize` are a set of classes that are specific to the Android platform. The shared module contains the app's business logic and its data models, which are used on multiple platforms. Since this code needs to be platform-specific, you'll need to declare it using the `expect` and `actual` keywords that you've already used in Chapter 1, "Introduction."

`Parcelize` is part of a plugin named `kotlin-parcelize`, which contains the `Parcelable` code generator. Before writing the Android declaration for this class, you'll need to define it in `libs.versions.toml`, located inside the `gradle` folder. Here go to the `[plugins]` section and add:

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

Now open the `build.gradle.kts` file on the root folder and add to the `plugins` section the declaration that you've just defined:

```
alias(libs.plugins.jetbrains.kotlin.parcelize) apply false
```

Finally, go to the **shared** module's **build.gradle.kts** file. Open it, and in the **plugin** section add:

```
alias(libs.plugins.jetbrains.kotlin.parcelize)
```

Synchronize the project.

To set a data class as **Parcelable**, one would add the annotation `@Parcelize` to the top of the class declaration and then extend the **Parcelable** generator. It should be something similar to:

```
import kotlinx.parcelize.Parcelize

@Parcelize
data class KodecoEntry(val id: String, val entry: String):
    Parcelable
```

However, since this platform-specific code shouldn't exist on **commonMain**, you'll need to define this behavior at the platform level — in this case in **androidMain**.

Defining expect/actual for Android Target

As a rule of thumb, the name of the classes that are platform-specific start with the prefix **Platform-**. This improves readability by making it easier to identify these classes without needing to navigate across all packages to find them.

To implement **Parcelize**, you need to declare:

- The **Parcelable** interface, which is extended by the data class.
- The **Parcelize** annotation, which is used to activate the **kotlin-parcelize** plugin.

Go to **commonMain** in the **shared** module, navigate to **platform** and create a file named **Parcelable.common.kt**. Open it and add:

```
package com.kodeco.learn.platform

//1
expect interface Parcelable

//2
@OptIn(ExperimentalMultiplatform::class)
@OptionalExpectation
//3
@Target(AnnotationTarget.CLASS)
@Retention(AnnotationRetention.BINARY)
//4
```

```
expect annotation class Parcelize()
```

Let's break this code snippet into parts:

1. The **Parcelable** interface that needs to be defined.
2. Declaring an annotation that it's still experimental, meaning that the API might change in the future. These two annotations notify the user about this behavior and prevent Android Studio from warning the developer every time there's a call to this **Parcelize** class.
3. Target and Retention annotations are part of Kotlin's **Parcelize** annotation. To keep the same behavior as the native one, they're also added here.
4. The **Parcelize** annotation that will be defined.

Note: When using the `OptIn` annotation, you might see warning messages on your build log stating:

This class can only be used with the compiler argument '-opt-in=kotlin.RequiresOptIn'

Open the **shared build.gradle.kts** file and add the following code at the end of this file:

```
kotlin.sourceSets.all {  
    languageSettings.optIn("kotlin.RequiresOptIn")  
}
```

Depending on the data classes that you're using, you might need to create an **expect/actual** class for other annotations. One of these examples is `@RawValue`, which is used along with default serializers for custom types. You can follow the same approach used on **Parcelize** to achieve the same goal:

```
@OptIn(ExperimentalMultiplatform::class)  
@OptionalExpectation  
@Target(AnnotationTarget.TYPE)  
@Retention(AnnotationRetention.BINARY)  
expect annotation class RawValue()
```

With the expected declarations defined, go over to **androidMain** and create the corresponding **actual** implementation. It should be located in the same directory as the file you've just added on **commonMain**.

Navigate to the **platform** directory and create the corresponding **Parcelable.android.kt** file. Open it and add:

```
package com.kodeco.learn.platform  
  
actual typealias Parcelable = android.os.Parcelable  
actual typealias Parcelize = kotlinc.android.parcel.Parcelize
```

Since **Parcelable** and **Parcelize** already exist on the Android platform, there's no need to create them. Instead, use **typealias** to create a reference link between the expected class and the actual class itself. In other words, it's as if you're calling **android.os.Parcelable** or **kotlinc.android.parcel.Parcelize** directly.

Note: Type aliases don't create a new type of data, but instead create a link to an existing one. Get more information about Kotlin's **typealias** directly from the official documentation (<https://kotlinlang.org/docs/type-aliases.html>).

Working With Targets That Don't Support Parcelize

Go to the **iosMain** folder and inside the **platform** directory, create the corresponding **Parcelable.ios.kt** file.

Note: Press **Alt + Enter**, and Android Studio automatically suggests creating the files for the remaining platforms. Generate these files by pressing **OK**.

Open **Parcelable.ios.kt** and add:

```
package com.kodeco.learn.platform  
  
actual interface Parcelable
```

That's it! iOS doesn't have a corresponding **Parcelable** interface, so define an empty declaration. The compiler automatically removes both the annotation and class since there's no declaration set. The generated framework doesn't contain any reference to **Parcelize** and **Parcelable**.

Create the file **Parcelable.desktop.kt** inside **desktopMain/platform** and add the same declaration.

This is the expected behavior, since this is an Android-only feature.

Adding Parcelize to Existing Classes

With everything set, go to **commonMain** and search for the **KodecoContent** and **KodecoEntry** data classes. They're inside the **data/model** package.

In each of these files, add the `@Parcelize` annotation above the class declaration and extend `Parcelable`:

The **KodecoContent.kt** file will look like this:

```
@Parcelize
@Serializable
data class KodecoContent(
    val platform: PLATFORM,
    val url: String,
    val image: String
) : Parcelable
```

And, **KodecoEntry.kt** will look like this:

```
@Parcelize
data class KodecoEntry(
    val id: String = "",
    val link: String = "",
    val title: String = "",
    val summary: String = "",
    val updated: String = "",
    val imageUrl: String = "",
    val platform: PLATFORM = PLATFORM.ALL,
    val bookmarked: Boolean = false
) : Parcelable
```

Don't forget to add the necessary imports. Now, you can start sending this object across different Activities without any problems.

Sharing Your Data Classes With the Server

Another excellent use case for Kotlin Multiplatform is the possibility to share your data classes across all platforms, including the server. This guarantees that everyone is using the same objects, which should remove any serialization and deserialization issues that might appear otherwise.

To create a new shared module, go to **File** ▶ **New** ▶ **New Module...** and select **Kotlin Multiplatform Shared Module**. Here, define the module name as **shared-dto** and the package name as **com.kodeco.learn**. Click **Next** and select **No Activity**.

Click on **Finish** to add the module to the project.

After the synchronization ends, open the **build.gradle.kts** from the newly added **shared-dto** and inside the **kotlin** section add the JVM target.

```
jvm()
```

The current version of your module template might not be using the most recent APIs, so you might need to make a couple of updates:

1. If it's calling `android()` to set the Android target, replace the code block with `androidTarget()`.
2. In the `sourceSets` section, instead of declaring the `commonMain` and `commonTest` properties, you can access them by calling `getByName()`:

```
sourceSets {  
    getByName("commonMain") {  
        dependencies {  
            implementation(libs.kotlinx.serialization.json)  
        }  
    }  
    getByName("commonTest") {  
        dependencies {  
            implementation(kotlin("test"))  
        }  
    }  
}
```

3. Set the `compatibilityOptions` to version 17 on the `android` section:

```
compileOptions {  
    sourceCompatibility = JavaVersion.VERSION_17  
    targetCompatibility = JavaVersion.VERSION_17  
}
```

With these changes, you'll no longer have warnings in your new module.

Since desktop and server are JVM platforms, you can just add a single target.

Synchronize the project.

Add a new **jvmMain** directory. To easily create this target, right-click on **shared-dto** folder and go-to **File** ▶ **New** ▶ **Directory** and select **jvmMain/kotlin**. Now go to **kotlin** folder and, once again, right-click and select **File** ▶ **New** ▶ **Package** and write **com.kodeco.learn**.

Now you'll need to move some files from **shared** to **shared-dto**:

1. Go to **shared/commonMain** and move the folder **data** to **shared-dto/commonMain**.
2. Create a **platform** directory on **androidMain**, **commonMain**, **iosMain** and **jvmMain**
3. Move the **Parcelable.*.kt** files from **shared** to the corresponding platform folders. Note, that **jvmMain** corresponds to **desktopMain**.
4. Since, **shared-dto** now contains **Parcelable** and **KSerializer** objects, you need to add the following plugins to **build.gradle.kts**:

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

And on **commonMain/dependencies**:

```
implementation(libs.kotlinx.serialization.json)
```

5. Remove the **Platform.*.kt** and **Greeting.kt** files from **androidMain**, **commonMain** and **iosMain** which were generated from the template.
6. Finally, open **shared/build.gradle.kts** to add the **shared-dto** to the project. Go to **commonMain/dependencies** and write:

```
api(project(":shared-dto"))
```

With this the Android and desktop applications can easily access these objects. For iOS, you need to take an extra step. If you now try to access **KodecoEntry** from Swift, you'll see that this object doesn't exist. Instead you need to call **SharedDto_KodecoEntry**. This is how Kotlin Multiplatform handles external libraries – it adds the module prefix. To overcome this and provide a better experience to iOS developers, you can export **shared-dto** when building the framework and then access **KodecoEntry** directly. Inside **it.binaries.framework.add**:

```
export(project(":shared-dto"))
```

Synchronize and compile the project.

With the clients updated, if you want to share the data classes with the server, you just need to compile the JVM version:

```
./gradlew :shared-dto:jvmJar
```

You'll see how to publish a library in Chapter 14, "Creating Your KMP Library".

Testing

Tests validate the assumptions you've written and give you an important safety net toward all future changes.

Testing Serialization

To test your code, you need to go to the **shared** module, right-click on the **src** folder and select **New > Directory**. In the drop-down, select **commonTest/kotlin**. Here, create a **SerializationTests** class:

```
class SerializationTests { }
```

You'll need to create encode and decode tests to validate that everything is working as expected. Start by writing the encoder. Add the following method to the class:

```
@Test
fun testEncodePlatformAll() {
    val data = KodecoContent(
        platform = PLATFORM.ALL,
        url = "https://www.kodeco.com/feed.xml",
        image = "https://play-lh.googleusercontent.com/
CAa4g9Ub0Jambautjl7l0fdiwjYoX040RbivxdkPDZNirQd23TXQAfFyPTN1VBW
yzDt"
    )

    val decoded = Json.encodeToString(KodecoContent.serializer(),
        data)

    val content = "{\"platform\":\"all\",\"url\":\"https://
www.kodeco.com/feed.xml\",\"image\":\"https://play-
lh.googleusercontent.com/
CAa4g9Ub0Jambautjl7l0fdiwjYoX040RbivxdkPDZNirQd23TXQAfFyPTN1VBW
yzDt\"}"
    assertEquals(content, decoded)
}
```

Here, you're validating that your JSON serialization is capable of encoding a **KodecoContent** object, `data`, to a string. This property corresponds to the **all** section in **learn**. If the serializer is working correctly, the result of `encodeToString` needs to be the same as `content` – otherwise the test will fail. Click the green arrow by the function on the left to run the test and select **android (:testDebugUnitTest)**. You'll see that the test passes.

For `assertEquals` you should import `kotlin.test.assertEquals`.

Now to test if the deserialization is correct, add the following method:

```
@Test
fun testDecodePlatformAll() {
    val data = "{\"platform\":\"all\",\"url\":\"https://www.kodeco.com/feed.xml\",\"image\":\"https://play-lh.googleusercontent.com/CAa4g9Ub0Jambautjl7l0fdiwjYoX040RbivxdkPDZNirQd23TXQAfFyPTN1VBWyzDt\"}

    val decoded =
        Json.decodeFromString(KodecoContent.serializer(), data)
    val content = KodecoContent(
        platform = PLATFORM.ALL,
        url = "https://www.kodeco.com/feed.xml",
        image = "https://play-lh.googleusercontent.com/CAa4g9Ub0Jambautjl7l0fdiwjYoX040RbivxdkPDZNirQd23TXQAfFyPTN1VBWyzDt"
    )

    assertEquals(content, decoded)
}
```

Essentially, you do the opposite. Starting with the JSON response, you'll need to call `decodeFromString` so **kotlinx.serialization** builds your **KodecoContent** object, `decoded`, and then you'll compare it with the one that you're expecting - `content`. If the content is the same, the test successfully passes. Run the test and see that it passes.

Testing Custom Serialization

To test your custom **KodecoSerializer**, you first need to define:

```
private val serializers = serializersModuleOf(PLATFORM::class,
    KodecoSerializer)
```

This **serializers** property contains the **serializer** you'll need to encode and decode your data.

Start by creating an encoding test:

```
@Test
fun testEncodeCustomPlatformAll() {
    val data = PLATFORM.ALL

    val encoded = Json.encodeToString(serializers.serializer(),
        data)
    val expectedString = "\"all\""
    assertEquals(expectedString, encoded)
}
```

When you receive a response, the body is a string response in a JSON format:

```
{
    "platform": "all",
    "url": "https://www.kodeco.com/feed.xml",
    "image": "https://play-lh.googleusercontent.com/
CAa4g9Ub0Jambautjl7l0fdiwjYoX040RbivxdkPDZNirQd23TXQAfFYPNTN1VBW
yzDt"
```

To test if **KodecoSerializer** is working correctly on the test above, check if the result corresponds to the JSON response "all" after encoding the **PLATFORM.ALL** property into a string.

If it does, the **assertEquals** function will return true, otherwise the test will fail.

Now, add the decoder test:

```
@Test
fun testDecodeCustomPlatformAll() {
    val data = PLATFORM.ALL
    val jsonString = "${data.value}\\""

    val decoded = Json.decodeFromString<PLATFORM>(jsonString)
    assertEquals(decoded, data)
}
```

Here, you're doing the opposite. From the string "all", returned from `data.value`, you want the corresponding `PLATFORM` enum value. For that, you call `decodeFromString` and confirm if the returned object is the one you're expecting.

Challenges

Here are some challenges for you to practice what you've learned in this chapter. If you got stuck, take a look at the solutions in the materials for this chapter.

Challenge 1: Loading an RSS Feed

You're currently loading the different sections from the `KODECO_CONTENT` property inside `FeedPresenter.kt`. In this challenge, you will:

- Create an `KODECO_ALL_FEED` property that contains the RSS feed content of one of the feed URLs from `KODECO_CONTENT`.
- Read this property and parse its content in the shared module so it can be available for all apps to use. Keep in mind that for this `KodecoEntry` needs to be serializable.
- In `androidApp` and `desktopApp`, open the `FeedViewModel.kt` file inside `ui/home`, and populate `items` with this new data.
- In `iOSApp`, open `FeedClient` inside `extensions`, and in the `fetchFeeds` function add the new feeds.

Challenge 2: Adding Tests to Your Implementation

Now that you've implemented this new feature, you'll add tests to guarantee your implementation is correct. Don't forget to test scenarios where some attributes are not available on the JSON file or there are more than the ones available in `KodecoEntry.kt`.

Note: You should be able to read a JSON string containing the content to be serialized and access the object afterward.

Key Points

- Exchanging data between local and remote applications requires the content that's transferred to be serialized and deserialized, depending on whether it's being sent or received, respectively.
- **kotlinx.serialization** is a multiplatform library that supports serialization. It allows serializing/ deserializing **JSON**, **Protocol buffers**, **CBOR**, **Properties**, **HOCON**, **YAML** and **Apache Avro**.
- You can create custom serializers by implementing the `serialize/ deserialize` for a custom type and then associating it with that class.
- You can use **typealias** with **actual** to automatically link a class declaration to an existing one at the platform level.

Where to Go From Here?

For other practical examples where **Parcelize** is used, read the Kotlin Android Extensions (<https://www.kodeco.com/84-kotlin-android-extensions>) article.

The next chapter starts with the features that you've implemented in this one, but instead of only loading the data locally, you'll also fetch it from the network.

12

Chapter 12: Networking

By Carlos Mota

Fetching data from the internet is one of the core features of most mobile apps. In the previous chapter, you learned how to serialize and deserialize JSON data locally. Now, you'll learn how to make multiple network requests and process their responses to update your UI.

By the end of the chapter, you'll know how to:

- Make network requests using Ktor.
- Parse network responses.
- Test your network implementation.

The Need for a Common Networking Library

Depending on the platform you're developing for, you're probably already familiar with Retrofit (<https://square.github.io/retrofit/>) (Android), Alamofire (<https://github.com/Alamofire/Alamofire>) (iOS) or Unirest (<http://kong.github.io/unirest-java/>) (desktop).

Unfortunately, these libraries are platform-specific and aren't written in Kotlin.

Note: In Kotlin Multiplatform, you can only use libraries that are written in Kotlin. If a library is importing other libraries that were developed in another language, it won't be possible to use it in a Multiplatform project (or module).

Ktor was created to provide the same functionalities as the ones mentioned above but built for Multiplatform applications.

Ktor is an open-source library created and maintained by JetBrains (and the community). It's available for both client and server applications.

Note: Find more information about Ktor on the official website (<https://ktor.io/>).

Adding Ktor

Open **libs.versions.toml**. Inside the [versions] section, add the following Ktor version:

```
ktor = "2.3.4"
```

Now scroll down to the [libraries] section and add:

```
ktor-client-core = { module = "io.ktor:ktor-client-core",
version.ref = "ktor" }
ktor-client-serialization = { module = "io.ktor:ktor-client-
serialization", version.ref = "ktor" }
```

Here, you're defining the Ktor core library along with the serializations library for JSON that will parse the responses and transform the data into objects the app can process.

Ktor has different HTTP client engines depending on the platform to which you're compiling the project. Although desktop doesn't require a specific library, since you're also targeting Android and iOS, you'll need to add the below-mentioned Ktor libraries:

```
ktor-client-android = { module = "io.ktor:ktor-client-android",
version.ref = "ktor" }
ktor-client-ios = { module = "io.ktor:ktor-client-ios",
version.ref = "ktor" }
```

Now that the libraries are defined it's time to add them to **build.gradle.kts** located inside **shared** module.

Start by adding Ktor to **commonMain** dependencies section:

```
implementation(libs.ktor.client.core)
implementation(libs.ktor.client.serialization)
```

Afterwards, add the client versions to Android and iOS in **androidMain** and **iosMain** respectively:

```
implementation(libs.ktor.client.android)
```

```
implementation(libs.ktor.client.ios)
```

Click **Sync Now** to fetch and import these new libraries.

Connecting to the API With Ktor

To build **learn**, you'll make three different requests to:

- The RSS feed of a specific topic.
- An article webpage.
- Your Gravatar account.

The data for the first one is in the `KODECO_CONTENT` property inside the `FeedPresenter.kt` file located in the `shared` module. It can be one of the following:

- All (<https://www.kodeco.com/feed.xml>)
- Android (<https://kodeco.com/android/feed>)
- iOS (<https://kodeco.com/ios/feed>)
- Flutter (<https://kodeco.com/flutter/feed>)
- Server-Side Swift (<https://kodeco.com/server-side-swift/feed>)
- Game Tech (<https://kodeco.com/gametech/feed>)
- Professional Growth (<https://kodeco.com/professional-growth/feed>)

Each of these requests loads the latest 20 articles published for its category.

The second request corresponds to the `link` field of the `KodecoEntry`. Since an RSS entry doesn't contain a URL for the article image, you'll need to fetch it manually from `kodeco.com`.

Finally, make the last request to Gravatar (<https://en.gravatar.com/>), a service that allows you to define an online profile that can be used across external sites. Your picture from Kodeco, for example, is retrieved from this service.

Making a Network Request

Create a `data` folder inside `shared/src/commonMain/kotlin/com.kodeco.learn` module and then a new file inside named `FeedAPI.kt`. Add the following code:

```
//1
public const val GRAVATAR_URL = "https://en.gravatar.com/"
public const val GRAVATAR_RESPONSE_FORMAT = ".json"

//2
@ThreadLocal
public object FeedAPI {

    //3
    private val client: HttpClient = HttpClient()

    //4
    public suspend fun fetchKodecoEntry(feedUrl: String):
    HttpResponse = client.get(feedUrl)

    //5
```

```
public suspend fun fetchMyGravatar(hash: String):  
    GravatarProfile =  
  
    client.get("$GRAVATAR_URL$hash$GRAVATAR_RESPONSE_FORMAT").body()  
}
```

When prompted for imports, use the following:

```
import io.ktor.client.HttpClient  
import io.ktor.client.call.body  
import io.ktor.client.request.get  
import io.ktor.client.statement.HttpResponse  
import kotlin.native.concurrent.ThreadLocal
```

In the code above:

1. The constants `fetchMyGravatar` will use to make its request: the URL and the response format.
2. This annotation is only valid for iOS (Kotlin/Native). It's ignored on both Android and desktop. Using `@ThreadLocal`, the FeedAPI won't be shared across other threads that try to access it. Instead, a new copy will be made. This guarantees the object won't freeze. Read more about this in Chapter 13, "Concurrency".
3. Initialization of the `HttpClient` that you'll use to make the requests.
4. This function receives a feed URL for a specific topic, makes the request and returns it as a response via a `HttpResponse`. In this object, you can get additional information about the status code of the response, its body, etc.
5. Finally, you'll access Gravatar to retrieve information about your profile. In this case, the method returns `GravatarProfile` instead of `HttpResponse` and you'll shortly see how this is handled.

You're making a **GET** request in **learn**. Other HTTP methods are also available with Ktor: **POST**, **PUT**, **DELETE**, **HEAD**, **OPTION** and **PATCH**.

Note: If you look closely at these functions, you'll see they're declared using the keyword `suspend`. It's used so the current thread won't get blocked while waiting for a response. You'll learn more about it and coroutines in Chapter 13, "Concurrency".

You've made the requests, and now it's time to process the responses.

Plugins

Ktor has a set of plugins already built in that are disabled by default. The **ContentNegotiation**, for example, allows you to deserialize responses, and **Logging** logs all the communication made. You'll see an example of both later in this chapter.

These plugins intercept all the requests and responses made, then process them according to their purpose.

Parsing Network Responses

To deserialize a JSON response you need to add two new libraries. First, open the **libs.versions.toml** file and in the **[libraries]** section below the other Ktor declarations add:

```
ktor-client-content-negotiation = { module = "io.ktor:ktor-client-content-negotiation", version.ref = "ktor" }
ktor-serialization-kotlinx-json = { module = "io.ktor:ktor-serialization-kotlinx-json", version.ref = "ktor" }
```

Second, navigate to **build.gradle.kts** file from **shared** module and in **commonMain/dependencies** section, add:

```
implementation(libs.ktor.client.content.negotiation)
implementation(libs.ktor.serialization.kotlinx.json)
```

Click **Sync Now**.

In **FeedAPI**, you've got two functions that return an **HttpResponse**:

- **fetchKodecoEntry** accesses the Kodeco XML feed. Since there's no direct way to support its serialization from Ktor or an official library from JetBrains at the moment, you'll use one from the community: KorIO (<https://docs.korge.org/korio/>).
- **fetchMyGravatar** is set to receive a JSON response containing information about your Gravatar account.

You'll start with **fetchMyGravatar**. Since it's JSON, you can install **ContentNegotiation** for **json** so the response from this function will be the deserialized object.

To achieve this, update the client initialization with:

```
private val client: HttpClient = HttpClient {  
    install(ContentNegotiation) {  
        json(nonStrictJson)  
    }  
}
```

Ktor will now use `json` to deserialize the response body. Additionally, you also need to define the `nonStrictJson` property. Declare it before the `HttpClient`:

```
private val nonStrictJson = Json { isLenient = true;  
ignoreUnknownKeys = true }
```

When prompted for imports, add:

```
import  
io.ktor.client.plugins.contentnegotiation.ContentNegotiation  
import io.ktor.serialization.kotlinx.json.json  
import kotlinx.serialization.json.Json
```

To keep your app stable on any future server update, it's always a good approach to define `isLenient` and `ignoreUnknownKeys` as true. Otherwise, the deserialization might throw an exception if there's malformed input or there are properties in the JSON that don't exist in the serializable object.

Now, when you call `fetchMyGravatar`, instead of receiving a `HttpResponse` that you would need to process, you'll receive the deserialized object that you can use.

Open the `GravatarContent.kt` file in the `shared-dto -> commonMain -> data` folder and ensure that the `@Serializable` annotation is present for `GravatarProfile` and `GravatarEntry`.

Logging Your Requests and Responses

Logging all the communication with the server is important so you can identify any error that might exist.

Ktor has native support for logging. Before writing the logger, you need to open the `libs.versions.toml` file and in `[libraries]` section, after the existing Ktor declarations write:

```
ktor-client-logging = { module = "io.ktor:ktor-client-logging",  
version.ref = "ktor" }
```

Now return to **build.gradle.kts** from the **shared** module, and in the **commonMain** dependencies, add:

```
implementation(libs.ktor.client.logging)
```

Do a Gradle sync.

When ready, return to **FeedAPI.kt** and add the following code inside the **HttpClient** initialization lambda:

```
//1
install(Logging) {
    //2
    logger = Logger.DEFAULT
    //3
    level = LogLevel.HEADERS
}
```

Here's what's happening in the code above:

1. You install the **Logging** feature in the app to intercept all the network requests and responses.
2. You specify the logger class that you'll use to log all the network communication. Here **DEFAULT** refers to the default logger implementation provided by Ktor itself which uses an **SLF4J** logging framework. Using it falls back to calling the **println** function.
3. Specifies the data to be logged.

The different types of logging levels are:

- **LogLevel.ALL**: Where everything is logged. Importantly, with this log level, if you're uploading a large file, all of its content will be printed. This ultimately can lead to a buffer overflow error and crash your app. Don't forget to cover this scenario.
- **LogLevel.HEADERS**: Logs the request and response headers.
- **LogLevel.BODY**: Logs the request and response body.

- `LogLevel.INFO`: Logs the URL and the method of the requests. For responses, this means its status, method and the “from” field.
- `LogLevel.NONE`: Nothing will be logged. As a safety mechanism, if you’re building your app for production, you should select this level. Otherwise, you risk that someone might access your network logs by simply opening **Logcat** with the device plugged into the computer.

Don’t forget to import:

```
import io.ktor.client.plugins.logging.DEFAULT
import io.ktor.client.plugins.logging.LogLevel
import io.ktor.client.plugins.logging.Logger
import io.ktor.client.plugins.logging.Logging
```

Additionally, you can define a custom logger class. To accomplish this, go to the **data** folder inside the **shared** module and create a **HttpClientLogger.kt** file with the following code:

```
import com.kodeco.learn.platform.Logger

private const val TAG = "HttpClientLogger"

public object HttpClientLogger : io.ktor.client.plugins.logging.Logger {

    override fun log(message: String) {
        Logger.d(TAG, message)
    }
}
```

Here, you’re extending the Ktor Logger and configuring how to log the requests and responses. You do this by overriding the `log` function. Instead of using the default logger, you’re using the app Logger defined on **shared**.

Now, return to **FeedAPI.kt** and update the previously added `install` call to instead use:

```
logger = HttpClientLogger
```

Build and run the apps to confirm everything is correct.

For now, since there are no requests made, you won't find any log message when filtering for **HttpClientLogger** both in Android Studio and Xcode. After completing the next section, you'll try this again.



Fig. 12.1 – Android Studio Logcat filtered by *HttpClientLogger*



Fig. 12.2 – Xcode Console filtered by *HttpClientLogger*

You can use the filter fields both in Android Studio and Xcode to display only messages that match a specific tag.

Note: Since the logger you created receives a TAG parameter that corresponds to the **HttpClientLogger** class, you can use that to filter on **Logcat** for all the network requests and responses made.

Retrieving Content

Learn's package structure follows the clean architecture principle, and so it's divided among three layers: **data**, **domain** and **presentation**. In the data layer, there's the **FeedAPI.kt** that contains the functions responsible for making the requests. Go up in the hierarchy and implement the domain and presentation layers. The UI will interact with the presentation layer.

Interacting With Gravatar

Open the **GetFeedData.kt** file inside the **domain** folder of the **shared** module. Inside the class declaration, replace the `TOD0` commentary with:

```
//1
public suspend fun invokeGetMyGravatar(
    hash: String,
    onSuccess: (GravatarEntry) -> Unit,
    onFailure: (Exception) -> Unit
) {
    try {
        //2
        val result = FeedAPI.fetchMyGravatar(hash)
        Logger.d(TAG, "invokeGetMyGravatar | result=$result")

        //3
        if (result.entry.isEmpty()) {
            coroutineScope {
                onFailure(Exception("No profile found for hash=$hash"))
            }
        } //4
        else {
            coroutineScope {
                onSuccess(result.entry[0])
            }
        }
    } //5
    catch (e: Exception) {
        Logger.e(TAG, "Unable to fetch my gravatar. Error: $e")
        coroutineScope {
            onFailure(e)
        }
    }
}
```

Add the following imports:

```
import com.kodeco.learn.data.FeedAPI
import com.kodeco.learn.data.model.GravatarEntry
import com.kodeco.learn.platform.Logger
import kotlinx.coroutines.coroutineScope
```

Here's what's happening:

1. This function receives a `hash` property that's you'll use to build the request to Gravatar. There are two lambda functions: `onSuccess` will be called if the operation succeeded and `onFailure` in case the operation failed.
2. `fetchMyGravatar` uses the `ContentNegotiation` you previously installed. So it will return an object containing the response data instead of returning `HttpResponse` (unlike the other function).
3. A response is valid if there's at least one element in `result`. If this list is empty, it means the response is empty, and therefore `onFailure` is triggered.
4. If it retrieves a response containing at least one entry, though, `onSuccess` is called with the first object of the list.
5. Finally, if anything fails during this process, `onFailure` is called with the exception that caused the problem.

Now that the domain logic is ready, move to the presentation layer. Open `FeedPresenter.kt`. Before the class declaration, add and define your `GRAVATAR_EMAIL`:

```
private const val GRAVATAR_EMAIL = "YOUR_GRAVATAR_EMAIL"
```

Create an account on Gravatar if you don't already have one, and replace `YOUR_GRAVATAR_EMAIL` with your Gravatar email. Once done, add the following function the UI will call inside the existing class:

```
//1
public fun fetchMyGravatar(cb: FeedData) {
    Logger.d(TAG, "fetchMyGravatar")

//2
    MainScope().launch {
        //3
        feed.invokeGetMyGravatar(
            //4
            hash = GRAVATAR_EMAIL.toByteArray().md5().toString(),
            //5
```

```
        onSuccess = { cb.onMyGravatarData(it) },
        onFailure = { cb.onMyGravatarData(GravatarEntry()) }
    )
}
}
```

Here's a step-by-step breakdown of this logic:

1. This is a function that allows you to set a listener for the UI to receive updates for the call to `fetchMyGravatar`. The `FeedData` argument is an interface used to notify the UI when new data is available. This scenario triggers `onMyGravatarData`.
 2. Since `invokeGetMyGravatar` is declared using a suspend function, you need to call it from a coroutine. To keep things simple in this chapter, you're going to use `MainScope` for that.
 3. Calls the `invokeGetMyGravatar` to make the request for the Gravatar.
 4. The Gravatar request requires an md5 hash of the email the user has registered. For this you'll use the `korIO` library that's already added to the project.
 5. If the request succeeds, it calls the `onSuccess` expression with the received data. Otherwise, `onFailure` is triggered and an empty `GravatarEntry` is sent.

Don't forget to import:

```
import com.kodeco.learn.data.model.GravatarEntry
import com.kodeco.learn.platform.Logger
import io.ktor.utils.io.core.toByteArray
import korlibs.crypto.md5
import kotlinx.coroutines.MainScope
import kotlinx.coroutines.launch
import com.kodeco.learn.domain.cb.FeedData
```

Finally, it's time to update the apps.

Go over to **androidApp** and in the **FeedViewModel.kt** file inside the **ui/home** folder, update the existing `fetchMyGravatar` to call the entry point that you defined before:

```
fun fetchMyGravatar() {  
    Logger.d(TAG, "fetchMyGravatar")  
    presenter.fetchMyGravatar(this)  
}
```

When the Gravatar profile is available, it triggers `onMyGravatarData`. Update it to set this data on the `_profile` property:

```
override fun onMyGravatarData(item: GravatarEntry) {
    Logger.d(TAG, "onMyGravatarData | item=$item")
    viewModelScope.launch {
        _profile.value = item
    }
}
```

Finally, add the following imports:

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

Now that you've got everything ready, build and run the Android app.

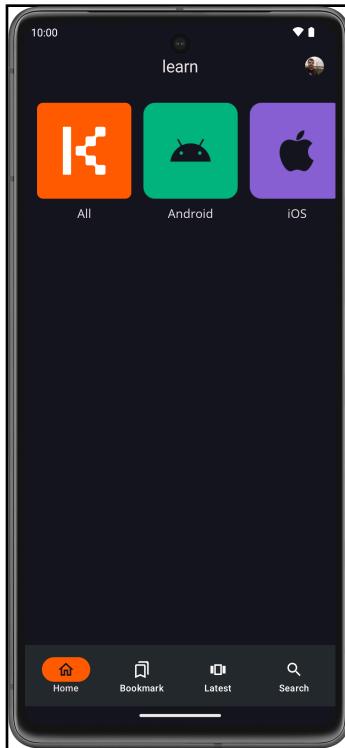


Fig. 12.3 – Profile picture in Android App

You can see your avatar on the top right corner of the top bar.

To implement the same feature on the desktop app, open its `FeedViewModel.kt`, located in the `desktopApp` module's `ui/home` folder.

Similar to what you added for Android, update the existing `fetchMyGravatar` function to:

```
fun fetchMyGravatar() {
    Logger.d(TAG, "fetchMyGravatar")
    presenter.fetchMyGravatar(this)
}
```

To make the corresponding requests and update the `onMyGravatarData` to notify the UI once they are available, update the `onMyGravatarData` method to the following:

```
override fun onMyGravatarData(item: GravatarEntry) {
    Logger.d(TAG, "onMyGravatarData | item=$item")
    viewModelScope.launch {
        profile.value = item
    }
}
```

Don't forget to import the libraries. Finally, compile and run your app using the following command:

```
./gradlew desktopApp:run
```

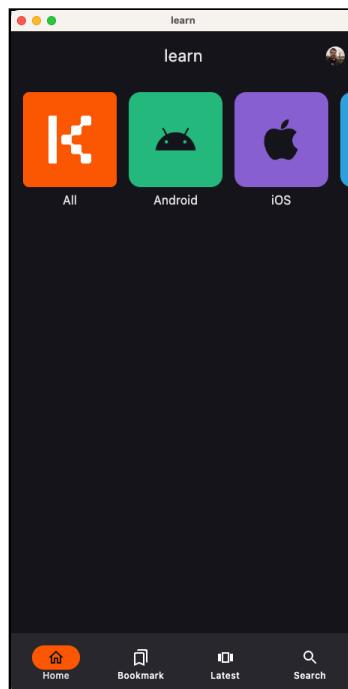


Fig. 12.4 – Profile picture in Desktop App

Switch to Xcode, and navigate to the **extensions** folder. Here, open the **FeedClient.swift** class and find the `fetchProfile` function. Before assigning the completion to the `handlerProfile`, add this code to fetch the Gravatar profile:

```
feedPresenter.fetchMyGravatar(cb: self)
```

Build and run your iOS app.

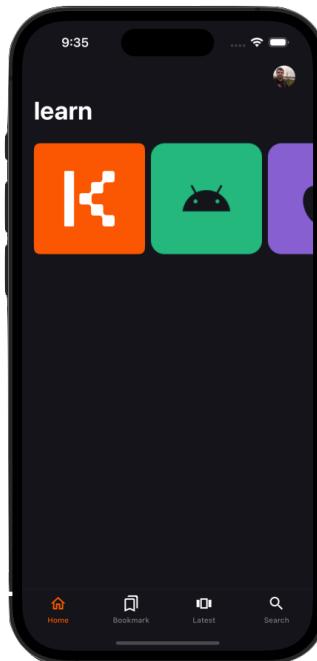


Fig. 12.5 – Profile picture in iOS App

Interacting With the Kodeco RSS Feed

Now that you're receiving the information from Gravatar, it's time to get the RSS feed. Once again, open the **GetFeedData.kt** file in **shared/domain** and add the following above `invokeGetMyGravatar` and add any imports if needed:

```
//1
public suspend fun invokeFetchKodecoEntry(
    platform: PLATFORM,
    imageUrl: String,
    feedUrl: String,
    onSuccess: (List<KodecoEntry>) -> Unit,
    onFailure: (Exception) -> Unit
){
```

```
try {
    //2
    val result = FeedAPI.fetchKodecoEntry(feedUrl)

    Logger.d(TAG, "invokeFetchKodecoEntry | feedUrl=$feedUrl")
    //3
    val xml = Xml.parse(result.bodyAsText())

    val feed = mutableListOf<KodecoEntry }()
    for (node in xml.allNodeChildren) {
        val parsed = parseNode(platform, imageUrl, node)

        if (parsed != null) {
            feed += parsed
        }
    }

    //4
    coroutineScope {
        onSuccess(feed)
    }
} catch (e: Exception) {
    Logger.e(TAG, "Unable to fetch feed:$feedUrl. Error: $e")
    //5
    coroutineScope {
        onFailure(e)
    }
}
}
```

Here's a step-by-step breakdown of this logic:

1. This function receives a `PLATFORM` enum value that corresponds to one of the different areas of articles you have at Kodeco: all, Android, iOS, Flutter, Server-Side Swift, Game Tech and Professional Growth. You use this to give the functionality to the UI to filter for specific types if required.
2. `result` holds the `HttpResponse` that's returned from `fetchKodecoEntry`. The parameter sent here is the URL where the request should be made.
3. Since there's no direct support for XML serialization in Ktor, you need to use a third-party library. In this case, due to its popularity, you're going to use **KorIO**. It will parse through all the nodes of the XML and return a list of `KodecoEntry`.
4. If everything worked until this next code block, this function ends by sending the feed to the `onSuccess` lambda.
5. On the contrary, if there was any issue, `onFailure` is triggered instead.

It's now time to move up in the hierarchy and open the **FeedPresenter.kt** file on the **presentation** layer inside **shared**. With the request implemented, you need to add an entry point the UI can call.

To achieve this, add the following functions above `fetchMyGravatar`:

```
//1
public fun fetchAllFeeds(cb: FeedData) {
    Logger.d(TAG, "fetchAllFeeds")

//2
    for (feed in content) {
        fetchFeed(feed.platform, feed.image, feed.url, cb)
    }
}

private fun fetchFeed(
    platform: PLATFORM,
    imageUrl: String,
    feedUrl: String,
    cb: FeedData
) {
    MainScope().launch {
        // 3
        feed.invokeFetchKodecoEntry(
            platform = platform,
            imageUrl = imageUrl,
            feedUrl = feedUrl,
            // 4
            onSuccess = { cb.onNewDataAvailable(it, platform,
null) },
            onFailure = { cb.onNewDataAvailable(emptyList(),
platform, it) }
        )
    }
}
```

Here's a logic breakdown:

1. The cb you'll use to notify the UI when new data is available.
2. content corresponds to the deserialization of the KODECO_CONTENT property. It should contain five different platform types: all, Android, iOS, Flutter, Server-Side Swift, Game Tech, and Professional Growth, each with its own feed URL. You're going to fetch them all.
3. invokeFetchKodecoEntry will call the GetFeedData that then calls the FeedAPI and sends the network request.
4. Finally, the onSuccess and onFailure expressions call the cb functions with the response data. In case the operation succeeds, the received list of KodecoEntry is sent, otherwise an empty list is sent.

With this, you've finished the business (shared) logic for the network requests. It's now time to connect it to the Android, desktop and iOS apps. Starting with Android, open the **FeedViewModel.kt** file. Look for the `fetchAllFeeds` function and add the following code inside the function:

```
presenter.fetchAllFeeds(this)
```

This will trigger the network request that you defined before. Scrolling down this file, you'll see the `onNewDataAvailable` implementation. Update it with the following code block so the `items` property can be updated:

```
override fun onNewDataAvailable(items: List<KodecoEntry>,
platform: PLATFORM, exception: Exception?) {
    Logger.d(TAG, "onNewDataAvailable | platform=$platform items=$
{items.size}")
    viewModelScope.launch {
        _items[platform] = items
    }
}
```

This is important because **MainActivity.kt** is observing all the changes on `items`.

Build and run the Android application. You'll see a screen similar to this one:

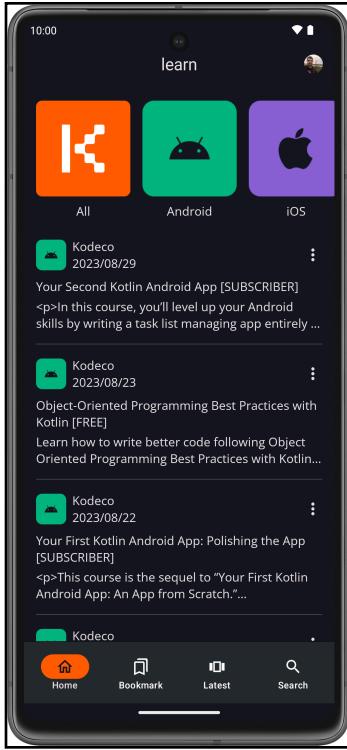


Fig. 12.6 – Feed in Android App

Navigate to the **desktopApp** project and add the same logic. On **FeedViewModel.kt**, find the `fetchAllFeeds` function and add:

```
presenter.fetchAllFeeds(this)
```

Main.kt calls this function to fetch all the available feeds. When they're ready, `onNewDataAvailable` is called with all the items. Update this function to:

```
override fun onNewDataAvailable(items: List<KodecoEntry>,
platform: PLATFORM, exception: Exception?) {
    Logger.d(TAG, "onNewDataAvailable | platform=$platform items=$
{items.size}")
    viewModelScope.launch {
        _items[platform] = items
    }
}
```

Now that the desktop app is ready, enter the compilation and run command at the Android Studio terminal:

```
./gradlew desktopApp:run
```

You'll see an app similar to this one:

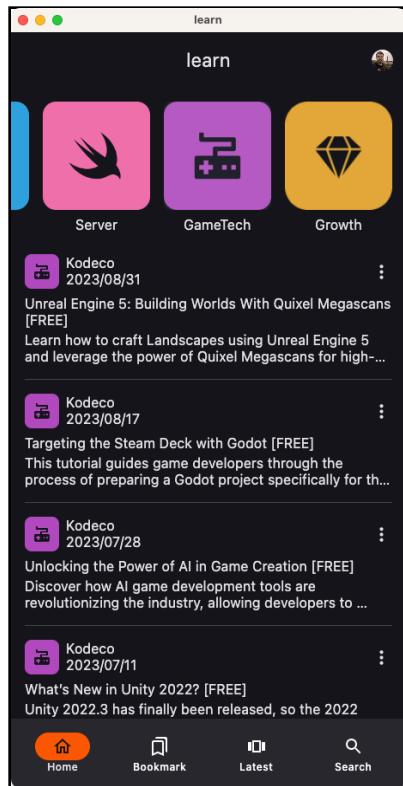


Fig. 12.7 – Feed in Desktop App

Finally, update the iOS app. Open the **FeedClient** file inside the **extensions** folder, and search for `fetchFeeds`. Here, before assigning the completion to the handler, add:

```
feedPresenter.fetchAllFeeds(cb: self)
```

That's it! Build and run the app, then see which articles the team recently published.

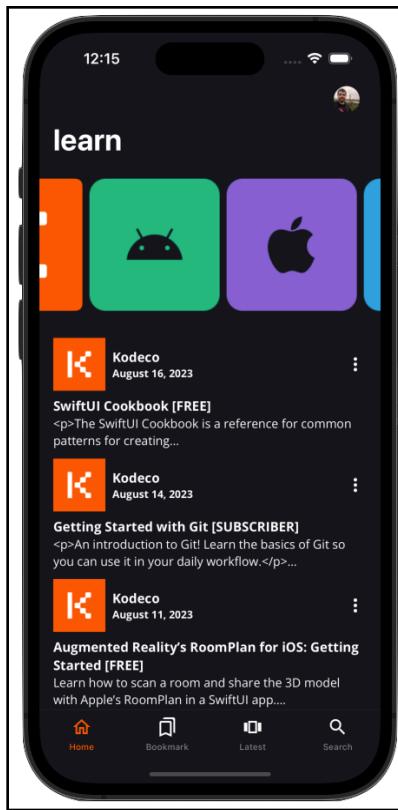


Fig. 12.8 – Feed in iOS App

Adding Headers to Your Request

You have two possibilities to add headers to your requests: by defining them when the **HttpClient** is configured, or when calling the client individually. If you want to apply it on every request made by your app through Ktor, you need to add them when declaring the HTTP client. Otherwise, you can set them on a specific request.

Imagine that you want to add a custom header to identify your app name.

First create a **Values.kt** file in the **shared/commonMain** module root folder. It should be located at the same level as **domain** and **platform**.

Then, add a constant that's going to be used to identify the parameter that you want to add as a header:

```
public const val X_APP_NAME: String = "X-App-Name"
```

This constant will be the header's key on both implementations.

Now, define its value by adding another property — this time it should correspond to the app name:

```
public const val APP_NAME: String = "learn"
```

Since this value should be the same for both platforms, you're going to use it as the value for the header request.

Now, if you want to add this header to all requests done through Ktor, you need to locate `client` in the **FeedAPI.kt** file. When you're overriding the `client`, before the call to `install` add:

```
defaultRequest {  
    header(X_APP_NAME, APP_NAME)  
}
```

Import the missing libraries. Calling `defaultRequest` directly is the equivalent of:

```
install(DefaultRequest)
```

In other words, similar to what you did for logging, you're setting the default configuration for every request. In this case, you're adding an `X_APP_NAME` header.

Now, compile the app on all three applications. By opening Logcat (Android), terminal (desktop) and Xcode console (iOS), confirm in the log messages that you're sending this new header.

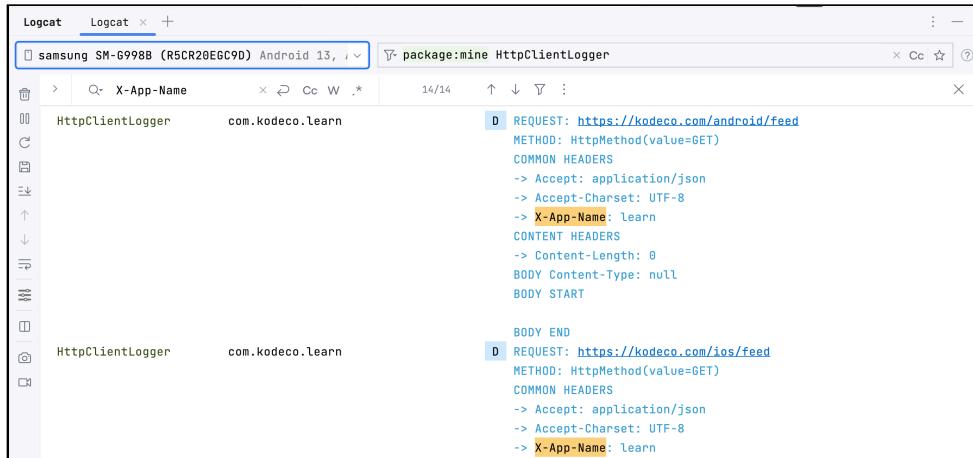


Fig. 12.9 – Android Studio Logcat showing all requests with a specific header

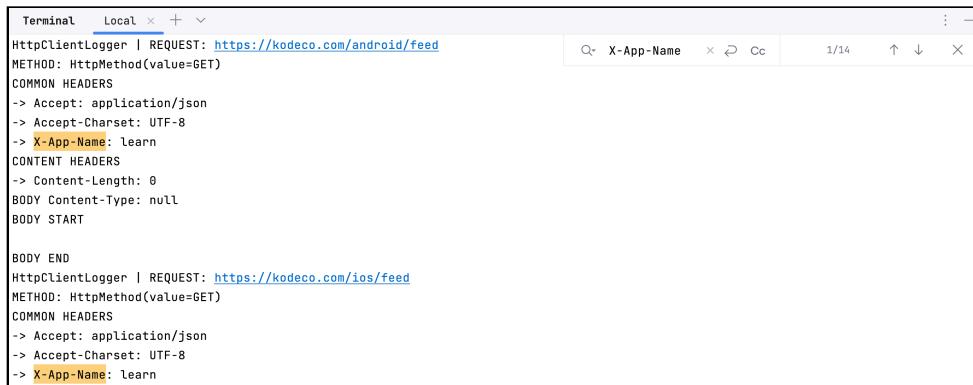


Fig. 12.10 – Terminal showing all requests with a specific header



Fig. 12.11 – Xcode showing all requests with a specific header

Hint: Don't forget that you can filter your logs using the tag `HttpClientLogger`.

On the contrary, if you want to add this header for a specific request, you just need to override the `HttpRequestBuilder` to set it. Here's a real example: imagine that you want to add it only when you're fetching your Gravatar profile. Remove the previously added header, and in the `fetchMyGravatar` declaration, update it to:

```
public suspend fun fetchMyGravatar(hash: String):  
    GravatarProfile =  
        client.get("$GRAVATAR_URL$hash$GRAVATAR_RESPONSE_FORMAT") {  
            header(X_APP_NAME, APP_NAME)  
        }.body()
```

With this, only this request contains the `X-APP_NAME` header.

To validate your implementation, compile the project again, and with the `HttpClientLogger` filter, search for this particular request.

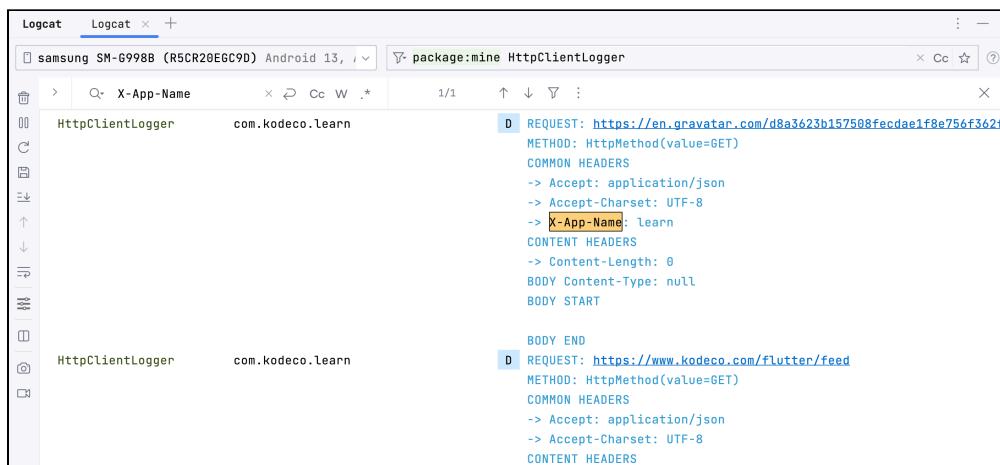


Fig. 12.12 – Android Studio Logcat showing a request with a specific header

```
Terminal Local x + v
HttpClientLogger | REQUEST: https://kodoco.com/professional-growth/feed
METHOD: HttpMethod(value=GET)
COMMON HEADERS
-> Accept: application/json
-> Accept-Charset: UTF-8
CONTENT HEADERS
-> Content-Length: 0
BODY Content-Type: null
BODY START

BODY END
HttpClientLogger | REQUEST: https://en.gravatar.com/d8a3623b157508fecdae1f8e756f362f.json
METHOD: HttpMethod(value=GET)
COMMON HEADERS
-> Accept: application/json
-> Accept-Charset: UTF-8
-> X-App-Name: learn
CONTENT HEADERS
-> Content-Length: 0
```

Fig. 12.13 – Terminal showing a request with a specific header

```
iosApp | Find v X-App-Name
HttpClientLogger | REQUEST: https://en.gravatar.com/d8a3623b157508fecdae1f8e756f362f.json
METHOD: HttpMethod(value=GET)
COMMON HEADERS
-> Accept: application/json
-> Accept-Charset: UTF-8
-> X-App-Name: learn
CONTENT HEADERS
-> Content-Length: 0
BODY Content-Type: null
BODY START

BODY END
HttpClientLogger | REQUEST: https://www.kodoco.com/feed.xml
METHOD: HttpMethod(value=GET)
COMMON HEADERS
-> Accept: application/json
-> Accept-Charset: UTF-8
CONTENT HEADERS
```

Fig. 12.14 – Xcode Console showing a request with a specific header

Uploading Files

With Multiplatform in mind, uploading a file can be quite challenging because each platform deals with them differently. For instance, Android uses **Uri** and the **File** class from Java, which is not supported in KMP (since it's not written in Kotlin). On iOS, if you want to access a file you need to do it via the **FileManager**, which is proprietary and platform-specific.

The solution is to find a common ground — in this case at a lower level. Their implementations generate a **ByteArray** that can be accessed and processed at the shared module.

Start by creating a data class that's going to represent an image. Go to **commonMain** and inside **platform** create a **MediaFile.common.kt** file:

```
public expect class MediaFile
```

```
public expect fun MediaFile.toByteArray(): ByteArray
```

Here, you're defining the class and function that's you'll use to represent a file. At the platform level, the `MediaFile` class and the corresponding `toByteArray` function will be defined.

With this in mind, navigate to `androidMain`. Inside `platform`, create the corresponding `actual` file – `MediaFile.android.kt`:

```
public actual typealias MediaFile = MediaUri

public actual fun MediaFile.toByteArray(): ByteArray =
    contentResolver.openInputStream(uri)?.use {
        it.readBytes()
    } ?: throw IllegalStateException("Couldn't open inputStream
$uri")
```

Here, you're defining the reference of `MediaFile` as `MediaUri`. Every time `MediaFile` is accessed, the properties and functions that will be called are the ones from `MediaUri`. This class doesn't yet exist. You'll need to create it, because in order to get the `ByteArray` from a file, Android needs to access the `openInputStream` from `contentResolver` that only exists in the activity context.

Create a new directory named `data` and then create a `MediaUri.kt` file inside it. Add the following code:

```
import android.content.ContentResolver
import android.net.Uri

public data class MediaUri(public val uri: Uri, public val
contentResolver: ContentResolver)
```

This `contentResolver` property is the one that's accessed in `toByteArray`, from which you can `openInputStream`.

Once done, it's now time to define the iOS implementation. Create the `MediaFile.ios.kt` in the `platform` package inside the `iosMain` folder and add:

```
import kotlinx.cinterop.ExperimentalForeignApi
import kotlinx.cinterop.addressOf
import kotlinx.cinterop.usePinned
import platform.Foundation.NSData
import platformUIKit.UIImage
import platformUIKit.UIImageJPEGRepresentation
import platform.posix.memcpy

public actual typealias MediaFile = UIImage
```

```
public actual fun MediaFile.toByteArray(): ByteArray {
    return UIImageJPEGRepresentation(this, compressionQuality = 1.0)?.toByteArray() ?: emptyArray<Byte>().toByteArray()
}

@OptIn(ExperimentalForeignApi::class)
fun NSData.toByteArray(): ByteArray {
    return ByteArray(length.toInt()).apply {
        usePinned {
            memcpy(it.addressOf(0), bytes, length)
        }
    }
}
```

In this case, `MediaFile` is represented as a `UIImage`. The `ByteArray` required for the upload is retrieved from the call to `UIImageJPEGRepresentation`.

With these implementations, you can now access the file's content and upload it. Although it's beyond the scope of this chapter, it's worth showing you an example of how it can be made at Ktor level.

Imagine that you selected an image to upload. Assuming your server supports multipart requests, you could write a similar function:

```
//1
public suspend fun uploadAvatar(data: MediaFile): HttpResponse {
//2
    return client.post(UPLOAD_AVATAR_URL) {
//3
        body = MultiPartFormDataContent(
            formData {
                appendInput("filedata", Headers.build {
//4
                    append(HttpHeaders.ContentType, "application/octet-
stream")
                })
                //5
                buildPacket { writeFully(data.toByteArray()) }
            }
        )
    }
}
```

Here's what's happening:

1. You need to receive the `MediaFile` that contains a reference to your image. The important part of this object is the `toByteArray` function that's used on 5.
2. The `client` in this example is the same that you've been using until now. There's no need to install additional plugins or set any configuration.

3. In this case, the file will be sent through a multipart request, so the body of the request needs to contain this information.
4. Most servers require that the request contains the content type of the file – in this case, `application/octet-stream`.
5. Depending on the total size of the file, more than one part might need to be sent. Although the result is always an array of bytes, depending on the platform that your app is running, `toByteArray` will call different functions.

Note: Depending on the file type you want to send and the server requirements, you may need to implement a different method. For more information, read the official documentation (https://ktor.io/docs/request.html#upload_file) from Ktor.

Testing

To write tests for Ktor, you need to create a mock object of the `HttpClient` and then test the different responses that you can receive.

Before writing, you need to open the `libs.versions.toml` file inside `gradle` folder and include first the JUnit version on `[versions]` section:

```
junit = "4.13.2"
```

And then add the libraries under the `[libraries]`:

```
junit = { module = "junit:junit", version.ref = "junit" }
ktor-client-mock = { module = "io.ktor:ktor-client-mock",
version.ref = "ktor" }
```

Now go to `build.gradle.kts` file from `shared` and update `commonTest` with these libraries:

```
implementation(kotlin("test-junit"))
implementation(libs.junit)
implementation(libs.ktor.client.mock)
```

Click **Sync Now**.

After it finishes, open `commonTest` and inside `shared`, create a `NetworkTests` class. All your network tests will be here.

Before creating the tests, you need to mock some objects. After the class declaration, add:

```
private val profile = GravatarProfile(  
    entry = listOf(  
        GravatarEntry(  
            id = "1000",  
            hash = "1000",  
            preferredUsername = "Ray_Wenderlich",  
            thumbnailUrl = "https://avatars.githubusercontent.com/u/  
4722515?s=200&v=4"  
    )  
)
```

This will be the `GravatarProfile` that you're expecting to receive on mocked network calls.

Now, you'll need to mock the `HttpClient`. Add it below the code you just pasted:

```
private val nonStrictJson = Json { isLenient = true;  
ignoreUnknownKeys = true }  
  
private fun getHttpClient(): HttpClient {  
    //1  
    return HttpClient(MockEngine) {  
  
        //2  
        install(ContentNegotiation) {  
            json(nonStrictJson)  
        }  
  
        engine {  
            addHandler { request ->  
                //3  
                if (request.url.toString().contains(GRAVATAR_URL)) {  
                    respond(  
                        //4  
                        content = Json.encodeToString(profile),  
                        //5  
                        headers = headersOf(HttpHeaders.ContentType,  
Content-Type.Application.Json.toString())  
                }  
                else {  
                    //6  
                    error("Unhandled ${request.url}")  
                }  
            }  
        }  
    }  
}
```

Here's a step-by-step breakdown of this logic:

1. Unit tests need to be mocked since you won't be making any network calls. The goal is to go through all the possible scenarios and validate that the app behaves accordingly. For that, you're initializing the `HttpClient` with a `MockEngine`.
2. To create a valid test, you need to follow the same configuration that you used when defining the requests. In this case, you need to use the `ContentNegotiation` plugin.
3. This `HttpClient` can be used by different requests, so you need to be able to identify who made the request and which response should be created.
4. The `content` defines the body of the response.
5. Defines the `content-type` of the response.
6. Generates an error in case the request URL doesn't match with any of the existing conditions.

Add the following imports as well:

```
import com.kodeco.learn.data.GRAVATAR_URL
import com.kodeco.learn.data.model.GravatarEntry
import com.kodeco.learn.data.model.GravatarProfile
import io.ktor.client.HttpClient
import io.ktor.client.engine.mock.MockEngine
import io.ktor.client.engine.mock.respond
import io.ktor.client.plugins.contentnegotiation.ContentNegotiation
import io.ktor.http.ContentType
import io.ktor.http.HttpHeaders
import io.ktor.http.headersOf
import io.ktor.serialization.kotlinx.json.json
import kotlinx.serialization.json.Json
import kotlinx.serialization.encodeToString
```

Finally, with the request and response defined, write the following test:

```
@Test
public fun testFetchMyGravatar() = runTest {
    val client = getHttpClient()
    assertEquals(profile, client.request(
        "$GRAVATAR_URL${profile.entry[0].hash}
$GRAVATAR_RESPONSE_FORMAT").body())
}
```

Resolve the imports as follows:

```
import com.kodeco.learn.platform.runTest
import kotlin.test.assertEquals
import io.ktor.client.request.request
import com.kodeco.learn.data.GRAVATAR_RESPONSE_FORMAT
import io.ktor.client.call.body
import kotlin.test.Test
```

The test passes if the response it receives is the same as the `profile` object mocked; it fails otherwise.

To run a test, right-click the class name **NetworkTests**, then click in “Run ‘NetworkTests’”, or with the file open, just click on the green arrows shown next to a test and choose **android (local)**.

Challenge

Here is a challenge for you to practice what you’ve learned in this chapter. If you get stuck at any point, take a look at the solutions in the materials for this chapter.

Challenge: Send Your Package Name in a Request Header

You’ve learned how to define a header in a request. In that example, you were sending the app name as its value. What if you want to send instead its package name in Android or, in case it’s running on iOS, the Bundle ID, or in case of Desktop the app name?

For this challenge, implement a way to get platform specific app identifiers and send the value with the X-App-Name header.

Note: You should implement this logic on the shared module.

Key Points

- Ktor is a set of networking libraries written in Kotlin. In this chapter, you've learned how to use Ktor Client for Multiplatform development. It can also be used independently in Android or desktop. There's also Ktor Server; that's used server-side.
- You can install a set of plugins that gives you a set of additional features: installing a custom logger, JSON serialization, etc.

Where to Go From Here?

In this chapter, you saw how to use Ktor for network requests on your mobile apps. Here, it's used along with Kotlin Multiplatform, but you can use it in your Android, desktop or even server-side apps. To learn how to implement these features on other platforms, you should read Compose for Desktop (<https://www.kodeco.com/26791460-compose-for-desktop-get-your-weather>), or — if you want to use it server-side — watch this video course (<https://www.kodeco.com/2885892-server-side-kotlin-with-ktor>). Additionally, there's also a tutorial focused on the integration of Ktor with GraphQL (<https://www.kodeco.com/18858740-ktor-and-graphql-getting-started>) that you might find interesting.

The next chapter is focused on concurrency — in particular, how to use coroutines in your application.

See you there.

Chapter 13: Concurrency

By Carlos Mota

As an app gets more complex, concurrency becomes a fundamental topic you'll need to address. **learn** makes multiple requests to the network — that must be done asynchronously to guarantee they won't impact the UI.

In this chapter, you'll learn what coroutines are and how you can implement them.

Concurrency and the Need for Structured Concurrency

Concurrency in programming simply means performing multiple sequences of tasks at the same time. Structured concurrency allows doing multiple computations outside the UI thread to keep the app as responsive as possible. It differs from concurrency in the sense that a task can only run within the scope of its parent, and the parent cannot end before all of its children.

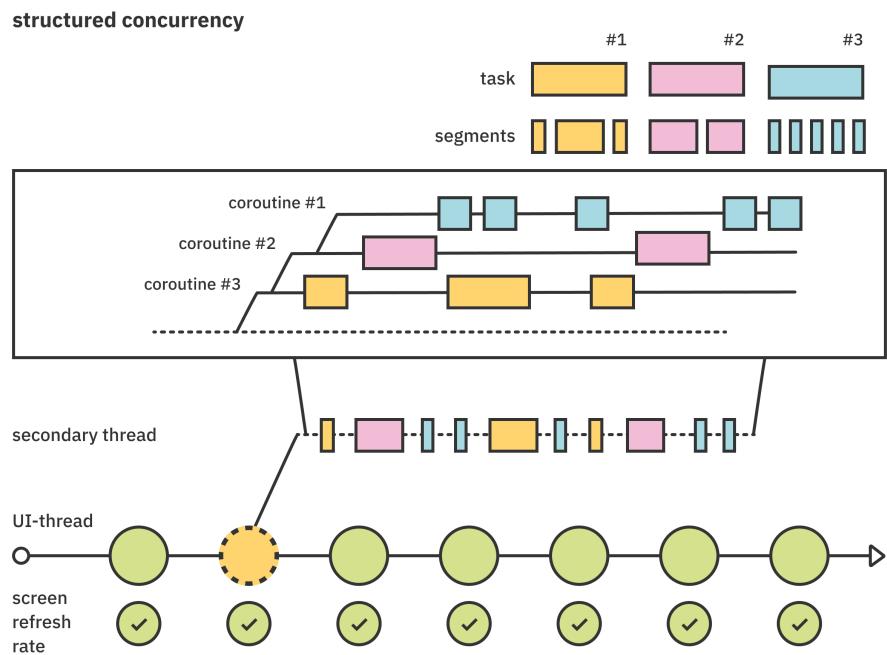


Fig. 13.1 – Diagram showing multiple tasks running in structured concurrency.

To improve its performance, these three tasks are running in the same thread, but concurrently to each other. They were divided into smaller segments that run independently.

Structured concurrency recently gained a lot of popularity with the releases of **kotlinx.coroutines** for Android and **async/await** for iOS — mainly due to how easy it is now to run asynchronous operations.

Different Concurrency Solutions

There are multiple Kotlin Multiplatform libraries that support concurrency:

- **kotlinx.coroutines**: The most popular one. It's lightweight, it allows running multiple coroutines in a single thread and it supports exception handling and cancellation.
- **Reaktive** (<https://github.com/badoo/Reaktive>): An implementation of Reactive Extensions using the **Observable** pattern.
- **CoroutineWorker** (<https://github.com/Autodesk/coroutineworker>): Supports multithreaded coroutines.

In this chapter, you'll learn how to use **kotlinx.coroutines**. Spoiler alert: You've already worked with coroutines before. :]

If you're already familiar with coroutines, you can skip the next few sections and go to "Structured concurrency in iOS", or directly to "Working with `kotlinx.coroutines`", for the next developments in **learn**.

Understanding `kotlinx.coroutines`

Ktor uses coroutines to make network requests without blocking the UI thread, so you've already used them unwittingly in the previous chapter.

Open the `FeedPresenter.kt` file from `shared/commonMain/presentation` and search for `fetchAllFeeds` and `fetchFeed`. In the first function, you've got:

```
for (feed in content) {  
    fetchFeed(feed.platform, feed.image, feed.url, cb)  
}
```

If you weren't using coroutines on `fetchFeed`, these instructions would run sequentially. In other words, the app would only iterate to the next item after `fetchFeed` returned, which would delay the app startup.

Suspend Functions

Suspend functions are at the core of coroutines. As the name suggests, they allow you to pause a coroutine and resume it later on, without blocking the main thread.

Network requests are one of the use cases for **suspend functions**. Open the **FeedAPI.kt** file in **shared/commonMain/data** and look at the function declarations:

```
public suspend fun fetchKodecoEntry(feedUrl: String):  
    HttpResponse = client.get(feedUrl)  
  
public suspend fun fetchMyGravatar(hash: String):  
    GravatarProfile =  
        client.get("$GRAVATAR_URL$hash$GRAVATAR_RESPONSE_FORMAT") {  
            header(X_APP_NAME, APP_NAME)  
        }.body()
```

They're both **suspend functions**. Since a response may take some time, the app cannot block and wait for any of these functions to return.

This image defines the flow that triggers `fetchKodecoEntry` to be called.

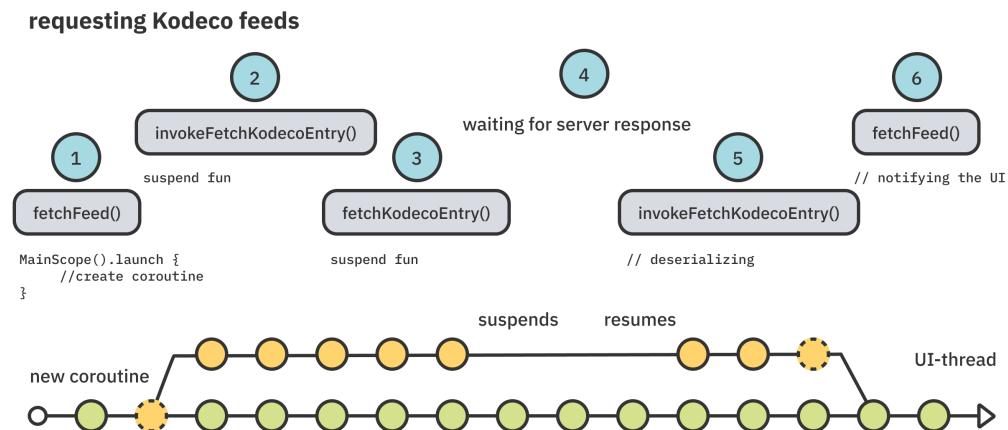


Fig. 13.2 – Diagram showing the different steps of a network request with coroutines.

The entry point, for all platforms, is the `fetchAllFeeds` function from **shared/commonMain/presentation/FeedPresenter.kt**. Once invoked, it iterates over all the RSS feeds, calling `fetchFeed` for each one of its URLs:

1. This is a heavy operation that might block the UI. To avoid this, you'll do it asynchronously. Create a coroutine by calling `launch`.
2. Once launched, it calls `invokeFetchKodecoEntry` from **shared/commonMain/domain/GetFeedData.kt**. A **suspend function** calls the `FeedAPI` to make the request.
3. This function suspends after making the request, and it waits until there's a response or the connection times out.
4. This is done in a separate thread, so the UI doesn't get blocked.
5. Once there's a response, `fetchKodecoEntry` resumes and returns to `invokeFetchKodecoEntry`, which can now deserialize the information received.
6. When this process finishes, the `onSuccess` or the `onFailure` functions execute — depending on the result — and the UI receives an update. Since you're using `MainScope` to launch the coroutine, it will run on the UI thread. You'll see this in detail in the next section.

As a key point, you can only call a **suspend function** from another one or within a coroutine.

Coroutine Scope and Context

Return to `FeedPresenter.kt` from **shared/commonMain/presentation** and search for the `fetchFeed` function:

```
private fun fetchFeed(platform: PLATFORM, imageUrl: String,  
feedUrl: String, cb: FeedData) {  
    MainScope().launch {  
        // Call to invokeFetchKodecoEntry  
    }  
}
```

You already know that `launch` creates a new coroutine, but what's `MainScope`? A **coroutine scope** is where a coroutine will run — in this case, it will be the main thread.

If you open the source code of `MainScope`:

```
public fun MainScope(): CoroutineScope =  
    ContextScope(SupervisorJob() + Dispatchers.Main)
```

You can see that `ContextScope` is built using:

In a `SupervisorJob`, the children behave independently — if one fails, the others won't be affected — whereas in case of `Job`, if a parent fails, all of its children will be canceled.

When you create a coroutine, you have to define the **Dispatcher** where it should run, but you can always switch the context later on during execution by calling `withContext` with the prepended `Dispatcher` as an argument.

In the `fetchMyGravatar` from **FeedPresenter.kt**, you're running the coroutine on the main thread, although the only parts that are necessary to run on the main thread are the `onSuccess` and `onFailure` calls. Update the existing function to use the **IO** thread for the network requests and when the data is available, switch to the **Main** thread so the UI can be updated:

```
public fun fetchMyGravatar(cb: FeedData) {  
    //1  
    CoroutineScope(Dispatchers.IO).launch {  
        //2  
        val profile = feed.invokeGetMyGravatar(  
            hash = GRAVATAR_EMAIL.toByteArray().md5().toString()  
        )  
  
        //3  
        withContext(Dispatchers.Main) {  
            //4  
            cb.onMyGravatarData(profile)  
        }  
    }  
}
```

Here's what you're doing:

1. You create a new coroutine in a thread from the **IO** thread pool and start it.
2. `invokeGetMyGravatar` is a suspend function. When there's a request, it suspends until there's a server response. Once this happens, the coroutine resumes.

3. The UI can only be updated from the UI thread, so it's necessary to switch from the `I0` dispatcher to the `Main` one. This can only be done from within a coroutine.
4. `onMyGravatarData` is now called from the UI thread, so the user can see this newly received data.

You'll also need to update the `invokeGetMyGravatar` function to return the result instead. Open the `GetFeedData.kt` file from `commonMain/domain` and change `invokeGetMyGravatar` to:

```
public suspend fun invokeGetMyGravatar(  
    hash: String,  
) : GravatarEntry {  
    return try {  
        val result = FeedAPI.fetchMyGravatar(hash)  
        Logger.d(TAG, "invokeGetMyGravatar | result=$result")  
  
        if (result.entry.isEmpty()) {  
            GravatarEntry()  
        } else {  
            result.entry[0]  
        }  
    } catch (e: Exception) {  
        Logger.e(TAG, "Unable to fetch my gravatar. Error: $e")  
        GravatarEntry()  
    }  
}
```

In addition to `MainScope`, you also have `GlobalScope`. Typically, it's used in scenarios where the coroutine must live throughout the app execution.

You have to be extra careful when using this function. If the coroutine is unable to finish, it will keep using resources, potentially until the user closes the app.

If you have to update the UI, and you're using `GlobalScope`, you must switch to the UI thread first. Otherwise, when running your iOS app, you'll get the following exception:

```
kotlin.native.IncorrectDereferenceException: illegal attempt to access non-shared (...) from other thread
```

You also have the `coroutineScope` function that allows you to create a coroutine, but it uses the parent scope as context. It has some particularities, namely:

- If the parent gets canceled, it will cancel all of its children.
- Only after all the children end can the parent also terminate.

Coroutine Builders, Scope and Context

You've seen how to start a coroutine by calling `launch`. This function is part of the **Coroutine builders**:

- `runBlocking`: blocks the current thread until the coroutine that it creates ends.

Note: It shouldn't be used inside an existing coroutine, since it will stop its execution.

- `launch`: Creates a coroutine without blocking the current thread. You can define the **CoroutineScope** from where it should run. This scope guarantees structured concurrency — in other words, a coroutine only ends after all of its children have completed their operations.
- `async`: Similar to `launch` in the way it's constructed and how it runs. It differs on its return type in that in this case it's not a **Job**, but a **Deferred<T>** object that will contain the future result of this function.

Return to the `fetchMyGravatar` function and add this function below it:

```
private suspend fun fetchMyGravatar(): GravatarEntry {
    return CoroutineScope(Dispatchers.IO).async {
        feed.invokeGetMyGravatar(
            hash = GRAVATAR_EMAIL.toByteArray().md5().toString()
        )
    }.await()
}
```

This `fetchMyGravatar` is a suspend function. With this approach, you don't need the `onSuccess` and `onFailure` callbacks to update the UI, since you'll return a `GravatarEntry`. You need to call `await` at the end to return its final value instead of a `Deferred<GravatarEntry>`.

It's worth mentioning that this function is similar to using `withContext`:

```
private suspend fun fetchMyGravatar(): GravatarEntry {
    return
    withContext(CoroutineScope(Dispatchers.IO).coroutineContext) {
        feed.invokeGetMyGravatar(
            hash = GRAVATAR_EMAIL.toByteArray().md5().toString()
        )
    }
}
```

The main difference is that `CoroutineScope` doesn't use the same scope as its caller.

Following this approach means that you'll also have to make a few more updates. To use the same logic to notify the UI via callbacks, you'll need to change `fetchMyGravatar(cb: FeedData)` to:

```
public fun fetchMyGravatar(cb: FeedData) {
    Logger.d(TAG, "fetchMyGravatar")

    CoroutineScope(Dispatchers.IO).launch {
        cb.onMyGravatarData(fetchMyGravatar())
    }
}
```

Otherwise, you can return the `GravatarEntry` directly to the UI. You'll see how to implement this second approach in the “Creating a Coroutine With Async” section.

Cancelling a Coroutine

Although you're not going to use it in `learn`, it's worth mentioning that you can cancel a coroutine by calling `cancel()` on the `Job` object returned by `launch`.

In case you're using `async`, you'll have to implement a solution similar to this one:

```
val deferred = CoroutineScope(Dispatchers.IO).async {
    feed.invokeGetMyGravatar(
        hash = GRAVATAR_EMAIL.toByteArray().md5().toString()
    )
}

//If you want to cancel
deferred.cancel()
```

When you cancel a coroutine, a `CancellationException` is thrown silently. You can catch it to implement a specific behavior your app might need, or to clean up resources.

Structured Concurrency in iOS

Apple has a similar solution for structured concurrency: **async/await**.

Note: async/await is only available if you're running your app on iOS 13 or newer versions.

With **async/await**, you no longer need to use completion handlers. Instead, you can use the **async** keyword after the function declaration. If you want to wait for it to return, add **await** before calling the **suspend function**:

```
private func fetchMyGravatar() async -> GravatarEntry {
    return await feed.invokeGetMyGravatar(
        hash = GRAVATAR_EMAIL.toByteArray().md5().toString()
    )
}
```

Which in Kotlin is similar to:

```
private suspend fun fetchMyGravatar(): GravatarEntry {
    return withContext(Dispatchers.IO) {
        feed.invokeGetMyGravatar(
            hash = GRAVATAR_EMAIL.toByteArray().md5().toString()
        )
    }
}
```

Following the same logic as **suspend functions**, you can only call an **async function** from another one or from an asynchronous task. In Kotlin, this corresponds to calling the function from a coroutine.

Swift uses Task. Using Task, the previous example can be translated to:

```
private func fetchMyGravatar() {
    Task {
        let profile = await feed.invokeGetMyGravatar(
            hash = GRAVATAR_EMAIL.toByteArray().md5().toString()
        )
        await profile
    }
}
```

With **kotlinx.coroutines**, it's:

```
private suspend fun fetchMyGravatar() = {
    CoroutineScope(Dispatchers.IO).launch {
        async { feed.invokeGetMyGravatar(
            hash = GRAVATAR_EMAIL.toByteArray().md5().toString()
        ) }.await()
    }
}
```

Using kotlinx.coroutines

It's time to update **learn**. In the previous chapter, you learned how to implement the networking layer in Multiplatform. For this, you added the **Ktor** library and wrote the logic to fetch the kodeco.com RSS feed and parse its responses that later update the UI.

However, there's a little detail that was left for this section: **Ktor** is built using **kotlinx.coroutines**. This is why the `MainScope`, `launch` and `suspend` functions seemed familiar in the “Understanding `kotlinx.coroutines`” section.

Adding kotlinx.coroutines to Your Gradle Configuration

Since **Ktor** includes the **kotlinx.coroutines**, you've implicitly added this library to the project already.

Otherwise, if you want to include **kotlinx.coroutines** in your projects, you'll need to add:

```
implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.7.3")
```

Troubleshooting `kotlinx.coroutines` in iOS

As you continue your journey with Multiplatform outside this book, you'll probably find this error:

```
Uncaught Kotlin exception:  
kotlin.native.concurrent.InvalidMutabilityException: mutation attempt of  
frozen
```

This `InvalidMutabilityException` means you're accessing an object that belongs to another thread, which is currently not possible. Confirm if you're using the latest version of Kotlin and that you're using the `Dispatchers.Main` to access that object.

If you're still running into problems:

- Delete the **build** folder in the root directory of the project.
- Delete the **build** folder in the **shared** directory in the root directory of the project.

Frozen State

In some instances, you might need to freeze your objects when running your iOS app to avoid having the error mentioned above. Once `freeze()` is called over an object, it becomes immutable. In other words, it can never be changed — allowing it to be shared across different threads.

Another advantage of using the `kotlinx.coroutines` library is that this logic is already built into the latest version of the library, so you shouldn't need to do anything from your side.

Working With `kotlinx.coroutines`

In the app, go to the **latest** screen. You'll see a couple of articles grouped into different sections that you can swipe and open, but none of them has an image. It's time to change this!

Creating a Suspend Function

Start by opening the **FeedAPI.kt** file from **commonMain/data** in the **shared** module.

After the `fetchKodecoEntry`, add:

```
public suspend fun fetchImageUrlFromLink(link: String):  
    HttpResponse = client.get(link) {  
        header(HttpHeaders.Accept, ContentType.Text.Html)  
    }
```

When prompted, import:

```
import io.ktor.http.HttpHeaders  
import io.ktor.http.ContentType
```

`fetchImageUrlFromLink` receives the link from an article and returns the page source code as the `HttpResponse`. It's set as a suspend function, so the current thread won't block while it's waiting for the server response.

Note: You need to set the `Accept` header in this request otherwise the server will return a 406, not acceptable.

Next, open the **GetFeedData.kt** file from **shared/commonMain/domain** and add the following method inside the class:

```
//1  
public suspend fun invokeFetchImageUrlFromLink(  
    link: String,  
    //2  
    onSuccess: (String) -> Unit,  
    onFailure: (Exception) -> Unit  
) {  
    try {  
  
        //3  
        val result = FeedAPI.fetchImageUrlFromLink(link)  
        //4  
        val url = parsePage(link, result.bodyAsText())  
  
        //5  
        coroutineScope {  
            onSuccess(url)  
        }  
    } catch (e: Exception) {
```

```
        coroutineScope {  
            onFailure(e)  
        }  
    }  
}
```

Here's a step-by-step breakdown of this logic:

1. `invokeFetchImageUrlFromLink` is set as `suspend` since it will call the `FeedAPI` to retrieve the page source code.
2. The `onSuccess` and `onFailure` functions define how this function should behave, depending on whether it was possible to retrieve an image for the article or not.
3. The `FeedAPI` uses the **Ktor HttpClient** to make a network request.
4. Since there's no API to get the URL for the image, you're going to parse the HTML code and look for a specific image tag. Along with the network request, this will be a heavy task. So, this logic needs to be called from a coroutine.
5. The `coroutineScope` creates a new coroutine, using its parent scope to run the functions of `onSuccess` or `onFailure` depending on whether the operation succeeded or not.

In the next sections, you'll see different approaches to creating and starting a coroutine. Although both of them are valid, the APIs that they expose to the UI are different.

Note: When multiple teams will use a **shared** module, it's best for representatives from each team to discuss and agree on conventions they all feel comfortable following. This is especially important for iOS programmers who are new to Kotlin and can feel overwhelmed having to adapt to a new language. Interacting with your shared module should be similar to any other library that exists for iOS.

Creating a Coroutine With `launch`

Now that you've implemented the functions for requesting and parsing data, you're just missing creating a coroutine, and it's... `launch`. :]

Open the **FeedPresenter.kt** file inside **commonMain/presentation**. In the **shared** module and before the `fetchMyGravatar(cb: FeedData)` function, add:

```
public fun fetchLinkImage(platform: PLATFORM, id: String, link: String, cb: FeedData) {
    CoroutineScope(Dispatchers.IO).launch {
        feed.invokeFetchImageUrlFromLink(
            link,
            onSuccess = { cb.onNewImageUrlAvailable(id, it, platform, null) },
            onFailure = { cb.onNewImageUrlAvailable(id, "", platform, it) }
        )
    }
}
```

In this approach, you're using a `FeedData` listener that's defined at the UI level. Once the `invokeFetchImageUrlFrom` finishes, it will either call the `onSuccess` or `onFailure` functions that in their turn will call the `onNewImageUrlAvailable` callback at the UI with the new data received or with an exception in case there was an error.

Now, connect your app's UI to this new function.

In **androidApp** and **desktopApp**, the changes are similar. In both projects, go to **ui/home**, open the **FeedViewModel.kt** file, and update the `onNewImageUrlAvailable` callback with:

```
override fun onNewImageUrlAvailable(id: String, url: String, platform: PLATFORM, exception: Exception?) {
    Logger.d(TAG, "onNewImageUrlAvailable | platform=$platform | id=$id | url=$url")
    viewModelScope.launch {
        val item = _items[platform]?.firstOrNull { it.id == id } ?: return@launch
        val list = _items[platform]?.toMutableList() ?: return@launch
        val index = list.indexOf(item)

        list[index] = item.copy(imageUrl = url)
        _items[platform] = list
    }
}
```

When this method receives a new `url`, the item to which it corresponds is updated. Updating the `_items` map automatically updates the UI.

Note: `viewModelScope` runs on the UI-thread.

Inside the `viewModelScope.launch` of `onNewDataAvailable`, replace the existing code with:

```
_items[platform] = if (items.size > FETCH_N_IMAGES) {
    items.subList(0, FETCH_N_IMAGES)
} else{
    items
}

for (item in _items[platform]!!) {
    fetchLinkImage(platform, item.id, item.link)
}
```

Now, when the app receives new articles, it will automatically request its images.

Create the `fetchLinkImage` function:

```
private fun fetchLinkImage(platform: PLATFORM, id: String, link: String) {
    Logger.d(TAG, "fetchLinkImage | link=$link")
    presenter.fetchLinkImage(platform, id, link, this)
}
```

`fetchLinkImage` calls the `fetchLinkImage` from the **FeedPresenter.kt** file that you created earlier.

In the **iosApp**, open the **FeedClient.swift** file that's inside the **extensions** directory and search for `fetchLinkImage`. To also call the `fetchLinkImage` from the **FeedPresenter.kt** class, update this function to:

```
public func fetchLinkImage(_ platform: PLATFORM, _ id: String, _ link: String, completion: @escaping FeedHandlerImage) {
    feedPresenter.fetchLinkImage(platform: platform, id: id, link: link, cb: self)
    handlerImage = completion
}
```

Now, navigate to **KodecoEntryViewModel.swift** and, similar to what you've done on the other platforms, create the `fetchLinkImage`:

```
func fetchLinkImage() {
    for platform in self.items.keys {
        guard let items = self.items[platform] else { continue }
        let subsetItems = Array(items[0 ..<
```

```

Swift.min(self.fetchNImages, items.count]))
    for item in subsetItems {
        FeedClient.shared.fetchLinkImage(item.platform, item.id,
item.link) { id, url, platform in
            guard let item =
self.items[platform.description]?.first(where: { $0.id == id })
else {
                return
            }

            guard var list = self.items[platform.description] else {
                return
            }
            guard let index = list.firstIndex(of: item) else {
                return
            }

            list[index] = item.doCopy(
                id: item.id,
                link: item.link,
                title: item.title,
                summary: item.summary,
                updated: item.updated,
                platform: item.platform,
                imageUrl: url,
                bookmarked: item.bookmarked
            )

            Logger().d(tag: TAG, message: "\(list[index].title)Updated
to:\(list[index].imageUrl)")

            self.items[platform.description] = list
        }
    }
}
}

```

Finally, call it from the existing the `fetchFeeds` function as follows:

```

func fetchFeeds() {
    FeedClient.shared.fetchFeeds { platform, items in
        Logger().d(tag: TAG, message: "fetchFeeds: \(items.count)
items | platform: \(platform)")
        DispatchQueue.main.async {
            self.items[platform] = items
            self.fetchLinkImage()
        }
    }
}

```

Compile and run the apps for the three platforms and navigate to the **Latest** screen.



Fig. 13.3 – Android App: Browse Through the Latest Articles

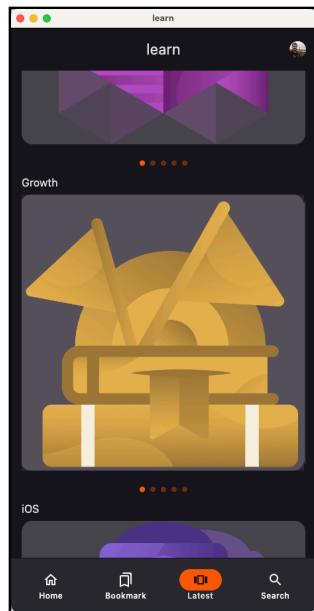


Fig. 13.4 – Desktop App: Browse Through the Latest Articles



Fig. 13.5 – iOS App: Browse Through the Latest Articles

Creating a Coroutine With `Async`

As an alternative to the previous approach where you’re using callbacks to notify the UI when new data is available, you can suspend the `fetchLinkImage` function until there’s a final result. For that, you’ll need to use `async` instead of `launch`.

Return to the `FeedPresenter.kt` file in `commonMain/presentation` in the `shared` module, and update the function `fetchLinkImage`:

```
public suspend fun fetchLinkImage(link: String): String {
    return CoroutineScope(Dispatchers.IO).async {
        feed.invokeFetchImageUrlFromLink(
            link
        )
    }.await()
```

As you can see, it’s no longer necessary to have the `platform` and `id` parameters, since you’re going to return the image URL in case it exists. The `async` function allows returning an object while `await` waits for the response to be ready. Instead of returning a `Deferred<T>` – in this case it would be a `Deferred<String?>`.

Depending on the Android Studio version you're using, it's probable that it would suggest you replace the previous implementation with:

```
public suspend fun fetchLinkImage(link: String): String {
    return
    withContext(CoroutineScope(Dispatchers.IO).coroutineContext) {
        feed.invokeFetchImageUrlFromLink(
            link
        )
    }
}
```

Both approaches produce similar results, but they're quite different under the hood.

You can remove the `onNewImageUrlAvailable` from the **FeedData.kt** interface, located in the **domain/cb** directory.

Open **GetFeedData.kt** and update `invokeFetchImageUrlFromLink` to the following:

```
public suspend fun invokeFetchImageUrlFromLink(
    link: String
): String {
    return try {
        val result = FeedAPI.fetchImageUrlFromLink(link)
        parsePage(link, result.bodyAsText())
    } catch (e: Exception) {
        ""
    }
}
```

Now it's time to update the UI! You'll need to change how you're calling the `fetchLinkImage` function:

- On both **androidApp** and **desktopApp**, go to the **FeedViewModel.kt** file inside **ui/home**, and replace the existing `fetchLinkImage` function with:

```
private fun fetchLinkImage(platform: PLATFORM, id: String, link: String) {
    Logger.d(TAG, "fetchLinkImage | link=$link")
    viewModelScope.launch {
        val url = presenter.fetchLinkImage(link)

        val item = _items[platform]?.firstOrNull { it.id == id } ?: return@launch
        val list = _items[platform]?.toMutableList() ?: return@launch
        val index = list.indexOf(item)
```

```
        list[index] = item.copy(imageUrl = url)
    }
}
```

This is the code that was in `onNewImageUrlAvailable`, along with the call to `presenter.fetchLinkImage`. Since you no longer use that callback, you can remove it.

- For **iOSApp**, you also need to update the **FeedClient.swift** file, which is inside the **extensions** folder. Start by updating the `FeedHandlerImage` that no longer has to receive all of its parameters:

```
public typealias FeedHandlerImage = (_ url: String) -> Void
```

Update the `fetchLinkImage` to:

```
@MainActor
public func fetchLinkImage(_ link: String, completion: @escaping
FeedHandlerImage) {
    Task {
        do {
            let result = try await feedPresenter.fetchLinkImage(link:
link)
            completion(result)
        } catch {
            Logger().e(tag: TAG, message: "Unable to fetch article
image link")
        }
    }
}
```

Since you're now accessing a **suspend function** from Swift, you'll have to use **await** to wait for the result to be available. The `@MainActor` annotation guarantees the Task runs on the UI thread. Otherwise, you might have a `InvalidMutabilityException`.

Now, remove the `onNewImageUrlAvailable` from the `FeedClient` extension at the bottom of the file since this callback no longer exists.

Because this function needs to be declared as `@MainActor` and the `id`, `platform` and `cb` are no longer necessary, you have to update the `fetchLinkImage` method from `KodecoEntryViewModel.swift`:

```
@MainActor
func fetchLinkImage() {
    for platform in self.items.keys {
        guard let items = self.items[platform] else { continue }
        let subsetItems = Array(items[0 ..<
            Swift.min(self.fetchNImages, items.count)])
        for item in subsetItems {
            FeedClient.shared.fetchLinkImage(item.link) { url in
                guard var list = self.items[platform.description] else {
                    return
                }
                guard let index = list.firstIndex(of: item) else {
                    return
                }

                list[index] = item.doCopy(
                    id: item.id,
                    link: item.link,
                    title: item.title,
                    summary: item.summary,
                    updated: item.updated,
                    platform: item.platform,
                    imageUrl: url,
                    bookmarked: item.bookmarked
                )
                self.items[platform.description] = list
            }
        }
    }
}
```

Compile and run your app, and browse through the outstanding artwork of the Kodeco articles. :]



Fig. 13.6 – Android App: Browse Through the Latest Articles

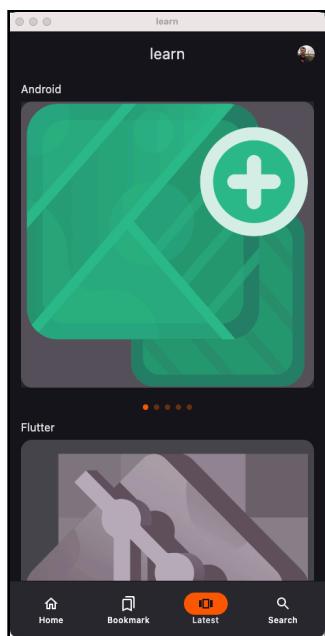


Fig. 13.7 – Desktop App: Browse Through the Latest Articles

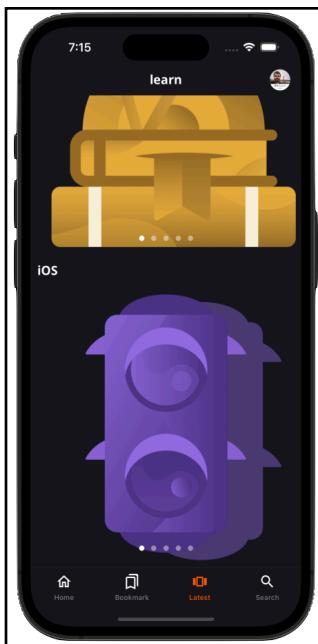


Fig. 13.8 – iOS App: Browse Through the Latest Articles

Improving Coroutines Usage for Native Targets

A key point when showing the benefits of using Kotlin Multiplatform on multiple targets is to keep the developer experience as close to using the platform language and tools as possible. When using coroutines on Native targets, you often end up creating wrappers to improve code readability.

The following libraries were developed by the community to improve coroutines integration with Swift:

- KMP-NativeCoroutines (<https://github.com/rickclephas/KMP-NativeCoroutines>): allows canceling an existing coroutine from Swift and adds support to **Flow** without losing any property. It's recommended by JetBrains and it gets continuous updates.
- Koru (<https://github.com/FutureMind/koru>): supports generating wrappers for **suspend** functions and **Flow**.

- SKIE (<https://skie.touchlab.co/intro>): has the primary goal of making it seamless to use Kotlin from Swift by providing support for **suspend** functions, **Flow** and default arguments. It also makes it easier to use Kotlin enums, sealed classes and interfaces.

Configuring KMP NativeCoroutines

To use this library, you need to add it first to the **shared** module and then to the **iOSApp** via the Swift Package Manager. Let's start by opening the **libs.versions.toml** file located inside the **gradle** folder. In the **[versions]** section define the library versions that you're going to use:

```
ksp = "1.9.10-1.0.13"  
nativeCoroutines = "1.0.0-ALPHA-18"
```

Then scroll down to **[plugins]** and add:

```
google-ksp = { id = "com.google.devtools.ksp", version.ref =  
  "ksp" }  
kmp-NativeCoroutines = { id =  
  "com.rickclephas.kmp.nativecoroutines", version.ref =  
  "nativeCoroutines" }
```

KSP (Kotlin Symbol Processing API) is required by KMP NativeCoroutines.

Now, open the **build.gradle.kts** file in the root folder and update the **plugins** sections with these new ones:

```
alias(libs.plugins.google.ksp) apply false  
alias(libs.plugins.kmp.nativeCoroutines) apply false
```

shared will use this library, so you also need to set the plugin on its **build.gradle.kts** file. In the **plugins** block, add:

```
alias(libs.plugins.google.ksp)  
alias(libs.plugins.kmp.nativeCoroutines)
```

To generate the wrappers for Native, you'll need to opt in to use the annotation **ObjCName**, otherwise, you'll keep seeing a warning when compiling your project. Scroll down to the end of the file and after the existing **kotlin.RequiresOptIn** add a second one:

```
languageSettings.optIn("kotlin.experimental.ExperimentalObjCName")
```

Synchronize the project.

Open Xcode and go to **File > Add Package Dependencies...** and in the top-right corner where it says, “Search or Enter Package URL” enter:

```
https://github.com/rickclephas/KMP-NativeCoroutines.git
```

Click **Add Package** to add it to your app.

That's it! All libraries are set.

Using KMP NativeCoroutines With a Suspend Function

Now that all libraries are set, it's time to update your code. Return to Android Studio and open the **FeedPresenter.kt** file located in the **shared** module.

KMP NativeCoroutines, has two annotations that you can use:

- **@NativeCoroutineScope**: it defines the scope to use, and allows you to have more control over it.
- **@NativeCoroutines**: functions that use this are not generated for Objective-C, but instead, an extension is created that can more easily be accessed from Swift. This is generated via the KSP plugin that you just added.

You can open the **SharedKit.framework** directly from Xcode by clicking on `import SharedKit` or any function from it in any Swift file that uses it, or alternatively, you can go to **shared/build/bin/ios*Arm64/debugFramework/Headers/SharedKit.h**.

Look for the `fetchMyGravatar` declaration. Here you're going to find two of them, one that receives a completion handler and another one that receives a callback.

```
- (void)fetchMyGravatarCb:(id<SharedKitFeedData>)cb
__attribute__((swift_name("fetchMyGravatar(cb:)")));

- (void)fetchMyGravatarWithCompletionHandler:(void (^)
(SharedKitGravatarEntry * _Nullable, NSError * _Nullable))completionHandler
__attribute__((swift_name("fetchMyGravatar(completionHandler:)"))
);
```

To use these functions from Swift, you need to create a handler and an extension for retrieving data, as you can see in **FeedClient.swift**. With the KMP NativeCoroutines library, the generated code can be improved, making it easier and more friendly to access. For that, return to the **FeedPresenter.kt** file and update `fetchMyGravatar(): GravatarEntry`:

```
@NativeCoroutines
public suspend fun fetchMyGravatar(): GravatarEntry {
    return CoroutineScope(Dispatchers.IO).async {
        feed.invokeGetMyGravatar(
            hash = GRAVATAR_EMAIL.toByteArray().md5().toString()
        )
    }.await()
}
```

It's now set as `public` and has the `@NativeCoroutines` annotation set.

Compile the project and open the **SharedKit.framework** again. You'll see that there's no longer a `fetchMyGravatarWithCompletionHandler` function, but instead, you've got an interface with `fetchMyGravatar` declared, that you can easily call from Swift:

```
@interface SharedKitFeedPresenter : NSObject
- (SharedKitKotlinUnit * (^)(SharedKitKotlinUnit *^,
                           SharedKitGravatarEntry *, SharedKitKotlinUnit *),
    SharedKitKotlinUnit * (^)(NSError *, SharedKitKotlinUnit *),
    SharedKitKotlinUnit * (^)(NSError *, SharedKitKotlinUnit *))  

(void)fetchMyGravatar  

__attribute__((swift_name("fetchMyGravatar())));
@end
```

Now open the **FeedClient.swift** file and import the `KMPNativeCoroutinesAsync` library:

```
import KMPNativeCoroutinesAsync
```

Afterward, update the `fetchProfile` function to:

```
//1
public func fetchProfile() async -> GravatarEntry? {
//2
    let result = await asyncResult(for:
feedPresenter.fetchMyGravatar())
    switch result {
//3
    case .success(let value):
        return value
    case .failure(let value):
```

```
    Logger().e(tag: TAG, message: "Unable to fetch profile.  
Reason:\(value)")  
    return nil  
}  
}
```

Here's a step-by-step breakdown of this logic:

1. You're going to access a function that accesses the internet to retrieve the user's Gravatar information. To accomplish this, `fetchMyGravatar` is set as a **suspend** function and returns a `GravatarEntry`. This is an asynchronous operation so `fetchMyProfile` declaration needs to reflect that, this is why it's now set as `async`.
2. `asyncResult` is a wrapper from `KMPNativeCoroutinesAsync` that returns the result of a function along with the operation state: `.success` or `.failure`.
3. If the call is successful and new data is available, its content is returned. Otherwise, a log message is printed, and the result is `nil`.

With this change, you can now safely remove the `ProfileHandler` declaration and all of its usage.

Since the behavior of this function changed, you also need to update its caller; otherwise, the project won't compile. Open the `KodecoEntryViewModel.kt` file and look for `fetchProfile`, update it to reflect these changes:

```
func fetchProfile() {  
    //1  
    Task {  
        //2  
        guard let profile = await FeedClient.shared.fetchProfile()  
        else { return }  
        DispatchQueue.main.async {  
            self.profile = profile  
        }  
    }  
}
```

Let's break this code snippet into parts:

1. Since `fetchProfile` is asynchronous, to avoid blocking the main thread, this call will be executed inside a `Task`.
2. If the result is `.success`, the main thread is resumed and the `self.profile` updated. Otherwise, the `Task` ends.

All done! Compile and run the project.



Fig. 13.9 – iOS App: Browse Through the Home Screen

Using KMP NativeCoroutines With Flow

None of the functions in **FeedPresenter.kt** uses **Flow**, so the first step is to update the `fetchAllFeeds` function:

```
//1
@NativeCoroutines
//2
public fun fetchAllFeeds(): Flow<List<KodecoEntry>> {
    Logger.d(TAG, "fetchAllFeeds")

    //3
    return flow {
        for (feed in content) {
            //4
            emit(
                fetchFeed(feed.platform, feed.image, feed.url)
            )
        }
    //5
    }
}
```

```
    }.flowOn(Dispatchers.IO)
}
```

There are a couple of changes to this function, so let's go over them step-by-step:

- Similar to before, the use of the `@NativeCoroutines` annotation allows generating an interface with the `fetchAllFeeds` function which is easier and more friendly to call from Swift:

```
@interface SharedKitFeedPresenter (Extensions)
- (SharedKitKotlinUnit * (^)(^(SharedKitKotlinUnit *^)
(SharedKitGravatarEntry *, SharedKitKotlinUnit *),
SharedKitKotlinUnit * (^)(NSError *, SharedKitKotlinUnit *),
SharedKitKotlinUnit * (^)(NSError *, SharedKitKotlinUnit *)))
(void)fetchMyGravatar
__attribute__((swift_name("fetchMyGravatar())));
@end
```

- Previously, `fetchAllFeeds` would receive a callback, which on Native would be translated to a completion handler. The first step is to remove this argument, and instead return a `Flow` with the list of all the `KodecoEntry`. Ideally, it would return a `Map<PLATFORM, List<KodecoEntry>>`, but this is currently not possible with the current version of KMP NativeCoroutines.
- The `Flow` that your function will return.
- For each `Kodeco` topic: All, Android, iOS, Flutter, Server, GameTech, and Growth, you're going to retrieve its RSS feed. Once this data is available, it will be emitted automatically, to whoever is listening to it. In this case, it will be `FeedViewModel.kt` on Android and Desktop and `KodecoEntryViewModel.swift` on iOS.
- Finally, this will run on the `I0` dispatcher.

With the removal of the `FeedData` callback from `fetchAllFeeds` you need to do the same thing on the `fetchFeed` function called in step 4. Update it to:

```
private suspend fun fetchFeed(
    platform: PLATFORM,
    imageUrl: String,
    feedUrl: String,
): List<KodecoEntry> {
    return CoroutineScope(Dispatchers.I0).async {
        feed.invokeFetchKodecoEntry(
            platform = platform,
            imageUrl = imageUrl,
            feedUrl = feedUrl
        )
    }
}
```

```
        )  
    }.await()  
}
```

Now, instead of receiving cb it returns the list of KodecoEntry that corresponds to the RSS feed for a specific topic. This change, also requires that invokeFetchKodecoEntry be modified. Open the **GetFeedData.kt** file and update this function to:

```
public suspend fun invokeFetchKodecoEntry(  
    platform: PLATFORM,  
    imageUrl: String,  
    feedUrl: String  
): List<KodecoEntry> {  
    return try {  
        val result = FeedAPI.fetchKodecoEntry(feedUrl)  
  
        Logger.d(TAG, "invokeFetchKodecoEntry | feedUrl=$feedUrl")  
  
        val xml = Xml.parse(result.bodyAsText())  
  
        val feed = mutableListOf<KodecoEntry>()  
        for (node in xml.allNodeChildren) {  
            val parsed = parseNode(platform, imageUrl, node)  
  
            if (parsed != null) {  
                feed += parsed  
            }  
        }  
  
        feed  
    } catch (e: Exception) {  
        Logger.e(TAG, "Unable to fetch feed:$feedUrl. Error: $e")  
        emptyList()  
    }  
}
```

Similarly, as before, instead of calling the cb with the result of the network requests, it will return a list of RSS feeds in case it successfully received and parsed them or an empty list in case any of these operations fail.

Since you're now returning the data directly, you can remove the onNewDataAvailable function from **FeedData.kt**.

With the **shared** logic updated, it's now time to update the apps. Starting with Android, open the **FeedViewModel.kt** file and replace the existing `fetchAllFeeds` function to:

```
fun fetchAllFeeds() {
    Logger.d(TAG, "fetchAllFeeds")
    viewModelScope.launch {
        presenter.fetchAllFeeds().collect {
            val platform = it.first().platform
            _items[platform] = it

            for (item in _items[platform]!!) {
                fetchLinkImage(platform, item.id, item.link)
            }
        }
    }
}
```

And remove the `onNewDataAvailable` function. Now that you've got the Android app ready, repeat both steps for Desktop.

Build and run to see the result of the refactoring that you just did.

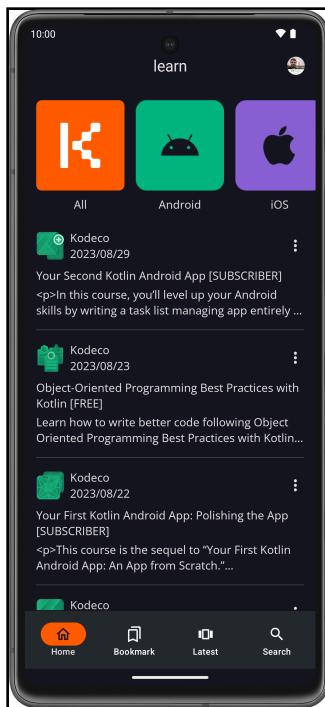


Fig. 13.10 – Android App: Browse Through the Home Screen

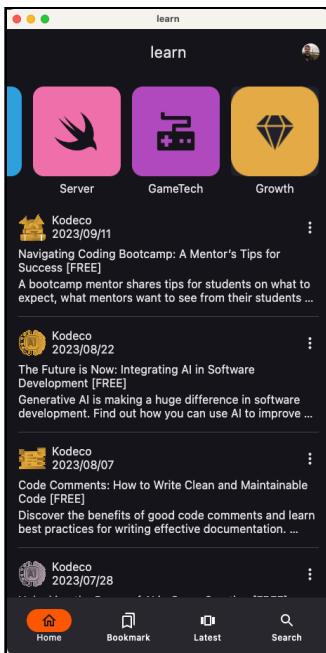


Fig. 13.11 — Desktop App: Browse Through the Home Screen

Return to Xcode and open the **FeedClient.swift** file. Similarly, to what you've changed in the previous section, you need to update the `fetchFeeds` function to return a dictionary with the platform name and the list of `KodecoEntry`:

```
public func fetchFeeds() async -> [String: [KodecoEntry]] {
    var items: [String: [KodecoEntry]] = [:]
    do {
        let result = asyncSequence(for:
feedPresenter.fetchAllFeeds())
        for try await data in result {
            guard let item = data.first else { continue }
            items[item.platform.name] = data
        }
    } catch {
        Logger().e(tag: TAG, message: "Unable to fetch all feeds")
    }
    return items
}
```

1. `KMPNativeCoroutinesAsync` has different functions depending on the type of data that you're going to access. For `Flow` you need to use the `asyncSequence` that allows you to collect the values from it.
2. Since it's an asynchronous operation, you need to wait for it to return, or in other words, to `emit` (from `fetchAllFeeds` on `FeedPresenter.kt`) the data that you're expecting.
3. If data is valid, you're going to update the current `items` list, otherwise it's discarded.

Remove the `onNewDataAvailable` extension function and the `FeedHandler` declaration and usage, which are no longer necessary.

Finally, open the `KodecoEntryViewModel.swift` and update the `fetchFeeds` function:

```
func fetchFeeds() {
    Task {
        let test = await FeedClient.shared.fetchFeeds()
        DispatchQueue.main.async {
            self.items = test
            self.fetchLinkImage()
        }
    }
}
```

It creates a task to avoid blocking the main thread. Once new data is available it returns to it and updates `self.items` which in turn notifies the app that there's new content to update. After the RSS feed is received, the app fetches its corresponding images.

Compile and run the app.



Fig. 13.12 – iOS App: Browse Through the Home Screen

Challenge

Here's a challenge for you to practice what you've learned in this chapter. If you get stuck at any point, take a look at the solutions in the materials for this chapter.

Challenge: Fetch the Article Images From the Shared Module

Instead of requesting the article images from the UI, move this logic to the shared module.

Remember that you don't need to run this logic sequentially — you can launch multiple coroutines to fetch and parse the response, making this operation faster.

The requests should run in parallel.

Key Points

- A **suspend function** can only be called from another **suspend function** or from a coroutine.
- You can use `launch` or `async` to create and start a coroutine.
- A coroutine can start a thread from **Main**, **IO** or **Default** thread pools.
- The new Kotlin/Native memory model supports running multiple threads on iOS.

Where to Go From Here?

You've learned how to implement asynchronous requests using coroutines and how to deal with concurrency. If you want to dive deeper into this subject, try the Kotlin Coroutines by Tutorials (<https://www.kodeco.com/books/kotlin-coroutines-by-tutorials>) book, where you can read in more detail about Coroutines, Channels and Flows in Android. There's also Concurrency by Tutorials (<https://www.kodeco.com/books/concurrency-by-tutorials>), which focuses on multithreading in Swift, and Modern Concurrency in Swift (<https://www.kodeco.com/books/modern-concurrency-in-swift>), which teaches you the new concurrency model with `async/await` syntax.

In the next chapter, you'll learn how to migrate a feature to support Kotlin Multiplatform and release your libraries so that you can later reuse them in your projects.

Chapter 14: Creating Your KMP Library

By Carlos Mota

In the previous chapters, you've built the **learn** app for Android, iOS and desktop. All of these apps fetch the [kodeco.com](#) RSS feed and show you the latest articles written about Android, iOS, Flutter, Server-side Swift, Unity, and professional growth. You can search for a specific topic or save an article locally to read it later. During the app's development process, you've worked with:

- Serialization
- Networking
- Databases
- Concurrency

And, along this journey, you've also built additional tools that are useful and can be reused in other projects:

- Logger
- Parcelize support

In this chapter, you're going to learn how you can create and publish a library so you can reuse it in the other apps that you develop in this book — and for the next one you're going to build. :]

Migrating an Existing Feature to Multiplatform

Throughout this book, you've learned how to develop a project that had a library already shared across different platforms. However, you may want to migrate an existing app to Kotlin Multiplatform.

In this section, you're going to see how a simple feature like opening a website link in a browser can easily be moved to KMP.

Learning How to Open a Link on Different Platforms

In **learn**, when you click on an article, a web page opens — whether it's on Android, iOS or desktop. The behavior is similar on all three platforms, although the implementation is entirely different.

In **Android**, a prompt is shown so you can select which app it should use to send the **Intent**. Or, if you have one already set as default, it will automatically open it and load the article you've clicked on. **MainActivity.kt**, in **androidApp/ui**, defines this function:

```
private fun openEntry(url: String) {
    val intent = Intent(Intent.ACTION_VIEW)
    intent.data = Uri.parse(url)
    startActivity(intent)
}
```

To ensure the right app opens a URL, Android filters installed apps based on specific criteria. It looks for apps with the **ACTION_VIEW** attribute defined in their **AndroidManifest.xml** and ones capable of parsing **URIs**.

iOS has a different approach. To open a URL, you just need to use the **OpenURLAction** from the environment. Open **LatestView.swift** from **iosApp** module and scroll to the **Section** struct:

```
@Environment(\.openURL)
var openURL
```

The `var openURL` allows you to send a URL that will open the default browser on your device:

```
openURL(URL(string: "\$(item.link)")!)
```

When the user clicks on one of the articles, the app creates a URL from that item link and calls `openURL` with it to open the link in the browser.

`desktopApp` uses another approach. Desktop has a `browse` function that you can use to launch the default browser on your computer.

Open the `Main.kt` from the `desktopApp` module and scroll down to the end of this file:

```
fun openEntry(url: String) {
    try {
        val desktop = Desktop.getDesktop()
        desktop.browse(URI.create(url))
    } catch(e: Exception) {
        Logger.e(TAG, "Unable to open url. Reason: ${e.stackTrace}")
    }
}
```

The `getDestkop` call returns an instance of `Desktop` that contains its context as well as a couple of functions that let you access some of your computer's features — like open and edit files, browser, mail, print, and more. Here, you're using `browse` to open your default browser with the `url` from the item that you click on.

The `try... catch` block is necessary — on some platforms, the desktop API might not be available. This might lead to unwanted behaviors. Following this approach guarantees that in the worst case, although the app won't open a link, it also won't crash.

Note: Alternatively, you could also use `isDesktopSupported` to check if the desktop API is available. In any case, be careful, because calling `browse` might trigger an `IOException`.

Now that you're familiar with how the three platforms open a URL, it's time to move this logic to KMP.

Adding a New Module

The first thing to decide is if you want to move this logic to the existing **shared** module or create a new one. Since adding a new library also requires you to migrate the code, you're going with this more complete solution.

The first step is to add a new **Kotlin Multiplatform Shared Module**. Go to **File ▶ New ▶ New Module...**, and select the **Kotlin Multiplatform Shared Module** template at the bottom of the list. Here, define the:

- **Module Name:** shared-action
- **Package Name:** com.kodeco.learn.action
- **iOS framework distribution:** XCFramework

Click **Finish** and wait for the project to synchronize.

If you look at the Android Studio **Project** tab, you'll see a new **shared-action** module added.

Note: In case you get an error stating ‘Unresolved reference: kotlinMultiplatform’, open the **shared-action** module’s **build.gradle.kts** and replace the **plugins** section with the following code:

```
plugins {  
    alias(libs.plugins.jetbrains.kotlin.multiplatform)  
    alias(libs.plugins.android.library)  
}
```

Additionally, remove the **commonTest** block from **sourceSets** as you won’t be writing any tests in this module.

Do a Gradle Sync.

Open **settings.gradle.kts** file in the project root folder. Confirm that **shared-action** is now part of **learn**:

```
include(":shared-action")
```

The Android Studio template for Kotlin Multiplatform Mobile only generates the Android and iOS targets, so you’ll need to manually add the desktop platform.

In the **shared-action** module, open the **build.gradle.kts** file. In the **kotlin** section, after the `listOf` iOS targets, add:

```
jvm("desktop")
```

Now, go to `sourceSets`, and at the bottom add the `desktopMain` variable:

```
getByName("desktopMain") {  
    dependencies { }  
}
```

Inside the `kotlinOptions` block, change the `jvmTarget` as follows:

```
jvmTarget = JavaVersion.VERSION_17.toString()
```

Scroll down to the end of the file and inside the `android` section, add:

```
compileOptions {  
    sourceCompatibility = JavaVersion.VERSION_17  
    targetCompatibility = JavaVersion.VERSION_17  
}
```

To guarantee that all the modules are using the same Java version – 17.

With this structure, when you compile the project, you're going to see the following warning:

```
Variable 'commonMain' is never used
```

This is because in the `sourceSets` section, `commonMain` is declaring a variable that won't be used. To remove this warning, replace the existing implementation with:

```
getByName("commonMain") {  
    dependencies {  
        //put your multiplatform dependencies here  
    }  
}
```

Synchronize the project and wait for this operation to finish.

You still need to add the `desktopMain` folders on the **shared-action** module. An easy solution to implement this is to right-click `src` and select **New ▶ Directory**. You'll see a new window with a couple of folder suggestions. Search for “desktop” and select **desktopMain/kotlin**.

Now you're just missing the package structure. You can easily create this directory by right-clicking **desktopMain/kotlin**. This time, select **New > Package**. In this new window, enter: **com.kodeco.learn.action**.

You can also remove the **Platform.*.kt** and **Greeting.kt** files that Android Studio generated in the **androidMain**, **commonMain** and **iosMain** folders.

That's it!

Your project structure will look like this:

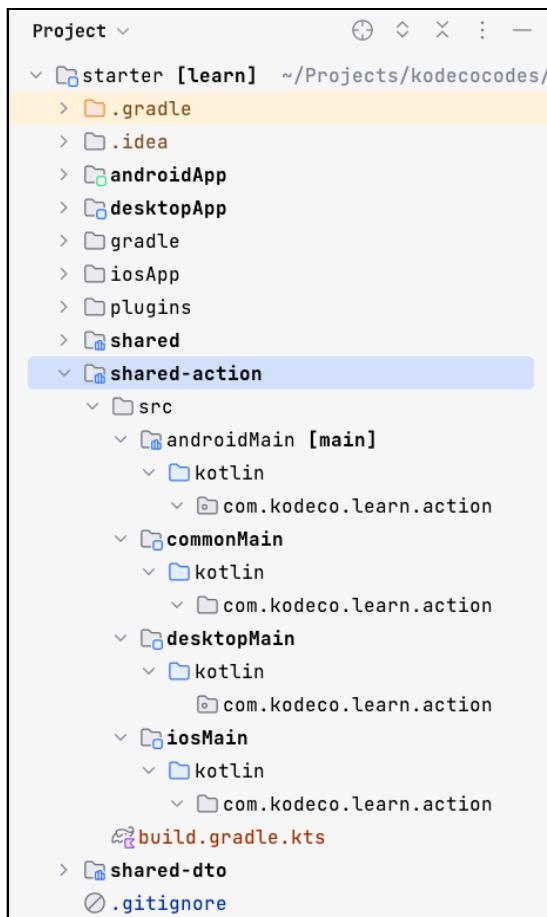


Fig. 14.1 – Project structure

Depending on the view type you have selected on the Android Studio **project** tab, you might have a different tree structure. To see the same one, select the **Project** option on top.

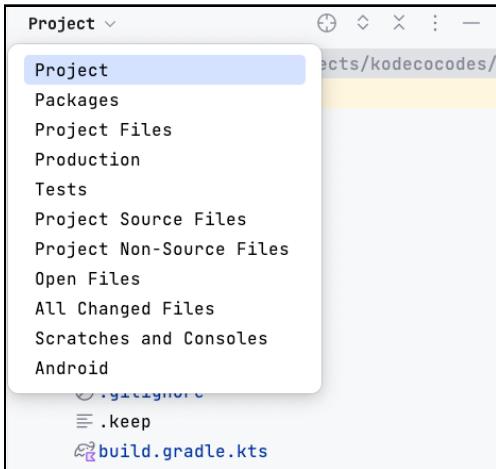


Fig. 14.2 – Android Studio project view

Configuring an Android Library to Publish

To publish the Android libraries, add the following code to the the `androidTarget` definition in the `kotlin` section of the `build.gradle.kts` file from **shared-action** to:

```
androidTarget {  
    publishLibraryVariants("release", "debug")  
}
```

If you don't define Android to publish its libraries, your project will use the one created for desktop by default. This is possible since JVM supports Android. However, this won't work because the platform-specific code is entirely different on both platforms.

Configuring a Multiplatform Swift Package

You have different possibilities to generate a library. Since Apple has its own package manager — Swift Package Manager — and many libraries are now available through it, you're going to use it in this chapter.

However, there's no official plugin to generate a Swift Package from a KMP project. So, you'll need to use the Multiplatform Swift Package (<https://github.com/ge-org/multiplatform-swiftpackage>) plugin. In the **starter** project, you've got a **plugins** folder that contains an updated version of this library. Open the **settings.gradle.kts** file to include it in the project as follow:

```
includeBuild("plugins/multiplatform-swiftpackage-m1_support")
```

Note: This customized version of the plugin supports the Apple M1 architecture and uses the KMP **XCFramework** functions internally to generate the Frameworks.

Synchronize the project. Now, open the **build.gradle.kts** from **shared-action**, and add in the **plugins** section:

```
id("com.chromaticnoise.multiplatform-swiftpackage-m1-support")
```

You can configure your Swift package by defining a couple of parameters. To accomplish this, add before the **kotlin** section:

```
//1
multiplatformSwiftPackage {
    //2
    xcframeworkName("SharedAction")
    //3
    swiftToolsVersion("5.3")
    //4
    targetPlatforms {
        iOS { v("13") }
    }
    //5
    outputDirectory(File(projectDir, "sharedaction"))
}
```

Here's what's happening:

1. This is the function that allows configuring the generated Swift package.
2. You define a specific name for the generated framework by setting the **xcframeworkName**. Otherwise, it will use the module's name as default.
3. As the name indicates, it corresponds to the Swift tools version required by the generated package.

4. `targetPlatforms` defines which platforms and OS versions the framework should support. In this case, it's going to work on all iOS devices and simulators that have version 13 or newer.
5. By default, `swiftpackage` is the output folder of the Swift package. You can define a different location and name through the `outputDirectory` parameter.

This plugin generates the Swift Package Manager Manifest and the XCFramework that you can use in `iosApp`.

For the Framework to be generated as “SharedAction” you need to, additionally, set the XCframework name, and its `baseName`. Go to the iOS target section and update:

```
val xcf = XCFramework("SharedAction")
listOf(
    iosX64(),
    iosArm64(),
    iosSimulatorArm64()
).forEach {
    it.binaries.framework {
        basePath = "SharedAction"
        xcf.add(this)
    }
}
```

Synchronize the project. Open the terminal, and in the project root folder, run:

```
./gradlew shared-action:createSwiftPackage
```

The above gradle command allows you to only generate a Swift package for the `shared-action` module. Since you explicitly mentioned `shared-action:createSwiftPackage`.

You should see:

```
BUILD SUCCESSFUL
```

Look at the `shared-action` folder. You'll see a new `sharedaction` directory that only contains one file: `Package.swift`. Since the project doesn't contain any code, no XCFrameworks were generated.

Create a `Action.common.kt` file inside `shared-action/commonMain`, and execute the `createSwiftPackage` command once again.

After its execution, return to the **sharedaction** folder. You now have the **XCFrameworks** for **arm64** and **arm64_x86_64-simulator**.

That's it! Your module is ready to generate Swift packages.

You can also make the same update on the **shared** module. Similar to what you've done on **shared-action**, open the **shared/build.gradle.kts** file and add the plugin:

```
id("com.chromaticnoise.multiplatform-swiftpackage-m1-support")
```

And then its configuration:

```
multiplatformSwiftPackage {  
    xcframeworkName("SharedKit")  
    swiftToolsVersion("5.3")  
    targetPlatforms {  
        iOS { v("13") }  
    }  
}
```

If you want to generate the Swift package for both shared modules, you can easily do it by omitting the library name as prefix:

```
./gradlew createSwiftPackage
```

Migrating the Code to Multiplatform

Now that you've got everything configured, it's time to move the code from the app's UI to Multiplatform. Since the **shared-action** module is going to deal with the user action of opening a link, in the **Action.common.kt** that you've created inside **commonMain**, add:

```
public expect object Action {  
    public fun openLink(url: String)  
}
```

This is the object the UI will call.

Alternatively, you can just define the **openLink** function without adding it to an object:

```
public expect fun openLink(url: String)
```

You can call this function directly from **androidApp** and **desktopApp**, since you reference it directly. However, from **iosApp**, you would need to access it via:

```
PlatformActionKt.openLink(url: "\${item.link}")
```

Since there's no class defined, the compiler creates one when generating the framework and uses the class name plus the extension of the file as its name.

Although you could define a different name, you're always going to have the **kt** prefix — which isn't the most sympathetic name, especially when you're trying to convince the iOS team to adopt a shared module written in Kotlin. :]

Android Studio prompts a suggestion to automatically generate the missing files. Ignore it for now. In some IDE versions, this feature is not working as expected and ends up creating the files in the wrong directories. To avoid any unexpected errors, you're going to add all of these files manually.

Create a **Action.android.kt** file for **androidMain**, **Action.ios.kt** for **iosMain**, and **Action.desktop.kt** for **desktopMain**. They should all be in the same directory for each of the platforms: **com.kodeco.learn.action**.

Define an empty **actual** function in all the newly created files:

```
public actual object Action {  
    public actual fun openLink(url: String) {}  
}
```

Implement the **openLink** functions for the different targets:

- **androidApp**: Open the **MainActivity.kt** file and scroll until you see an **openEntry** function. Copy its content and paste it on the **openLink** function you created in **shared-action/androidMain**:

```
public actual fun openLink(url: String) {  
    val intent = Intent(Intent.ACTION_VIEW)  
    intent.data = Uri.parse(url)  
    startActivity(intent)  
}
```

Since this code block exists outside the scope of an **Activity**, you need its **Context** to call **startActivity**. To overcome this, you're going to declare an **activityContext** variable outside the **Action** object:

```
public lateinit var activityContext: Context
```

Now, update the `startActivity` call to:

```
activityContext.startActivity(intent)
```

You've got access to the Android SDK, so for the above function you'll need to import:

```
import android.content.Context
import android.content.Intent
import android.net.Uri
```

- **desktopApp:** Go to the `Main.kt` file and search for `openEntry`. Copy its content into the `openLink` function from **shared-action/desktopMain**.

```
public actual fun openLink(url: String) {
    try {
        val desktop = Desktop.getDesktop()
        desktop.browse(URI.create(url))
    } catch(e: Exception) {
        Logger.e(TAG, "Unable to open url. Reason: ${e.stackTrace}")
    }
}
```

The `Logger` class belongs to the **shared** module you're not using on **share-action**. To solve this, you can do one of the following:

- Replace the call to use `println` instead.
- Add the **shared** library as a dependency. That's excessive, though, since you'll end up increasing the app size with unnecessary features.
- Create a logger library and add it to **shared-action**.

For now, you're going to follow the first approach. However, the third one is tempting, so don't forget to do the first challenge of this chapter, and afterward come back to this step and replace the `println` function with `Logger`. :]

Replace the current `Logger` call with:

```
println("Unable to open url. Reason: ${e.stackTrace}")
```

And import:

```
import java.awt.Desktop
import java.net.URI
```

- **iosApp:** Open the **LatestView.swift** file and search for `openURL`. You'll find two results: the first one declares the variable from `Environment`, and the second one its invocation.

Moving this logic to **shared-action/iosMain** is more difficult than the previous ones because you'll need to convert this code to Kotlin and find the corresponding iOS functions.

It's important to remember that although you're writing Swift code, KMP uses Objective-C signatures. This is why you use the `NSLog` on the **PlatformLogger.kt** file from **shared/iosMain**.

Occasionally, it can be difficult to find the module of a specific function. The native libraries follow the same structure as the ones from the iOS SDK, so the first step is to go to the official documentation website (<https://developer.apple.com/documentation/uikit/uiapplication/1648685-openurl?language=objc>) and change it to Objective-C:

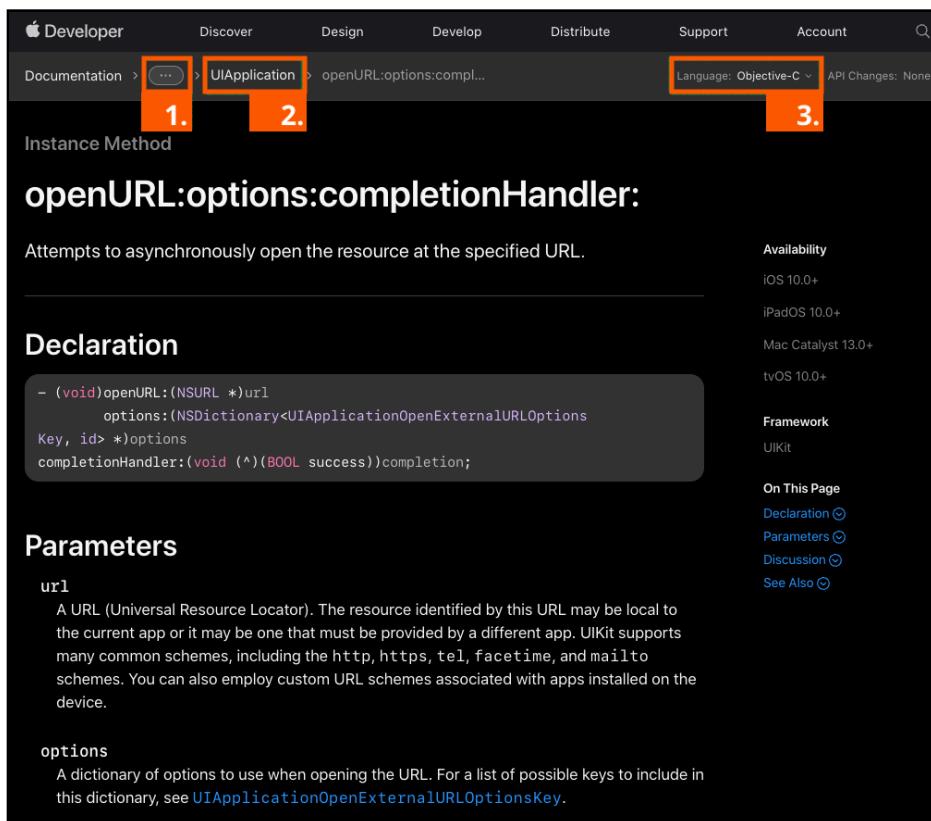


Fig. 14.3 – Apple documentation for `OpenURL` function

In this image, you can find the `openURL` documentation:

1. These dots show you the path where `openURL` is located. You can see that `UIApplication` belongs to `UIKit` if you click on them. So, to access this function from `iosMain`, you'll need to import:

```
import platformUIKit.UIApplication
```

2. Open the `UIApplication` page. According to the documentation, the `UIApplication` is a singleton that you can access via `sharedApplication`. Therefore, to access `openURL`, you need to call:

```
UIApplication.sharedApplication.openURL(url)
```

3. This drop-down allows you to switch between Swift and Objective-C.

Note: You can see all the classes that exist inside `platform` if you go to JetBrains' kotlin-native repository (<https://github.com/JetBrains/kotlin-native/tree/archive/platformLibs/src/platform/ios>).

Now that you learned how to implement `openURL`, return to `Action.ios.kt` from `shared-action/iosMain` and update the existing `openLink` function with:

```
public actual fun openLink(url: String) {
    val application = UIApplication.sharedApplication
    val nsurl = NSURL(string = url)
    if (!application.canOpenURL(nsurl)) {
        println("Unable to open url: $url")
        return
    }
    application.openURL(nsurl)
}
```

When prompted, import the following libraries:

```
import platform.Foundation.NSURL
import platformUIKit.UIApplication
```

Note: There's currently an issue on the Android Studio for Mac M1 where the `platform` package seems to not be resolved. In other words, you might see the `platform` import along with the `UIApplication` and `NSURL` calls in red. If this is the case, don't worry — you can compile the project without any problems.

That's it! To compile the project and generate the JVM, Android libraries and XCFramework, run:

```
./gradlew assemble
```

And to generate the Swift Package (shared and shared-action), run:

```
./gradlew createSwiftPackage
```

Now you've got two libraries you can publish!

Adding a New Library to the Project

Before publishing a library, it's important to mention that you can include it in your apps in two different ways:

Add as a New Dependency

At the same level as **shared**, you include it on Android and desktop **build.gradle.kts** files and add it to the iOS app project.

On the **androidApp** and then on **desktopApp build.gradle.kts** files, in the dependencies section after the **shared** implementation, add:

```
implementation(project(":shared-action"))
```

Synchronize the project.

For iOS, you need to first open the project with **Xcode**, and follow these steps:

1. Open the **Project** file and click the **General** tab on top.
2. Scroll down to **Frameworks, Libraries, and Embedded Content** and click the plus sign below the **SDWebImageSwiftUI**.
3. A new window will open asking you to choose the framework. Click **Add Other...** then **Add Files...**
4. Navigate to **./shared-action/sharedaction/**, select the **SharedAction.xcframework** and click **Open**.

Currently, **shared** is being added through the **embedAndSignAppleFrameworkForXcode** command from **Compile Kotlin** run script defined on **Build Phases**.

Optionally, you can remove it and add it manually, following the same process as the one described above. In case you follow this process, your Xcode will have both frameworks added to the project:

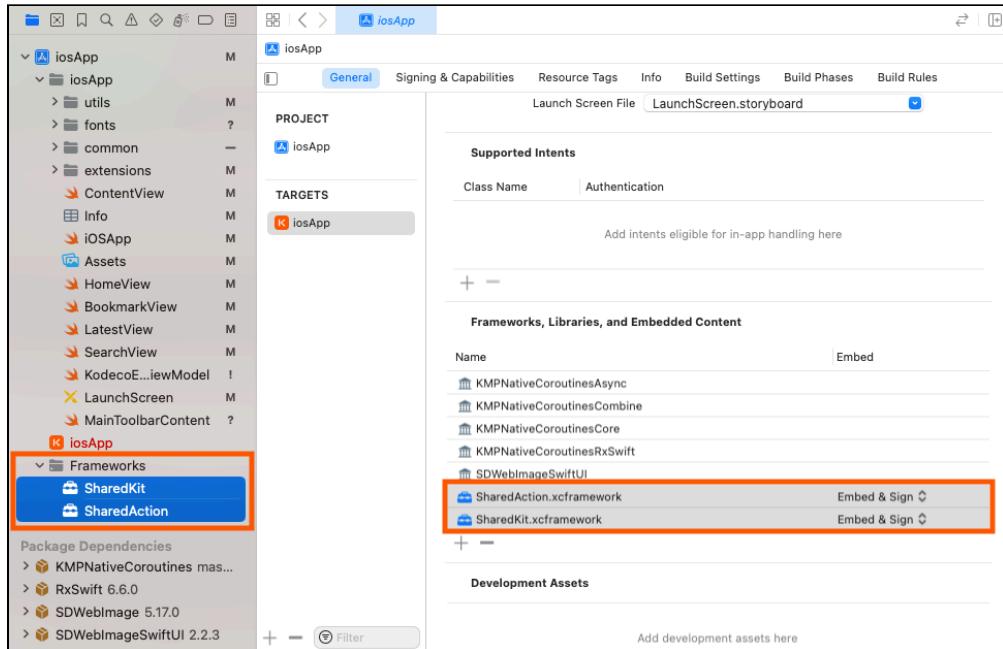


Fig. 14.4 – XCode project view

Include Inside the Shared Module

The existing **shared** module can import the library and makes its features available to all apps that use it.

This step is simpler to do — in this case, you just need to add the **shared-action** as an implementation to the **shared** module **build.gradle.kts** file in the **commonMain** dependencies section:

```
implementation(project(":shared-action"))
```

To have a more strict separation of concerns, you're going to follow the first option for **learn**.

Updating Your Apps to Use Your New Library

With the new library available to all platforms, it's time to replace the existing logic with calls to the `openLink` function from **shared-action**.

On `androidApp`, open the `MainActivity.kt` file and update the `openEntry` function to:

```
private fun openEntry(url: String) {
    activityContext = this
    openLink(url)
}
```

The `activityContext` that you're setting here will be used to open a new activity from **shared-action**.

Additionally, don't forget to add the requested imports:

```
import com.kodeco.learn.action.Action.openLink
import com.kodeco.learn.action.activityContext
```

And remove the redundant one:

```
import android.net.Uri
```

The next update that you need to do is on the `Main.kt` file on the `desktopApp` project. When invoking the `MainScreen` Composable, update the `onOpenEntry` call to:

```
onOpenEntry = { openLink(it) },
```

Don't forget to add the required import.

With this, you're going to use the function from **shared-action** to open an article on your default browser. You can now remove `openEntry` at the end of this file.

To update the iOS app, switch to Xcode and make the same update on the following files:

- `./common/KodecoEntryRow.swift`
- `LatestView.swift`

Remove the `openURL` declaration:

```
@Environment(\.openURL)  
var openURL
```

Replace the `Button` action when iterating over the `items`, from `openURL` to:

```
Action().openLink(url: "\$(item.link)")
```

Don't forget to add the shared-action framework import:

```
import SharedAction
```

And remove the `url` variable which is no longer necessary.

Now that you've updated all three platforms, compile and run the apps, browse through the articles list and select one to read.

Depending on your default browser, you'll see screens similar to these:



Fig. 14.5 – Android app: Open an article

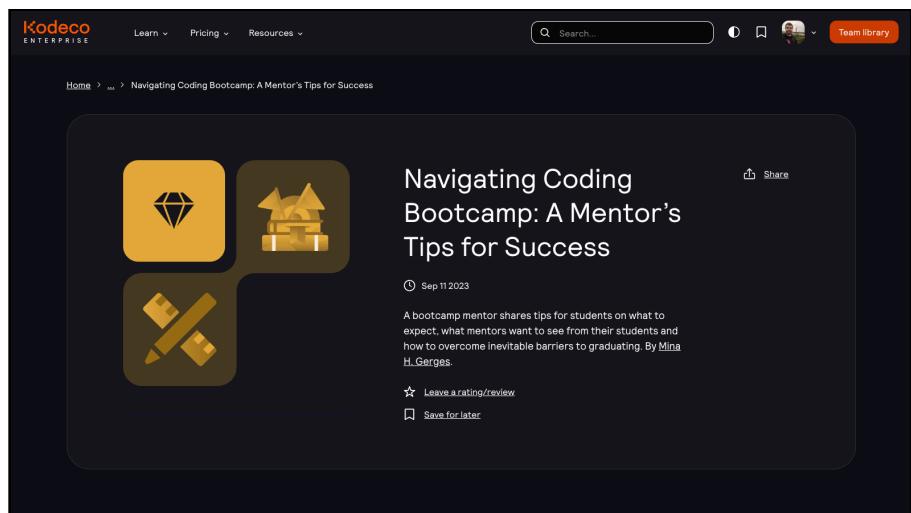


Fig. 14.6 – Desktop app: Open an article



Fig. 14.7 – iOS app: Open an article

Publishing Your KMP Library

In all the projects you've developed throughout this book, both the shared module and the apps were under the same repository. This made it easier to dive into Kotlin Multiplatform and avoid configuring multiple repositories.

With this, you can easily import any of them by just including it on the **settings.gradle.kts** file located in the project root directory:

```
include(":androidApp")
include(":desktopApp")

include(":shared")
include(":shared-action")
```

Each one of these `includes` represents a project that could be on a different repository. If you had them as a separate repository, you'd need to also set the project path:

```
include(":your-library")
project(":your-library").projectDir = file("../path/to/your-
library")
```

Both of these scenarios present a couple of disadvantages:

- The configuration is laborious. You need to add the projects both on **settings.gradle.kts** and on the **build.gradle.kts** of the project that will use them.
- Higher build time — particularly the first time the project builds. This happens because there's no library compiled at that moment.
- There's no versioning on these projects. If you want to use an older revision of the project, you need to manually checkout.

Alternatively to both scenarios, instead of including these modules, you can import its libraries either from a local maven repository or from a remote server.

In this section, you'll publish the **shared-action** library that you created before.

Configuring a Library

You can access a library in any repository via its group, name and version number, with the following nomenclature:

- **group:name:version**

The project name is the folder name – in this case, **shared-action**. You can define the group and version on the **build.gradle.kts** file from **shared-action**.

To set the version, add the following parameter to **shared-action build.gradle.kts** :

```
version = "1.0"
```

The group, if not defined, uses the parent name. In this case, it would be **learn**. This can be a bit misleading since there's no information about the author. To overcome this, add the following above version:

```
group = "com.kodeco.shared"
```

How to Publish a Library Locally

At the end of the plugin section, add:

```
id("maven-publish")
```

And now to publish it locally, run on the terminal:

```
./gradlew shared-action:publishToMavenLocal
```

When the operation ends, you can read **BUILD SUCCESSFUL** in the console logs.

If you want to publish all your libraries locally, you should run instead:

```
./gradlew publishToMavenLocal
```

The default location for your local maven repository is on:

```
~/.m2/repository
```

Navigate to this folder, and you can see a **com/kodeco/shared** directory with the **shared-action** library for the different platforms inside.

Return to Android Studio. Before continuing, remove the `include` of the **shared-action** module from **settings.gradle.kts**:

```
include(":shared-action")
```

On the root of the **learn** project, open the **build.gradle.kts**. In the **allprojects** section, under **repositories** after **google()**, add:

```
mavenLocal()
```

The next time Gradle synchronizes, it will also look for the project dependencies in your **.m2/repositories** directory.

You're just missing an update to the app's dependencies. Open the **build.gradle.kts** files from **androidApp** and **desktopApp** and replace the entry:

```
implementation(project(":shared-action"))
```

that looks for the project, with:

```
implementation("com.kodeco.shared:shared-action:1.0")
```

That uses the library instead.

Compile both Android and desktop apps and open an article from the list.

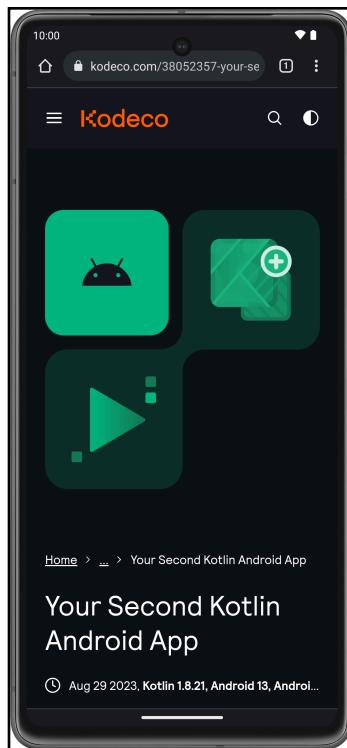


Fig. 14.8 – Android app: Open an article

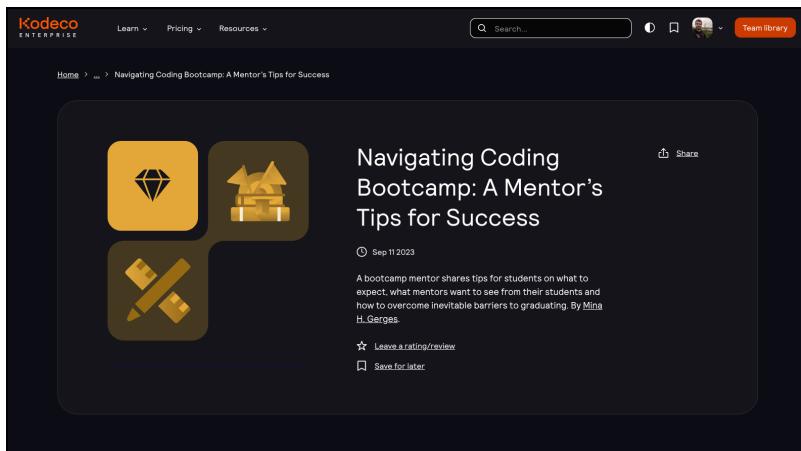


Fig. 14.9 – Desktop app: Open an article

How to Publish a Library to the GitHub Packages Repository

There are a set of repositories that you can use to publish your libraries: JitPack (<https://jitpack.io/>), Maven Central (<https://search.maven.org/>) and GitHub Packages (<https://github.com/features/packages>). These are the most common. Or, you can always set up your own package repository.

Depending on the repository that you select, the configuration process should be similar to the one presented in this section. Typically, the differences are the URL that you use to connect to and the authentication required.

Here, you're going to use GitHub Packages — mainly because it's simple to configure and has a free tier that you can use. You just need to create an account.

Before you can publish a library, you need to first create the access token that Gradle will use to authenticate your account.

Create Your Access Token

Log in to GitHub and go to your account **Settings**. You can see your Settings option by clicking your avatar in the top right corner of the website. Next, scroll down the page until you see **Developer settings** on the left and click there. You'll be redirected to a new screen. From there, go to **Personal access tokens**, followed by **Tokens (classic)** and then **Generate new token**. From the drop-down that appears select **Generate new token (classic)**.

Or, you can go directly to this link (<https://github.com/settings/tokens/new>).

In this screen, you can configure a name for your token, how long it will be valid and which permissions it should have. For the name, add: **Publish Maven Repository** and check the **write:packages** and **read:packages** checkboxes.

It will automatically select the **repo** attribute. Your screen will be similar to this one:

The screenshot shows the GitHub 'New personal access token' configuration interface. On the left, there's a sidebar with 'Settings / Developer settings' and three tabs: 'GitHub Apps', 'OAuth Apps', and 'Personal access tokens', with 'Personal access tokens' being the active tab. The main area has a title 'New personal access token'. Below it, a note states: 'Personal access tokens function like ordinary OAuth access tokens. They can be used instead of a password for Git over HTTPS, or can be used to authenticate to the API over Basic Authentication.' A 'Note' section contains the placeholder 'Publish Maven Repository'. A 'What's this token for?' section is present. An 'Expiration *' section shows '30 days' selected, with a note that the token will expire on 'Wed, Feb 9 2022'. A 'Select scopes' section follows, with a note: 'Scopes define the access for personal tokens. [Read more about OAuth scopes](#)'. A table lists various scopes with checkboxes:

Scope	Description
<input checked="" type="checkbox"/> repo	Full control of private repositories
<input checked="" type="checkbox"/> repo:status	Access commit status
<input checked="" type="checkbox"/> repo_deployment	Access deployment status
<input checked="" type="checkbox"/> public_repo	Access public repositories
<input checked="" type="checkbox"/> repo:invite	Access repository invitations
<input checked="" type="checkbox"/> security_events	Read and write security events
<input type="checkbox"/> workflow	Update GitHub Action workflows
<input checked="" type="checkbox"/> write:packages	Upload packages to GitHub Package Registry
<input checked="" type="checkbox"/> read:packages	Download packages from GitHub Package Registry

Fig. 14.10 – GitHub token configuration

Note: It's important to choose a name that you can easily remember later on. It helps when you receive an email from GitHub saying that your token is about to expire and you need to decide whether you want to renew it or not.

Click **Generate token**. Copy the authentication token.

Create a New Repository

To publish your libraries, you need a repository to push them. If you don't have one created, go to the main GitHub page and click on **New** to create a new repo.

Alternatively, you can go directly to this link (<https://github.com/new>).

Write a **repository name** — for instance, **shared-action** — decide if you want to make it public or private, and select the **Add a README file** checkbox, so there's already a branch created for you to use.

Publish Your Library

With the GitHub Package repository ready, return to Android Studio and open the **gradle.properties** file located in the root directory. Here, add your account username and the token that you copied earlier:

```
#Repository Credentials  
mavenUsername=YOUR_USERNAME  
mavenPassword=YOUR_TOKEN
```

These are the credentials that Gradle is going to use to authenticate.

Open the **build.gradle.kts** file from the **shared-action** module and scroll to the bottom.

After the `multiplatformSwiftPackage`, add:

```
publishing {  
    repositories {  
        maven {  
            //1  
            url = uri("https://maven.pkg.github.com/YOUR_USERNAME/  
YOUR_REPOSITORY")  
            //2  
            credentials(PasswordCredentials::class)  
            authentication {  
                create<BasicAuthentication>("basic")  
            }  
        }  
    }  
}
```

This Gradle task is responsible for publishing your libraries into the URL you defined. It's using **BasicAuthentication** as the authentication mechanism:

1. In this URL, you need to define:
2. **YOUR_USERNAME**: As the name implies, it's the username of your GitHub account.
3. **YOUR_REPOSITORY**: The repository name that you chose before. If you followed the same naming convention, it should be **shared-action**.
4. Gradle supports different types of authentication. You can find all of these methods on their documentation website (https://docs.gradle.org/current/userguide/declaring_repositories.html). The `PasswordCredentials::class` looks at the `mavenUsername` and `mavenPassword` to authenticate the request.

Alternatively, you could define these variables directly here by replacing `credentials` with:

```
credentials {  
    name = YOUR_USERNAME  
    password = YOUR_TOKEN  
}
```

It's a good practice to have these in a separate file that should be added to the **.gitignore** file to avoid unconsciously pushing the credentials into the repository.

Before publishing your library, you need to add shared-action module once again to the **settings.gradle.kts** file. Open it and after **shared** add:

```
include(":shared-action")
```

Now that everything is ready, go to the terminal and enter:

```
./gradlew shared-action:publish
```

When this operation ends, you'll see a **BUILD SUCCESSFUL** message in the console. Open your repository GitHub page and on the right side, you'll see a section named **Packages** that should have a list of the libraries that you just uploaded.

Go to the Packages section, and you'll see a screen similar to this one:

The screenshot shows a list of 7 published libraries on GitHub Packages. Each entry includes the library name, a brief description, the date it was published, the author, and a download count of 0.

Library	Description	Published	Author	Downloads
com.kodeco.shared.shared-action-android-debug	Published 4 minutes ago by Carlos Mota in com.kodeco.shared.shared-action-android-debug	4 minutes ago	Carlos Mota	0
com.kodeco.shared.shared-action-android	Published 3 minutes ago by Carlos Mota in com.kodeco.shared.shared-action-android	3 minutes ago	Carlos Mota	0
com.kodeco.shared.shared-action-iosarm64	Published 3 minutes ago by Carlos Mota in com.kodeco.shared.shared-action-iosarm64	3 minutes ago	Carlos Mota	0
com.kodeco.shared.shared-action-desktop	Published 3 minutes ago by Carlos Mota in com.kodeco.shared.shared-action-desktop	3 minutes ago	Carlos Mota	0
com.kodeco.shared.shared-action-iosx64	Published 2 minutes ago by Carlos Mota in com.kodeco.shared.shared-action-iosx64	2 minutes ago	Carlos Mota	0
com.kodeco.shared.shared-action-iossimulatorarm64	Published 3 minutes ago by Carlos Mota in com.kodeco.shared.shared-action-iossimulatorarm64	3 minutes ago	Carlos Mota	0
com.kodeco.shared.shared-action	Published 2 minutes ago by Carlos Mota in com.kodeco.shared.shared-action	2 minutes ago	Carlos Mota	0

Fig. 14.11 — GitHub published libraries

Now that you've confirmed that your libraries were successfully uploaded, return to Android Studio and in **build.gradle.kts** that's located in the root directory, add the following code after `mavenCentral`, which is inside the `allProject/repositories` section:

```
maven {  
    url = uri("https://maven.pkg.github.com/YOUR_USERNAME/  
YOUR_REPOSITORY")  
    credentials(PasswordCredentials::class)  
    authentication {  
        create<BasicAuthentication>("basic")  
    }  
}
```

Previously, you defined the URL for publishing your libraries. Now, you're adding to the list of repositories that Gradle should look into when downloading the project dependencies.

Note: To use the credentials that you defined on `gradle.properties`, you need to have this maven repository declared above the one from JetBrains. Otherwise, you'll get an error related to missing credentials. This is due to the multiple declarations of maven repositories. Gradle automatically matches the maven repositories added with the credentials on `gradle.properties`, so the first one corresponds to `mavenUsername/mavenPassword`, the second one to `maven2Username/maven2Password` and so on.

And that's it! The project is ready. Compile and run both apps: Android and desktop.

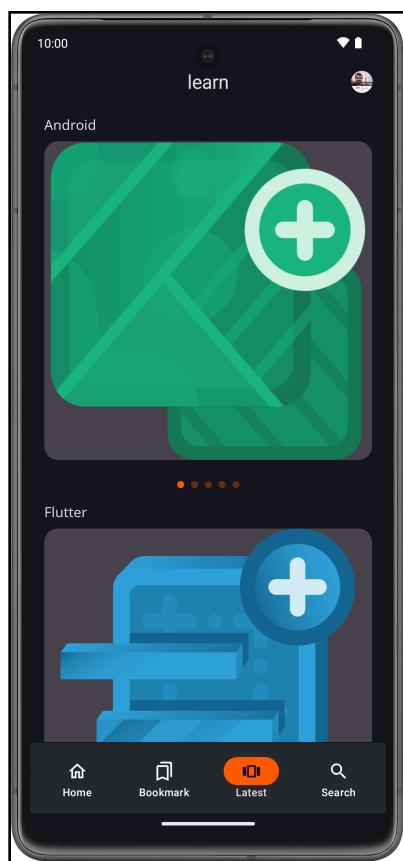


Fig. 14.12 – Android app: Browse through the latest articles

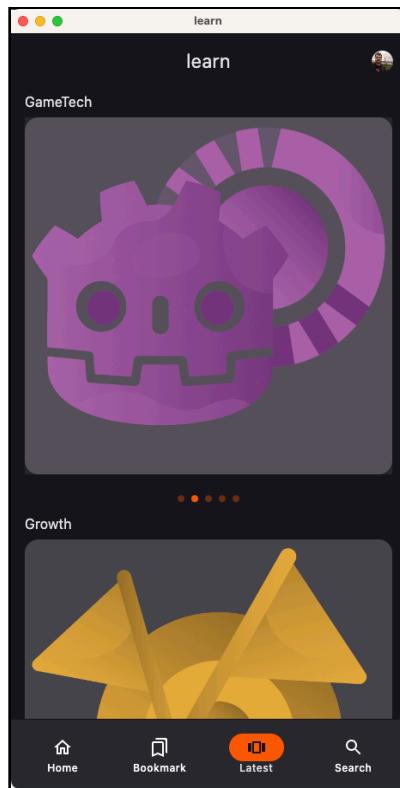


Fig. 14.13 – Desktop app: Browse through the latest articles

How to Publish Your Swift Package

With your GitHub repository already configured from the section above, you can use it to publish your Swift package.

Open the terminal and run:

```
./gradlew shared-action:createSwiftPackage
```

When the build ends, you'll see a **sharedaction** directory inside the **shared-action** module that contains your frameworks. Copy the contents of this folder to your GitHub repository and push these files.

In the root of your repository, you should have the following files:

- **SharedAction.xcframework.**
- **Package.swift.**
- **README.md.**
- **SharedAction-1.0.zip**

Note: You can't have them inside a folder. Otherwise, you won't be able to add them easily to your project.

Open Xcode and go to **Project** and select the **General** tab. Scroll down to **Frameworks, Libraries, and Embedded Content**, and in case you're still using the local **SharedAction** framework, remove it.

Next, click **+**, then **Add Package Dependency** on the bottom drop-down. A new window opens, and you can enter your repository URL in the top right corner. Depending on its visibility, Xcode might ask you for your GitHub credentials.

You'll see a similar screen to this one:

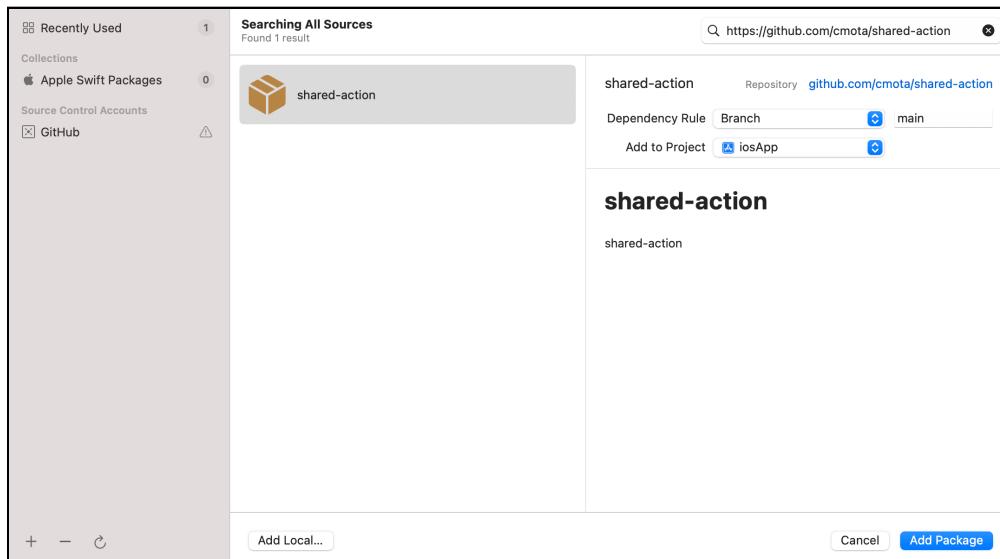


Fig. 14.14 – XCode add a Swift package from a custom URL

On the **Add to Project** drop-down, select **iosApp** and then **Add Package**. Xcode will download your library.

If everything works as expected, you'll see a second prompt asking you to confirm whether to add the **SharedAction** package or not:

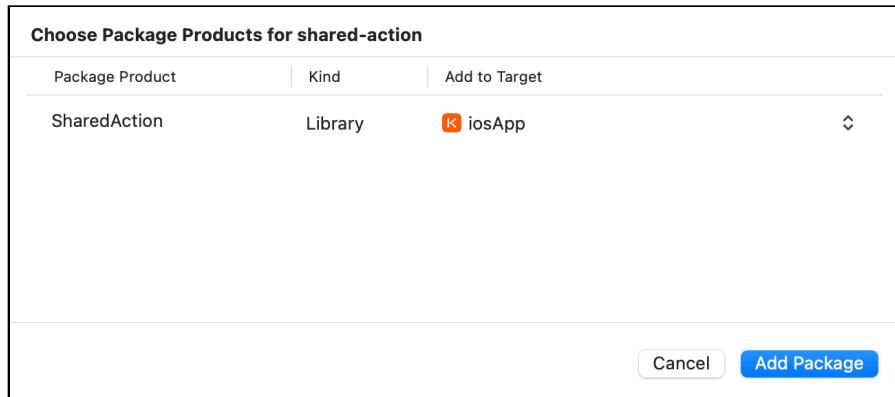


Fig. 14.15 – XCode add a Swift package: Confirm

Click **AddPackage**. When this operation ends, you can see the **SharedAction** framework added to the project.

Compile and run the app. There are new articles ready for you to read!



Fig. 14.16 – iOS app: Browse through the latest articles

Handling Multiple Frameworks

Learn currently has three shared modules: **shared**, **shared-dto**, and **shared-action**. Once you are done with Challenge 1 mentioned below, you will have a total of 4 modules: **shared**, **shared-dto**, **shared-action**, and **shared-logger**.

Now, if the first three modules use **shared-logger** as a dependency and you compile it to both Android and desktop you're going to have only one instance of your logger, while when you generate the frameworks for iOS, each one of them will have its own copy.

This is currently a limitation on iOS. If you have multiple libraries that need to communicate through a single instance, you'll need to create an umbrella framework and export all of them together.

You'll need to create a new shared module. To revisit these steps, see the “Adding a New Module” section in this chapter. Afterward, when defining the list of libraries to be exported, just add the ones that you need to be under this umbrella framework. For instance, if it's **shared** and **shared-logger** it would be:

```
val xcf = XCFramework("SharedAll")
listOf(
    iosX64(),
    iosArm64(),
    iosSimulatorArm64()
).forEach {
    it.binaries.framework {
        basePath = "SharedAll"
        xcf.add(this)

        export(project(":shared"))
        export(project(":shared-logger"))
    }
}
```

Now, instead of importing **SharedKit** and **SharedLogger** on Xcode , you have to import the newly created **SharedAll** framework.

Challenges

Here are some challenges for you to practice what you've learned in this chapter. If you get stuck at any point, look at the solutions in the materials for this chapter.

Challenge 1: Create a Logger

All the apps and the **shared** module use the **Logger** class defined on **shared/PlatformLogger.kt**. It's a simple logger that calls on:

- Android, the **android.util.Log**.
- Desktop, the **println**.
- iOS, the **platform.Foundation.NSLog**.

In this first challenge, create and publish a new library — **shared-logger** — that should contain the **PlatformLogger.kt** implementation for all three platforms: Android, desktop and iOS.

Challenge 2: Integrate the Logger Library

Throughout this book, you created three different apps:

- **Find time**, a time zone helper, in Section 1.
- **Organize**, a multiplatform TODO app, in Section 2.
- **learn**, an RSS feed reader for kodeco.com articles, in Section 3.

They each had a customized version of a logger. The second challenge is to use the library that you created from the first challenge, on all three apps. Don't forget to make the changes both at the business logic and UI levels.

Challenge 3: Use the Logger Library in the Shared-Action Module

At the beginning of this chapter, you successfully migrated the open links functions to Kotlin Multiplatform and created the **shared-action** module.

At the time, there was no logger class, so you used the **println** function as the module logger. With the recently created **shared-logger**, it's now time to update **shared-action** and use your new library.

Key Points

- If the features you want to migrate to KMP have any platform-specific code, you need to write this specific logic for all the platforms your library will target.
- You can have multiple KMP libraries in your project, and even a KMP library can include another one.
- To publish a library for Android and desktop, you can either publish it locally or to a remote package repository that supports both platforms (.jar and .aar). In this book, you've seen how to use GitHub Packages.
- For iOS, you're creating a Swift package to share your library. Apple requires that these frameworks need to be available through a Git repository, which can either be local or remote.

Where to Go From Here?

Congratulations! You've finished the last chapter of the book. Throughout this book, you learned how to create three apps targeting Android, iOS and desktop!

You started this journey by getting familiar with Jetpack Compose and Swift UI for UI development and moved toward sharing your app's business logic across these three platforms with Kotlin Multiplatform. You can now create an app from scratch and apply all of these new concepts, or migrate one that you've already written to KMP.

Now that you're a Kotlin Multiplatform master, you might be wondering what to read next. Perhaps you want to dive deeper into Jetpack Compose (<https://www.kodeco.com/books/jetpack-compose-by-tutorials>) and SwiftUI (<https://www.kodeco.com/books/swiftui-by-tutorials>)?

Or, do you prefer to sit back and watch a video course instead? You can see the Jetpack Compose (<https://www.kodeco.com/21959310-jetpack-compose>) and Your Second iOS & SwiftUI app (<https://www.kodeco.com/25836622-your-second-ios-swiftui-app>) that teach you the same concepts as the books.

Additionally, since you're already familiar with Ktor, why not try it on another platform? One that doesn't require you to design a UI: Server-Side Kotlin with Ktor (<https://www.kodeco.com/2885892-server-side-kotlin-with-ktor>). There are a lot more materials available for you to use as you learn — find them at [kodeco.com](https://www.kodeco.com).

Looking forward to seeing what you're going to build next. :]

Stay safe, stay curious.

Conclusion

Congratulations! After a long journey, you've learned many important things about Kotlin Multiplatform and how you can leverage it to share code across native apps. You also learned how to develop UI on iOS, Android and desktop using the latest UI toolkits. Now you can apply what you learned in your next app or start migrating features in your current app, thus saving development time.

Remember, if you want to further your understanding of Kotlin and Android app development after working through *Kotlin Multiplatform by Tutorials*, we suggest you read Jetpack Compose by Tutorials and SwiftUI Apprentice. Both are available in our online store:

- <https://www.kodeco.com/books/jetpack-compose-by-tutorials>
- <https://www.kodeco.com/books/swiftui-apprentice>

If you have any questions or comments as you work through this book, please stop by our forums at <https://forums.kodeco.com> and look for the particular forum category for this book.

Thank you again for purchasing this book. Your continued support is what makes the books, tutorials, videos and other things we do at Kodeco possible. We truly appreciate it!

– The *Kotlin Multiplatform by Tutorials* team

Section IV: Appendices

Appendix A: Kotlin: A Primer for Swift Developers

By Carlos Mota

Kotlin is a language developed by JetBrains that gained popularity and wide adoption when Google announced that from that point on, all of their Android libraries would no longer be written in Java. It gained wide popularity, and at the time of writing, it's estimated that it's used by more than 60% of the Android developers worldwide (<https://developer.android.com/kotlin>).

If you open its official website (<https://kotlinlang.org/>), you'll immediately read *modern, concise, safe, powerful, interoperable* (with Java for Android development) and *structured concurrency*. All of these keywords are functionalities that developers look for in any programming language, and Kotlin has all of them.

Even more importantly, Kotlin is not only for Android. It also supports Web front-end, server-side and – the focus of your work throughout this book – Multiplatform.

Kotlin and Swift: Comparing Both Languages

The syntax between both languages is quite similar. If you're a Swift developer, you can easily program in Kotlin. This appendix shows you how to start.

The examples shown in this appendix are code snippets from **learn**. A final version of the project is available in the **materials** repository.

Basics

In this section, you'll learn the Kotlin basics — or as Swift developers are familiar with, its foundations. :]

One good thing about Android Studio is that in most cases, if you're missing an import or using a wrong type for a variable, it will automatically warn you and suggest a fix.

Note: Every time Android Studio underlines your code or shows a tooltip box, you can automatically accept its suggestion by pressing **Alt-Enter**.

Package Declaration

The extension of a Kotlin file is **.kt**. The tree hierarchy of a Multiplatform project typically follows the Android naming convention for package names — you've got three folder levels. In **learn**, it's **com/kodeco/learn**, and they usually correspond to:

- **com**, domain
- **kodeco**, company name
- **learn**, app name

Since the package name is unique — you can't have two apps on the Google Play Store with the same one — this convention guarantees there is no conflict between apps from different companies.

Every time you declare a new class or object, you must define the package declaration. This should be the first instruction of a new file. Or, if you have a copyright header, right after it.

If you open the **FeedPresenter.kt** inside the **presentation** folder from the **shared** module, you can see that the import is:

```
package com.kodeco.learn.presentation
```

In this case, `presentation` is the subfolder where this class is. The package definition should correspond to the same tree hierarchy — otherwise, you might end up importing the wrong files.

There's no package declaration to add in Swift.

Imports

Typically, when an import is missing, Android Studio shows you a prompt with one or more suggestions, so you shouldn't have any issues. In any case, if you want to add one manually, you need to add it after the package declaration:

```
import com.kodeco.learn.data.model.GravatarEntry
```

This is different from Swift. There's no need to add classes — you just need to import the framework you're going to use.

Comments

Similar to Swift, you can add three types of comments:

- **Line**, where you just need to add `//` before the code or text that you want to comment. In this example, `Logger` won't be executed:

```
public fun fetchMyGravatar(cb: FeedData) {
    //Logger.d(TAG, "fetchMyGravatar"

    //Update the current listener with the new one.
    listener = cb
    fetchMyGravatar()
}
```

- **Block**, where you need to surround your code or text with `/* */`. This is also used for adding the copyright section at the beginning of a file:

```
/*
 * Copyright (c) 2021 Razeware LLC
 *
 */
```

- KDoc (<https://kotlinlang.org/docs/kotlin-doc.html>), which corresponds to the documentation that's going to be generated for your project. You can use tags like @property, @param, @return, @constructor, etc. to provide additional information about a function:

```
/**  
 * This method fetches your Gravatar profile.  
 *  
 * @property cb, the callback used to notify the UI that the  
 * profile was successfully fetched or not.  
 */  
public fun fetchMyGravatar(cb: FeedData) {  
    //Your code goes here  
}
```

Note: The equivalent version of Kdoc for Swift is Jazzy (<https://github.com/Realm/jazzy>).

Variables

Similar to Swift, in Kotlin you also have two types of variables:

- **val**, which corresponds to Swift's **let**. It's a read-only (immutable) variable that can only be set once – in its declaration. In this case, `scope` is initialized with a specific value when declared. This value can never change throughout the app execution:

```
private val scope = PresenterCoroutineScope(defaultDispatcher)
```

- **var** is the same keyword as in Swift. It's a mutable variable, so you can set it as many times as you need. In this example, the initial value of `listener` is `null`. When the UI makes a new request for data, it's going to be updated with a new callback reference:

```
private var listener: FeedData? = null
```

In Swift, the above declaration is:

```
private var listener: FeedData? = nil
```

You can have **optional** values in both languages. The only difference is Kotlin uses `null`, whereas Swift uses `nil` to represent the absence of a value.

Lazy Initialization

Kotlin supports lazy initialization through the use of the `lazy` keyword. This variable needs to be immutable — in other words, you need to declare it as `val`.

The value of this variable will only be calculated when it's first accessed. You should only define a variable as lazy if you don't need to access it right away and the variable does some heavy work.

Open the **FeedPresenter.kt** file inside **shared/presentation** and search for content declaration:

```
val content: List<KodecoContent> by lazy {  
    json.decodeFromString(KODECO_CONTENT)  
}
```

As you can see, it's defined as `lazy`. Do this to avoid decoding `KODECO_CONTENT` immediately when the app starts. It's one less thing to process.

If your app has a heavy startup, following this approach will give you a faster and smoother initialization of the app. The value will only be set when there's a call to `content`.

Late Initialization

You can delay the initialization of a variable until your app needs it. For that, you need to set it as `lateinit`, and it can't be set as immutable or null.

Change the listener on **FeedPresenter.kt** to:

```
private lateinit var listener: FeedData
```

Removing the `?` and `null` defines this object as non-null. You'll immediately see a couple of warnings through this file:

```
onSuccess = { listener?.onNewDataAvailable(it, platform,  
    null) },  
onFailure = { listener?.onNewDataAvailable(emptyList(),  
    platform, it) }
```

In particular, you'll see warnings, on the `?` in the above lambda expressions. Since, `listener` is not null, you can call the callback directly now that the value won't be null. This shouldn't be a problem since all the functions that can be called from the UI like `fetchAllFeeds` and `fetchMyGravatar` receive a non-null `FeedData` that updates the `listener` before any network call.

You need to be careful when using `lateinit`; if you try to access its value without having it initialized, your app will crash with the exception:

```
UninitializedPropertyAccessException: lateinit property has not been initialized
```

Additionally, you can check if it's initialized:

```
if (::listener.isInitialized) {  
    //Do something  
}
```

But this is seen as smelly code and not advised.

In Swift, there's no `lateinit` keyword for initialization. Instead, you need to use the operator `!`:

```
private var listener: FeedData!
```

Under the hood, `listener` is defined as **optional**. Be careful — before accessing its value you need to define it. Otherwise your app will crash.

The equivalent to see if it's initialized:

```
if listener != nil {  
    //Do something  
}
```

Nullability

Perhaps the most known trait of Kotlin is its nullability. Ideally, there are no more **NullPointerExceptions** — in other words, exceptions triggered by calls to objects that don't exist. The word "ideally" is needed here since developers have the final word and can always go against what the language advises.

To define if a variable can be null, you need to use the `?` operator.

You can see that `listener` has the type of `FeedData`, but its value can be `null`. Now, try to make any operation on this object. On `fetchAllFeeds`, before the `Logger` call, add:

```
listener.onMyGravatarData(GravatarEntry())
```

You're sending an empty `GravatarEntry` since this parameter cannot be `null`.

Looking at this instruction, you can see there's a red underline under the `.` with the message:

Only safe (?) or non-null asserted (!!.) calls are allowed on a nullable receiver of type `FeedData`?

Since this variable might be null, you shouldn't do any operation before checking its value. There are two different possibilities here:

1. Explicitly say that it won't be null. You can use the character `!` to tell the compiler that this value will never be `null`, so you can make any call that you need:

```
listener!!.onMyGravatarData(GravatarEntry())
```

Going against the language rules is never a good idea, so try to run away from this implementation.

The equivalent in Swift to this annotation is just to use a single `!`.

2. Only call the method if the value is not null. In this case, `listener` is mutable, so you can't just add an `if` condition to see if it's not null (since it might be changed by another thread). The solution is to use the `?` operator again. In this scenario it will only call `onMyGravatarData` if `listener` is not null:

```
listener?.onMyGravatarData(GravatarEntry())
```

Well, there might be a third possibility here. Don't make `listener` as nullable in the first place. :]

Additionally, you can also use `*?. let { ... }` as a verification to only run the code between brackets if the variable that you're accessing is not null:

```
listener?.let {  
    it.onMyGravatarData(GravatarEntry())  
}
```

it corresponds to `listener`.

Similarly, in Swift you could do this:

```
if let listener = listener {  
    listener.onMyGravatarData(GravatarEntry())  
}
```

String Interpolation

With string interpolation, you can easily concatenate strings and variables together. On `fetchAllFeeds`, you'll iterate over content and call `fetchFeed` with the platform and feed URL. Before this block of code, add:

```
Logger.d(TAG, "Fetching feed: ${feed.platform}")
```

To print the result of `feed.platform`, you need to add brackets to the instruction that you want to execute.

What happens if you don't add those brackets, and instead you have:

```
Logger.d(TAG, "Fetching feed: $feed.platform")
```

Compile your app and switch to the Logcat view to confirm that this log will show you the feed object followed by ".platform".

Type Inference

If you declare a variable and assign it a specific value, you don't need to define its type. Kotlin is capable of inferring it in most cases. If you look at the variables declared at `FeedPresenter.kt`, you can see that `json` uses type inference, but `content` doesn't.

Try to remove the type from content declaration. Android Studio immediately underlines this expression, and if you check the error it says:

```
Not enough information to infer type variable T
```

This is because `decodeFromString` doesn't know which type of object it should return. When you define the type at the variable level, `decodeFromString` uses it to know which objects it should return. You can define this type directly on the function if you want to use type inference on the variable declaration:

```
val content by lazy {
    json.decodeFromString<List<KodecoContent>>(KODECO_CONTENT)
}
```

Type Checks

Both languages use the `is` to check if an object is from a specific type.

Cast

Casting a variable is similar in both languages. You just need to use the keyword as followed by the type of the class that you want to cast.

Converting Between Different Types

You can easily convert between primitive types by calling `.to*()` for the type that you want:

```
// Convert String to Integer  
"kodeco".toInt()  
  
// Convert String to Long  
"kodeco".toLong()  
  
// Convert String to Float  
"kodeco".toFloat()  
  
// Convert Int to String  
42.toString()  
  
// Convert Int to Long  
42.toLong()  
  
// Convert Int to Float  
42.toFloat()
```

If you're dealing with custom objects, you can always create an extension function for it.

Extension Functions

As the name suggests, extension functions allow you to create additional behaviors for existing classes. Imagine that you want to add a method that needs to be available for all **String** objects, and it should return "Kodeco" when called:

```
fun String.toKodeco(): String {  
    return "Kodeco"  
}
```

This is it. You use the type that you want to extend, followed by the method name. Now this function is available for all **String** objects.

You can try this by adding the previous function and a new log to a **String** variable on **FeedPresenter.kt** – for instance to **KODECO_CONTENT**:

```
init {
    Logger.d(TAG, "content=${KODECO_CONTENT.toKodeco()}")
}
```

You can confirm in the Logcat that the output of this call will be similar to:

```
FeedPresenter | content=Kodeco
```

Comparing Objects

You can compare objects by reference through the use of `==` or by content `==`.

Control Flow

Although the syntax is quite similar in both languages, you'll find that Kotlin gives you powerful expressions that you can use.

if... else

This condition check is similar in both languages. If you open **GetFeedData.kt**, you can see different functions that use **if... else**.

The `invokeFetchKodecoEntry` only adds the parsed object if it isn't null:

```
if (parsed != null) {
    feed += parsed
}
```

Moreover, you don't need to add brackets when it's a single instruction.

Alternatively, you could just write:

```
if (parsed != null)
    feed += parsed
```

Or even inline:

```
if (parsed != null) feed += parsed
```

switch

It doesn't exist in Kotlin. Alternatively, you can use `when` which is similar.

when

when is a condition expression that supports multiple and different expressions. You can see an example of how to use it on the **ImagePreview.kt** file, which is inside the **components** folder of the **androidApp**:

```
when (painter.state) {
    is ImagePainter.State.Loading -> {
        AddImagePreviewEmpty(modifier)
    }
    is ImagePainter.State.Error -> {
        AddImagePreviewError(modifier)
    }
    else -> {
        // Do nothing
    }
}
```

In this case, you're checking the current state of an image that's being downloaded from the internet, and adding different composable depending on if its value is either Loading or Error.

Since all of these expressions are single-line, you could drop the brackets.

for

Back to the **FeedPresenter.kt** file from the **shared** module. You can find the for loop on **fetchAllFeeds**:

```
for (feed in content) {
    fetchFeed(feed.platform, feed.url)
}
```

Here, you're iterating through all the values of **content**. Starting with the first element in the list, on each iteration you'll get a different element that you can access through **feed**.

Additionally, there are other possibilities to write the same for cycle:

```
for (index in content.indices) {
    val feed = content[index]
    fetchFeed(feed.platform, feed.url)
}
```

That uses the `index` to go through all elements. Or, you could get the `index` and the `feed` directly via:

```
for ((index, feed) in content.withIndex()) {  
    fetchFeed(feed.platform, feed.url)  
}
```

Or, you can even get the `feed` from:

```
for (index in 0..content.size) {  
    val feed = content[index]  
    fetchFeed(feed.platform, feed.url)  
}
```

These are all possibilities that iterate through the list of all the elements from `content` to get the same result.

while

The `while` and `do...while` loops are similar to Swift. You just need to add the condition that should end the cycle and the code that should run while it isn't met.

```
while (condition) {  
    //Do something  
}  
  
do {  
    //Something  
} while (condition)
```

The difference between both is the same as in Swift: if the condition is false on `while` the code block will never run, while `do...while` will run once.

This is similar in Swift to the `while` and `repeat-while` loop:

```
while condition {  
    //Do something  
}  
  
repeat {  
    //Something  
} while condition
```

Ternary Operator

It doesn't exist in Kotlin. This is something that has been under discussion (<https://discuss.kotlinlang.org/t/ternary-operator/2116>) for a couple of years now, and the result has always been the same: you can achieve the same solution by using an inline **if... else** condition.

Collections

Kotlin supports different types of collections: arrays, lists and maps. These are immutable by default, but you can use their mutable counterpart by using: **mutableList** and **mutableMap**.

Although on Swift you can change the mutability of a list or a dictionary if you declare it with **let** (immutable) or **var** (mutable), the same is not valid for Kotlin. As mentioned above, you've got the **list** and **map** for immutable variables, and **mutableList** and **mutableMap** for mutable.

Lists

You can easily create a list in Kotlin from a source set by calling **listOf** and add the items as parameters. You can see an example where this is done on the **MainScreen.kt** file inside the **androidApp/main** folder:

```
val bottomNavigationItems = listOf(  
    BottomNavigationScreens.Home,  
    BottomNavigationScreens.Bookmark,  
    BottomNavigationScreens.Latest,  
    BottomNavigationScreens.Search  
)
```

In this case, this is the list of items in the navigation bar.

Imagine that you want to add a new item to this list. You can't. There's no add or remove method, since the **list** object is immutable. What you can do is create a mutable list:

```
val bottomNavigationItems = mutableListOf(  
    BottomNavigationScreens.Home,  
    BottomNavigationScreens.Bookmark,  
    BottomNavigationScreens.Latest,  
    BottomNavigationScreens.Search  
)
```

Or, you can convert the existing list to `mutableList`:

```
val bottomNavigationItems = listOf(
    BottomNavigationScreens.Home,
    BottomNavigationScreens.Bookmark,
    BottomNavigationScreens.Latest,
    BottomNavigationScreens.Search
).toMutableList()
```

Now you can add or remove elements to the list. Try removing the Search option:

```
bottomNavigationItems.remove(BottomNavigationScreens.Search)
```

Or, you could just use the minus and equal sign:

```
bottomNavigationItems -= BottomNavigationScreens.Search
```

Arrays

Arrays are mutable, but they have fixed size. Once you've created one, you can't add or remove elements. Instead, you change its content. Using the previous example, you can create an `arrayOf` with an initial number of items:

```
val bottomNavigationItems = arrayOf(
    BottomNavigationScreens.Home,
    BottomNavigationScreens.Bookmark,
    BottomNavigationScreens.Latest,
    BottomNavigationScreens.Search
)
```

And then if you want to change the value of one of its indexes:

```
bottomNavigationItems[0] = BottomNavigationScreens.Bookmark
bottomNavigationItems[1] = BottomNavigationScreens.Home
```

Maps (Swift Dictionaries)

Similar to what you've read in the examples above, you can create a map using `mapOf` function. It receives a `Pair` of objects that you can add or remove.

Modify the previous example to create a map containing the index as key and the screen as value:

```
val bottomNavigationItems = mapOf(
    0 to BottomNavigationScreens.Home,
    1 to BottomNavigationScreens.Bookmark,
    2 to BottomNavigationScreens.Latest,
```

```
    3 to BottomNavigationScreens.Search  
)
```

You can get any value on the map by using its key:

```
// Returns BottomNavigationScreens.HOME  
bottomNavigationItems[0]  
  
// Returns BottomNavigationScreens.HOME  
bottomNavigationItems.get(0)
```

Or, you can create a mutable map:

```
val bottomNavigationItems = mutableMapOf(  
    0 to BottomNavigationScreens.Home,  
    1 to BottomNavigationScreens.Bookmark,  
    2 to BottomNavigationScreens.Latest,  
    3 to BottomNavigationScreens.Search  
)
```

Or by converting toMutableMap:

```
val bottomNavigationItems = mapOf(  
    0 to BottomNavigationScreens.Home,  
    1 to BottomNavigationScreens.Bookmark,  
    2 to BottomNavigationScreens.Latest,  
    3 to BottomNavigationScreens.Search  
.toMutableMap()
```

Maps are equivalent to Swift's dictionaries.

Extra Functionalities

All of these collections also provide a set of functions that allow you to easily iterate and filter objects. Here's a short list of the ones that you might use daily:

- `*.isEmpty()` returns `true` if the collection is empty, `false` otherwise. On the contrary, you also have `*.isNotEmpty()` that returns the opposite values.
- `*.filter { ... }` allows filtering your collection according to a specific predicate.
- `*.first { ... }` returns the first object that meets the condition between brackets. There's also `*.firstOrNull { }` that returns `null` if there's no object that matches the predicate.

- `*.forEach { ... }` iterates over the collection.
- `*.last { ... }` is similar to `first`, but this time the last object found is returned.
- `*.sortBy { ... }` returns a new ordered list according to the predicate defined. You also can get the list on its descending order by calling: `*.sortByDescending { ... }`.

Classes and Objects

You can use different approaches to define class and objects in Kotlin depending on your use case.

Classes

You can create a class by using the keyword `class` followed by its name and any parameters that it might receive. If you open the `FeedPresenter.kt` file, you'll see:

```
class FeedPresenter(private val feed: GetFeedData)
```

Typically, each word of a class has an uppercase letter. In this case, `feed` has `val` set, so it can be accessed from any function on `FeedPresenter` scope.

Data Classes

You can create a data class by using the keyword `data` before declaring a class. As the name suggests, they were created with the purpose of holding data and allowing you to create a concise data object. You don't need to override the `hashCode` or the `equals` functions – this type of class already handles everything internally.

You can see an example of a data class if you open `KodecoContent.kt` from the `data/model` folder on the `shared` module:

```
data class KodecoContent(  
    val platform: PLATFORM,  
    val url: String,  
    val image: String  
)
```

However, they have a couple of differences when compared with a generic class: you can't inherit a data class or define it as abstract.

Sealed Classes

If you define a class or an interface as **sealed**, you can't extend it outside its package. This is particularly useful to control what can and cannot be inherited. Open the **BottomNavigationScreens.kt** file inside **ui/main** in the **androidApp**:

```
sealed class BottomNavigationScreens(  
    val route: String,  
    @StringRes val stringResId: Int,  
    @DrawableRes val drawResId: Int  
)
```

If you try to extend this class in any other class in the project, you'll see an error similar to the following:

```
Inheritor of sealed class or interface declared in package  
com.kodeco.learn.ui.home but it must be in package  
com.kodeco.learn.ui.main where base class is declared
```

To create an object of this sealed class:

```
object Home : BottomNavigationScreens("Home",  
    R.string.navigation_home, R.drawable.ic_home)  
  
object Search : BottomNavigationScreens("Search",  
    R.string.navigation_search, R.drawable.ic_search)
```

In this case, they represent the navigation tabs.

Although **sealed** classes don't exist in Swift, you can create a similar concept with enum:

```
enum BottomNavigationScreens {  
    struct Content {  
        let route: String  
        let stringResId: Int  
        let drawResId: Int  
    }  
}
```

There's no **@StringRes** or **@DrawableRes**, since these annotations are Android-specific.

Additionally, to create the corresponding objects, you can do something similar to:

```
enum BottomNavigationScreens {
    ...
    case home(route: String, stringResId: Int, drawResId: Int)
    case search(route: String, stringResId: Int, drawResId: Int)
}
```

Default Arguments

Kotlin allows you to define default arguments for class properties or function arguments. For instance, you can define the default value for `platform` to always be `PLATFORM.ALL`. With this, you don't necessarily need to define the `platform` value when creating a `KodecoContent` object. In these scenarios, the system will use the default one.

```
data class KodecoContent(
    val platform: PLATFORM = PLATFORM.ALL,
    val url: String,
    val image: String = "")
```

And to create this object:

```
val content = KodecoContent(
    url = "https://www.kodeco.com")
```

Now, if you print its content:

```
// > ALL
Logger.d(TAG, "platform=${content.platform}")
// > https://www.kodeco.com
Logger.d(TAG, "url=${content.url}")
// > ""
Logger.d(TAG, "image=${content.image}")
```

Singletons

To create a singleton in Kotlin, you need to use the keyword `object`. The `ServiceLocator.kt` file — since it deals with object initialization — is one such example:

```
public object ServiceLocator
```

This guarantees that at all times, you'll only have one reference to `ServiceLocator` throughout the scope of your app.

Interfaces

Interfaces are similar to Swift protocols. They define a set of functions that any class or variable that uses them needs to declare.

Open `FeedData.kt` from `domain/cb` in the shared module:

```
public interface FeedData {  
  
    public fun onNewDataAvailable(items: List<KodecoEntry>,  
        platform: PLATFORM, e: Exception?)  
  
    public fun onNewImageUrlAvailable(id: String, url: String,  
        platform: PLATFORM, e: Exception?)  
  
    public fun onMyGravatarData(item: GravatarEntry)  
}
```

`FeedData` defines three different functions that will be called when there's a network response. They're declared on `FeedViewModel.kt` (inside `androidMain/home`) and used to notify the UI that there are new data available.

Functions

Kotlin supports different types of functions:

(Non-Line) Functions

These functions are the ones that are more common to find in any source base. They're quite similar to Swift `func`, but in Kotlin, this keyword loses a letter because it's `fun`. :]

You can see different examples of functions in `FeedPresenter.kt`:

```
public fun fetchAllFeeds(cb: FeedData) {  
  
    listener = cb  
  
    for (feed in content) {  
        fetchFeed(feed.platform, feed.url)  
    }  
}
```

A function can also return an object. If you open **FeedAPI.kt** and look at `fetchKodecoEntry`, you can see that it's returning a `HttpResponse` object. Moreover, since there's only one instruction, you don't need to add brackets and the return can be written on the same line. You just need to add the `=` sign:

```
public suspend fun fetchKodecoEntry(feedUrl: String):  
    HttpResponse = client.get(feedUrl)
```

Lambda Expressions

Lambda expressions allow you to execute specific code blocks as functions. They can receive parameters and even return a specific type of object. You can see two of them on `fetchFeed`: `onSuccess` and `onFailure` parameters on **FeedPresenter.kt**.

```
private fun fetchFeed(platform: PLATFORM, feedUrl: String) {  
    GlobalScope.apply {  
        MainScope().launch {  
            feed.invokeFetchKodecoEntry(  
                platform = platform,  
                feedUrl = feedUrl,  
                onSuccess = { listener?.onNewDataAvailable(it, platform,  
                    null) },  
                onFailure = { listener?.onNewDataAvailable(emptyList(),  
                    platform, it) }  
            )  
        }  
    }  
}
```

Both of these expressions receive an `it` parameter. In the first case, it's a list of `KodecoEntry`, and in the second it's an `Exception`. Alternatively, you could define this expression like the following to better identify what `it` really is:

```
onSuccess = { list ->  
    listener?.onNewDataAvailable(list, platform, null)  
}
```

Higher-Order Functions

Higher-order functions support receiving a function as an argument.

A good example of this type of function is the `onSuccess` and `onFailure` arguments on `fetchFeed`. If you analyze these instructions, you can see that `onSuccess` and `onFailure` receive different `it` objects.

Open **FeedData.kt** and look for the `onDataAvailable`:

```
public fun onDataAvailable(items: List<KodecoEntry>,  
    platform: PLATFORM, e: Exception?)
```

The `it` in `onSuccess` is a list of `KodecoEntry`, while `onFailure` is an `Exception`. Navigate to `invokeFetchKodecoEntry` in **GetFeedData.kt** and look for the function:

```
public suspend fun invokeFetchKodecoEntry(  
    platform: PLATFORM,  
    feedUrl: String,  
    onSuccess: (List<KodecoEntry>) -> Unit,  
    onFailure: (Exception) -> Unit  
)
```

You can see that both parameters receive a function, but in one the type is a list of `KodecoEntry` and in the other it's an exception.

Inline Functions

If your app calls a high-level function multiple times, it can have an associated performance cost. Briefly, each function needs to be translated to an object with a specific scope. Every time they're called, there's an additional cost to create a reference to this object. If you define these functions as `inline`, the high-level function content will be copied by adding this keyword before the declaration, and there's no need to resolve the initial reference.

Suspend Functions

To use the `suspend` function, you need to add the **Coroutines** library to your project. You can run, stop, resume and pause a suspended function. This is why they're ideal for asynchronous operations — and why the app network requests use it:

```
public suspend fun fetchKodecoEntry(feedUrl: String):  
    HttpResponse = client.get(feedUrl)
```

This is similar to Swift's `async...await` functions.

Kotlin and Swift Syntax Table

You can find a comparison table between both languages in the materials repository (<https://github.com/kodecocodes/kmpf-materials/blob/editions/2.0/15-appendix-a-kotlin-a-primer-for-swift-developers/README.md>).

Where to Go From Here?

Are you looking to write code in Kotlin without the IDE just to test its power? JetBrains has the Kotlin Playground (<https://play.kotlinlang.org/>) that allows you to test some basic functions.

If you want to learn more about both languages, you've got the Kotlin (<https://www.kodeco.com/books/kotlin-apprentice>) and Swift (<https://www.kodeco.com/books/swift-apprentice>) Apprentice books that teach you everything you need to know about both languages in detail.

Appendix B: Debugging Your Shared Code From Xcode

By Carlos Mota

Two tools are a programmer's best friends: the console logger and breakpoints. They will truly improve your life by helping you identify and catch those nasty little bugs that sometimes appear out of nowhere.

You've used the logger throughout this book. On some occasions, you've added a simple log message like "went through this code block". On other occasions, you've printed all the variables in a method. Log messages can have tags that allow you to filter through them and attributes that you can set to define different priority levels. This can help you easily understand where something went wrong.

Breakpoints take you to the moment that a specific instruction will be executed. You can see all the steps that lead to this stop and all that will succeed. Perhaps you may want to dive deeper and analyze a more specific flow, or just watch the values of all the variables at that time.

Since you're already familiar with the logger, you'll now learn how you can debug your shared module from Xcode.

Debugging the Shared Module

Both Android Studio and Xcode have great debugging capabilities. The native code is simple to debug on both IDEs. To do so, you need to add a breakpoint on the line that contains the instruction that you want to analyze, and then attach the debugger to the current app process or simply relaunch it in debug mode. For both options, the app halts before executing that instruction.

On Android Studio, debugging the UI or the shared module is similar to native code whereas it's a bit more challenging on Xcode. Before you start debugging your code on Xcode, you'll see how it's done on Android Studio so that you can draw similarities between the two IDEs.

Debugging in Android Studio

The steps required to debug the shared module are the exact steps described earlier – but instead of adding a breakpoint on the Android app, you need to add it on the **shared** module.

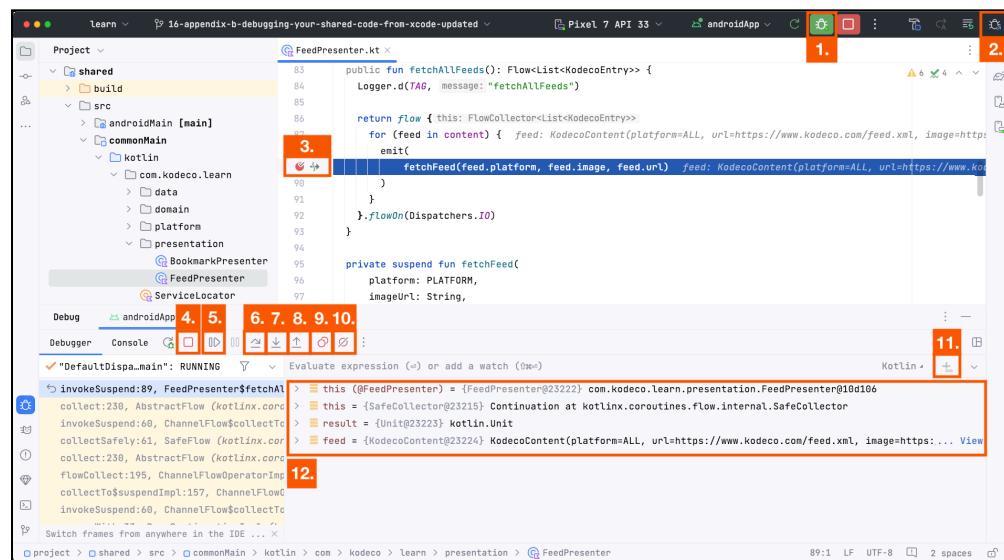


Fig. B.1 – Android Studio Debugger Window

Before reaching a breakpoint, the app halts. You'll see a screen similar to the one in the image. Here's a step-by-step description of what you can do in debug mode:

1. Relaunches the app in debug mode.
2. Attaches the debugger to the app, without relaunching it.
3. Line breakpoint. Identifies where the app should suspend. This instruction will only be executed if you continue the debug process.
4. Stops the app.
5. If you want to resume the app, you can click on this green arrow. The app will halt again at the next breakpoint.
6. Step over this instruction. Allows you to navigate to the next instruction inside the same scope.
7. Step into. Shows you the next method that's going to be invoked. It doesn't need to be at the same level as this line of code if the next instruction is a call to another class. This is contrary to step over, which would jump to the next instruction inside the same method.
8. Step out from the current instruction. The rest of the code will execute, and the debugger will halt again when the method that was suspended executes.
9. Shows the list of breakpoints that you've set.
10. Mutes breakpoints. While enabled, the app won't suspend on any breakpoint.
11. Add a new watch. With this option, you can inspect any property or run any method that's available on the current running scope.
12. The list of watchers. When the app halts, a list of variables that you can analyze is immediately shown. All the watchers that you add in the previous point will also be displayed here.

And that's it! You can find the final project of the book in this appendix materials. Open it in Android Studio, add a breakpoint and run the app in debug mode. Try out the actions that you have available, follow a network request and inspect its response — and have fun. :]

Debugging in Xcode

As you can see, debugging the shared module from Android Studio is simple. If you want to do the same thing in Xcode, it's more...challenging. :)

If you want to debug the iOS UI, it uses the process you're already familiar with. You just need to set a breakpoint, and the next time the app goes through that instruction it will halt.

But if you want to debug the shared module, it will take you a couple more steps. First, you'll need to install the Kotlin Native Xcode Support plugin. You can find it on the Touchlab (<https://touchlab.co/>) GitHub repository (<https://github.com/touchlab/xcode-kotlin>) or in the materials section of this appendix.

Note: A special thank you to Touchlab for maintaining this plugin throughout multiple Xcode releases and, for going the extra mile, and having it ready for the release of this book update. The last version of the plugin is from October 2023 and supports Xcode 15.0. It's important to mention, that there's no guarantee, for now, that it will support future Xcode versions.

Uninstalling Previous Versions

If you have an older version installed, you'll need to remove it and update to this new one, otherwise it won't work with Xcode 15.0.

Depending on the plugin version that you have on your machine, you may need to uninstall it differently:

1. If you've installed it using Homebrew (CLI), close Xcode and run:

```
xcode-kotlin uninstall
```

2. If you run a script file to install the plugin, you won't have any option to uninstall it from Xcode. The best solution is to close the IDE and go directly to the directory where they're installed:

```
~/Library/Developer/Xcode/Plug-ins/
```

Remove the ones that are no longer needed. In this case, it's the **Kotlin.ideplugin**.

Note: If you update Xcode you might need to re-install the **xcode-kotlin** plugin.

Installing the Kotlin Native Xcode Support Plugin

To install the plugin, you need to first close Xcode. Due to some limitations with Xcode 15.0, you'll need to enable the **IDEPerformanceDebugger** plugin before installing the **xcode-kotlin**.

Open the command line and enter:

```
defaults write com.apple.dt.xcode IDEPerformanceDebuggerEnabled  
-bool true && killall -u $USER cfprefsd
```

Then open Xcode, and close it afterward. You don't need to open any project.

Return to the command line and run:

```
defaults delete com.apple.dt.xcode IDEPerformanceDebuggerEnabled  
&& killall -u $USER cfprefsd
```

The command above disables the **IDEPerformanceDebugger** that you had enabled before.

You can now install the **xcode-kotlin** plugin normally via Homebrew:

```
brew install xcode-kotlin
```

Once installed, execute:

```
xcode-kotlin install
```

Here's what you'll see in the console:

```
Installing 1.3.0.  
Synchronizing plugin compatibility list.
```

This indicates you've installed the plugin successfully.

The next time you open Xcode, you'll see the following prompt:

Note: The “Kotlin.ideplugin” code bundle is not provided by Apple. Loading code not provided by Apple can have a negative effect on the safety and stability of Xcode or related tools.

Every time you install a third-party plugin, you'll see a similar notification. Since Apple didn't release or validate it, they cannot guarantee its behavior.

Until there's direct support on the IDE, you need to use this plugin — so click on **Load Bundle**. When this process ends, open the **project** from materials.

Compile and install the app to guarantee that everything is working as expected.

Now that you have the project up and running, on the left side panel of Xcode, right-click on **iosApp** and select **New Group**. This will add a new folder to the project. Rename it to **Shared**.

You're going to add the source code of the **shared** module that your iOS app uses. Once again, right-click over the newly added **Shared** folder and select **Add Files to “iosApp”**.

A new window will open. Navigate backward to the **shared** module, and in this folder select the **commonMain** and **iosMain** directories. Select the option **Create folder references** to avoid copying those files to the project, and then click on “Add”.

Your screen will be similar to this one:

The screenshot shows the Xcode interface with the following details:

- File Structure:** The left sidebar shows the project structure under "iosApp". It includes "Shared", "commonMain", "iosMain", and "Test". "commonMain" contains "kotlin" which further contains "com", "kodeco", "learn", "domain", "platform", and "presentation". "presentation" contains "BookmarkPresenter" and "FeedPresenter". "iosMain" contains "kotlin", "com", "kodeco", "learn", "platform", and "Database.ios". "Test" contains "iosApp" and "iosapp".
- Code Editor:** The main window displays the "FeedPresenter.kt" file. The code is as follows:

```

14 class FeedPresenter(private val feed: GetFeedData) {
15
16     private val json = Json { ignoreUnknownKeys = true }
17
18     val content: List<KodecoContent> by lazy {
19         json.decodeFromString(KODECO_CONTENT)
20     }
21
22     @NativeCoroutines
23     public fun fetchAllFeeds(): Flow<List<KodecoEntry>> {
24         Logger.d(TAG, "fetchAllFeeds")
25
26         return flow {
27             for (feed in content) {
28                 emit(
29                     fetchFeed(feed.platform, feed.image, feed.url)
30                 )
31             }
32         }.flowOn(Dispatchers.IO)
33     }
34
35 }
```
- Breakpoint:** A blue circle highlights line 89, indicating a breakpoint has been set.
- Toolbars and Status:** The top bar shows "iosApp" and "iPhone 14 Pro". The status bar at the bottom right says "Line: 70 Col: 1".

Fig. B.2 – Xcode Folder Hierarchy

The Kotlin classes now have syntax highlight, which makes it easier to read the code. Open **FeedPresenter.kt** and identify the method's visibility, strings, for cycle, nullability, etc.

Time to test the debugging. In this file, add a breakpoint on the call to `fetchFeed` inside the `for` cycle of `fetchAllFeeds`.

Note: To add a breakpoint on Xcode, you just need to click on the line number. Here, it can have two different states: disabled if it has a transparency and enabled in case it doesn't. To remove a breakpoint, click on it and drag it to the right.

Compile and run the app.

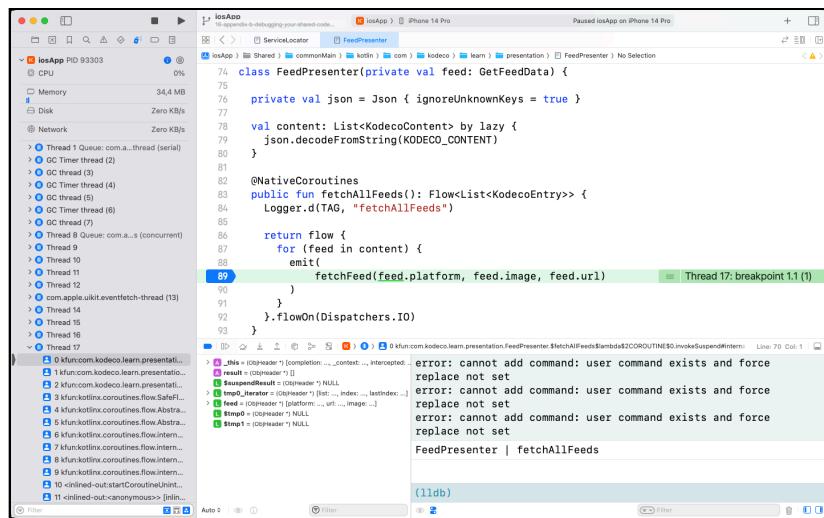


Fig. B.3 – Xcode Halt At a Breakpoint

Xcode suspends your app just before running this instruction.

Debugging Your iOS App

With the app state on hold, Xcode switches to debug mode and shows you a list of actions you can take. As you can see, they're similar to the ones that Android Studio offers:

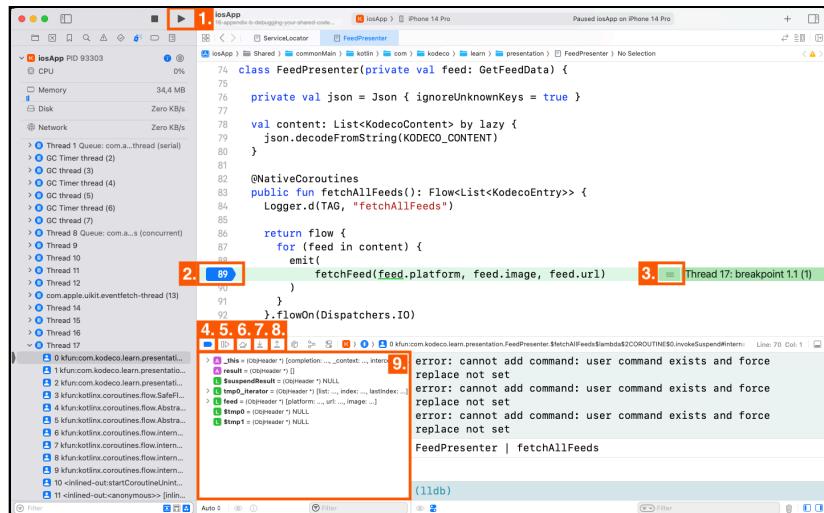


Fig. B.4 – Xcode Debugger Window

1. This action compiles and runs the application.
2. As soon as you add a breakpoint, the next time that line is about to be executed, the app will automatically halt. You don't necessarily need to relaunch or attach the debugger.
3. By dragging this action up, you can move the app execution to a previous line.
4. This action allows you to disable all breakpoints.
5. When the app reaches a breakpoint, it suspends. No more code will be executed without a user action. This one allows the app to resume and continue until it finds another breakpoint.
6. Step over. You can go to the next instruction without needing to add another breakpoint.
7. With the step into action, you can access the method that's going to be invoked.
8. Step out from the current execution. The app will continue to run until this method returns.
9. By right-clicking in this area, you can select "Add Expression..." and the instruction that you want to monitor will be displayed in this list.

As you can see, the Kotlin Native Xcode Support Plugin is a great tool to debug your business logic from Xcode.

Where to Go From Here?

Well done! Now that you've seen how you can debug your shared module, why not dive deeper into iOS (<https://www.kodeco.com/18770184-ios-debugging-fundamentals>) or Android (<https://www.kodeco.com/9261991-beginning-android-debugging>) debugging through these video courses? Or if you're looking for more advanced concepts, try the Advanced Apple Debugging & Reverse Engineering (<https://www.kodeco.com/books/advanced-apple-debugging-reverse-engineering>) book.

In the next appendix, you can learn how to reuse your UI between Android and desktop. Now that you know how to share your business logic, see how you can also share your Compose UI.

See you there. :]

Appendix C: Sharing Your Compose UI Across Multiple Platforms

By Carlos Mota

Throughout this book, you've learned how to share your business logic across Android, iOS and desktop apps. What if you could go a step further and also share your Compose UI?

That's right — along with Kotlin Multiplatform, you now have **Compose Multiplatform**, which allows you to share your Compose UI with Android and desktop apps.

Note: This appendix uses **learn**, the project you built in chapters 11 through 14.

Setting Up an iOS App to Use Compose Multiplatform

To follow along with the code examples throughout this appendix, download the project and open **16-appendix-b-sharing-your-compose-ui-across-multiple-platforms/projects/starter** with Android Studio.

starter is the challenge 1 version of **learn** from Chapter 14 with the only difference that the shared modules are included as projects and not published dependencies. It contains the base of the project that you'll build here, and **final** gives you something to compare your code with when you're done.

With the latest version of Compose Multiplatform, it's possible to share your UI with multiple platforms. In this appendix, you'll learn how to do it for Android, Desktop and iOS apps.

Although, you can find alternative solutions to create an iOS app with Compose Multiplatform, the one that you're going to use in this section is the one suggested by JetBrains, which uses the **compose-multiplatform-template** (<https://github.com/JetBrains/compose-multiplatform-template>), created and maintained by them.

Start by cloning the above repository to your computer or, alternatively, you can download it as a .zip file, and extract its content. Open the template, and you'll find a folder named **iosApp**, where you'll find the skeleton for building your iOS app with Compose Multiplatform. Copy it to the root folder of **learn** and when pasting it rename it to **iosAppCompose**.

Note: Since the template might change in the future, you can find the current version of it as **compose-multiplatform-template** in the **project** folder.

Your project structure should now be similar to this one:



Fig. B.1 – Project view hierarchy

Open the **iosApp.xcodeproj** file located on **iosAppCompose** with Xcode. Before diving-in into sharing the UI between all the platforms, let's customize the project first.

Open **ContentView.swift**. Here is the entry point for the (Compose) screen to be loaded. It's done via the `makeUIViewController` function, in this template, which internally calls `Main_iosKt.MainViewController()`. You'll create this implementation later in the chapter. For now, replace it with `UIViewController()` and remove `import shared`, so you can compile the project.

Open **iosApp** and go to **BuildPhases**. Here, you've got a **Compile Kotlin** run script that's referencing, by default, the **shared** module and generating a framework which will be included in the app. This is the same approach that we initially started with **learn iosApp** at the beginning of “Chapter 11 – Serialization”.

Now, go to **BuildSettings** and search for **Linking - General**. Here you've got a setting named **Other Linker Flags**. Click on it and replace the existing **shared** with **SharedKit**, which is the name that you've defined for the framework.

Compile the project. You should see an empty screen similar to this one:

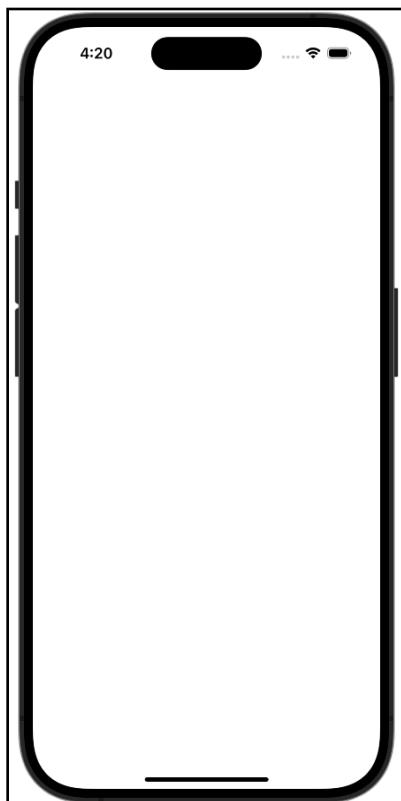


Fig. B.2 – iOS App

Depending on the current version of Java that you have set as your JAVA_HOME you might see an error similar to the following:

‘compileJava’ task (current target is 17) and ‘compileKotlin’ task (current target is 18) jvm target compatibility should be set to the same Java version.

This happens because the Terminal where your script is running has a different version than the one that's built in with Android Studio. You can change your JAVA_HOME to reflect the same directory, or you can just add the following before any instruction in the **Compile Kotlin** run script:

If you're not using the stable version of Android Studio, you'll need to change the directory to `Android Studio Preview.app`.

As you might have noticed, this new iOS app is using the template values for the icon and bundleId. Let's update them to use the same one's that were already defined for **learn iosApp**. Open the `Config.xcconfig` file located inside the **Configuration** folder and replace the existing content with:

If you now go to **iosApp** and click on the **General** section and look for the **Bundle Identifier** setting, which is under **Identity**, you'll see that both the app name and bundle ID were updated to `learn` and `com.kodeco.learn` respectively.

Finally, open **Assets** and remove the existing **AppIcon**. Open **Finder** and navigate to `iosApp/iosApp/Assets.xcassets` and copy the existing `AppIcon.appiconset` folder to `iosAppCompose/iosApp/Assets.xcassets`. Return to Xcode, and you should now see the Kodeco logo in **AppIcon**.

To confirm that your iOS app is ready, compile the project. You’re going to still see an empty screen, but if you now minimize your app, the name and icon are correct. :]



Fig. B.3 – iOS App icon

Updating Your Project Structure

To share your UI, you’ll need to create a new Kotlin Multiplatform module. This is required because different platforms have different specifications — which means you’ll need to write some platform-specific code. This is similar to what you’ve done throughout this book.

Start by creating a new KMP library. You can easily do this by clicking the Android Studio status bar **File**, followed by **New** and **New Module**.

Then, select **Kotlin Multiplatform Shared Module** and set:

- **Module Name:** shared-ui
- **Package Name:** com.kodeco.learn.ui
- **iOS framework distribution:** Regular framework

Click **Finish** and wait for the project to synchronize.

As you can see, there's a new **shared-ui** module in **learn**. Open the **settings.gradle.kts** file to confirm that it was added to your project.

Android Studio only has direct support for mobile targets. So, when you try to add a new module, and you're targeting other platforms — like desktop apps — you'll need to manually add these targets.

Open the **shared-ui build.gradle.kts** and add the **jvm** target inside the **kotlin** section, right after the **android** one:

```
jvm("desktop")
```

This is required — otherwise, you would only generate the **shared-ui** library for Android.

Replace the existing **android** target, and its configuration, with the new one:

```
androidTarget()
```

Scroll down to the **sourceSets** section and replace the existing implementation with:

```
getByName("commonMain") {
    dependencies {
        //put your multiplatform dependencies here
    }
}

getByName("commonTest") {
    dependencies {
        implementation(kotlin("test"))
    }
}
```

This change is to avoid compilation warnings. Previously, you were creating two variables: `commonMain` and `commonTest` that would never be used. Using `getByName` instead solves this.

Finally, scroll down to the `android` section on the bottom of the file and add Java 17 compatibility options:

```
compileOptions {
    sourceCompatibility = JavaVersion.VERSION_17
    targetCompatibility = JavaVersion.VERSION_17
}
```

Synchronize the project.

With the configuration set, click on the `shared-ui` folder and then **New > Directory** and select **desktopMain/kotlin**.

Look at the project structure. It should be similar to the one below:

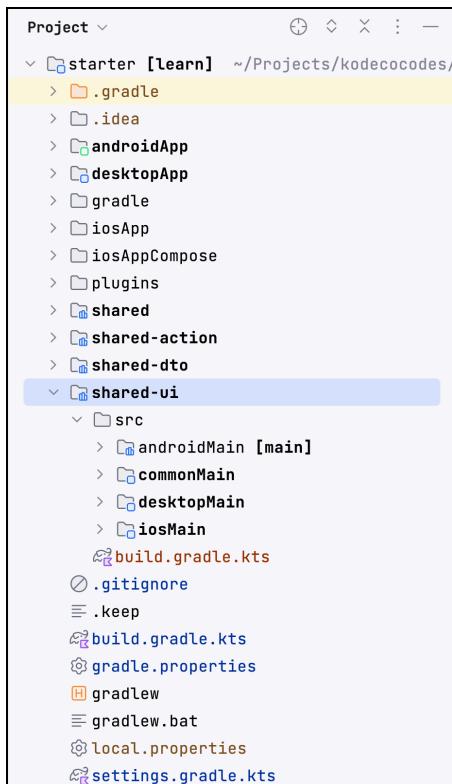


Fig. B.4 – Project view hierarchy

When generating a KMP library, Android Studio also adds **Platform.*.kt** inside all targets, and a **Greetings.kt** inside **commonMain**. You can remove these four files as you won't use them in this appendix.

Sharing Your UI Code

Although the code of both platforms is quite similar, the Android app uses platform-specific libraries. Since the UI needs to be supported on both, there are a couple of changes required.

Typically, the most common scenario is that you have an Android app built with Compose that you want to port to the desktop, or to iOS. So, you'll start by moving the UI from **androidApp** to **shared-ui**. In the end, you'll remove the classes that are no longer needed from **desktopApp**.

Before you start, there are a couple of things to consider:

- Android libraries that use the native SDK are platform-specific, so it won't be possible to use them on desktop apps.
- **shared-ui** follows the same principles of the **shared** module that you created before: the code needs to be written entirely in Kotlin — even its third-party libraries.

With that, it's time to start your journey. :]

Migrating Your Android UI Code to Multiplatform

Start by moving all the directories inside **androidApp/ui** into **shared-ui/commonMain/ui**. Don't move the **MainActivity.kt** file, since **activities** are Android-specific.

Note: Depending on the current view that you have selected for the project structure window on the left, you might not be able to move files directly to the right folder. To change this, select the window mode **Project Files**.

When prompted about how the move should be done, select “Move 8 packages to another package” and then before pressing refactor, confirm that you have the following settings selected:

- Search in comments and strings.
- Search for text occurrences.

Android Studio will open another window enumerating a couple of issues that were found during this process. They’re related to resources and libraries that need to be added to **shared-ui**. For now, don’t worry about this. Click **Continue**.

After this operation ends, move the **components** directory into **shared-ui/commonMain**. It should be at the same level as the **ui** folder. When prompted about possible problems that were detected, click once again in **Continue**.

You’ve got a **Utils.kt** file located inside **utils** folder that cannot directly be moved to **commonMain** because it’s using platform-specific code. In this case, it’s using Java libraries that won’t be available for **iOS**. You need to migrate this logic to Multiplatform.

Start by creating a **utils** folder in **com.kodeco.learn** for each one of the directories: **androidMain**, **commonMain**, and **iosMain**. For **desktopMain**, since you’ve manually added this target, you have to add the namespace first. You can easily do this by right-click on **desktopMain/kotlin** folder and select **New > Package** and add:

```
com.kodeco.learn.utils
```

With the folder structure set, go to **commonMain/utils**, create a **Utils.common.kt** file and add:

```
package com.kodeco.learn.utils

public const val TIME_FORMAT: String = "yyyy/MM/dd"

expect fun converterIso8601ToDate(date: String): String
```

Now that the **expect** function is declared, you need to create **actual** for each one of the targets. Starting with **androidMain** create the **Utils.android.kt** file and add the following code:

```
package com.kodeco.learn.utils

private const val TAG = "Utils"
```

```

@SuppressLint("ConstantLocale")
private val simpleDateFormat = SimpleDateFormat(TIME_FORMAT,
    Locale.getDefault())

actual fun converterIso8601ToDate(date: String): String {
    return try {
        val instant = date.toInstant()
        val millis = Date(instant.toEpochMilli())
        return simpleDateFormat.format(millis)
    } catch (e: Exception) {
        Logger.w(TAG, "Error while converting dates. Error: $e")
        "_"
    }
}

```

This code is similar to the one in **Utils.kt** from **androidApp**. You might have noticed that `kotlinx.datetime` and `Logger` imports aren't resolved, that's because both libraries haven't been imported in this new module. Open **build.gradle.kts** file from **shared-ui** and in the dependencies section of **commonMain** add:

```

api(project(":shared-logger"))

implementation(libs.kotlinx.datetime)

```

Click on **Sync** Now to add these libraries to the project.

Go back to **Utils.android.kt** and add the needed imports.

You can now remove **Utils.kt** from **androidApp**.

Go over to **desktopMain** and create the **Utils.desktop.kt** file inside the **utils** directory. Copy-paste the code that you've previously added to **Utils.android.kt**. Since they're both JVM targets, the only thing that you need to do here is to remove the `@SuppressLint` annotation along with its import and replace it with:

```

@SuppressLint("ConstantLocale")

```

Finally, go to **iosMain** and inside the **utils** folder create the **Utils.ios.kt** file and define its actual implementation:

```
package com.kodeco.learn.utils

actual fun converterIso8601ToDate(date: String): String
{
    val dateFormatter = NSDateFormatter()
    dateFormatter.dateFormat = TIME_FORMAT

    val nsdate = NSIS08601DateFormatter().dateFromString(date)
    return dateFormatter.stringFromDate(nsdate ?: NSDate())
}
```

When prompted, add the following imports:

```
import platform.Foundation.NSDate
import platform.Foundation.NSDateFormatter
import platform.Foundation.NSIS08601DateFormatter
```

Looking at the **androidApp** source folder, there are only two classes: **MainActivity** and **KodecoApplication**. All the other UI classes are now in **shared-ui**.

With your code moved to a different module, you need to import it to the **androidApp**. Otherwise, **MainActivity** won't be able to resolve its imports.

Open **build.gradle.kts** from **androidApp** and in the dependencies section, below **shared-action** add:

```
implementation(project(":shared-ui"))
```

Synchronize and wait for this operation to finish.

Once done, open **MainActivity.kt**. All the imports should now be resolved. Nevertheless, the view models still need to be addressed. Because they're Android-specific, you must use an external library to support them when targeting Multiplatform. You can read more about this in the "Using LiveData and ViewModels" section of this chapter.

You still have to migrate the resources files. However, to share the code through all the platforms, you'll need to use a new library named **moko-resources** (<https://github.com/icerockdev/moko-resources>) and make additional changes. The “Handling Resources” section in this chapter describes all the steps required.

Compose Multiplatform

Jetpack Compose was initially introduced for Android as the new UI toolkit where one could finally leave the XML declarations and the **findViewById** calls behind and shift towards a new paradigm – declarative UI.

Note: You can learn more about Jetpack Compose for Android in Chapter 3, Developing UI, and by reading the Jetpack Compose by Tutorials (<https://www.kodeco.com/books/jetpack-compose-by-tutorials>) from kodeco.com.

If you look at the official documentation (<https://developer.android.com/jetpack/androidx/releases/compose>) for Jetpack Compose, you can see that, at the time of writing, it's *composed* of seven libraries:

- **compose.animation**: Animations that you can easily use.
- **compose.material**: The material design system to use on components.
- **compose.material3**: The newest version of material design.
- **compose.foundation**: Contains the basic building Composables – Column, Text, Image, and so on.
- **compose.ui**: Handles input management, drawing, and layouts.
- **compose.runtime**: It's platform-agnostic, which means that it doesn't know what Android or UI are. It can be seen as a tree-management solution.
- **compose.compiler**: Transforms the @Composable into UI.

They can be structured into the following high-level diagram:

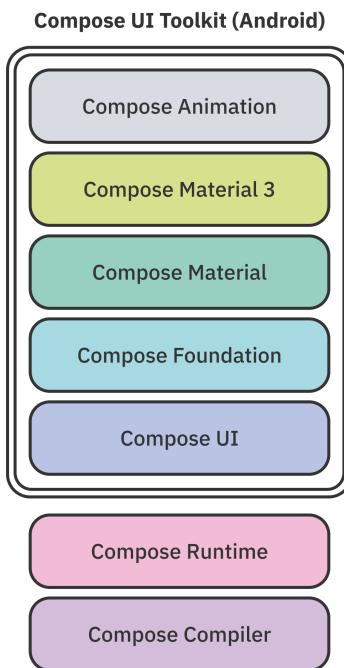


Fig. B.5 – Jetpack Compose high-level diagram

In this image, you can see that Jetpack Compose can be spliced into the:

- **Compose UI Toolkit**, which is platform-specific.
- **Compose Plugins**, which contains the Compose runtime and compiler.

By changing the Compose UI Toolkit, you can use Compose on other platforms.

With Compose Multiplatform (<https://www.jetbrains.com/lp/compose-multiplatform/>), JetBrains provides this exact support. It allows using Compose for desktop, iOS, and the web. The desktop app that you've been building throughout the book was built with this framework. In this chapter, you're going to share the same UI code across all the platforms, so the code from Android that you've moved to **shared-ui** needs to be migrated to Compose Multiplatform.

It's worth mentioning that to keep everything stable, the `org.jetbrains.compose` plugin replaces the `androidx.compose.*` artifacts with the ones from JetBrains. This is a temporary solution (<https://github.com/JetBrains/compose-jb/commit/d26c8f5c153d277923959a9285d727a5d85aae4c>) to deal with these different versions.

Migrating to Compose Multiplatform

Open the `BookmarkContent.kt` file from `shared-ui`. Here you'll see that the imports to `androidx.compose*` are not being resolved.

You need to add the Compose Multiplatform plugin and its libraries to solve this.

Open the `build.gradle.kts` file from `shared-ui`. In the `plugins` section, before `libs.plugins.androidLibrary`, add:

```
id("org.jetbrains.compose") version "1.5.1"
```

Since the project is using version catalogs, you can migrate the plugins added by the template, along with Compose Multiplatform:

```
alias(libs.plugins.kotlinMultiplatform)
alias(libs.plugins.jetbrains.compose)
alias(libs.plugins.androidLibrary)
```

Now add the Compose libraries the project is using. Scroll down to `sourceSets`, and inside `commonMain` dependencies section add:

```
api(compose.foundation)
api(compose.material)
api(compose.material3)
api(compose.runtime)
api(compose.ui)
```

Compose Multiplatform for iOS it's still experimental, therefore to use it, you need to enable it. Open `gradle.properties` file and add:

```
# Compose Multiplatform
org.jetbrains.compose.experimental.uikit.enabled=true
```

Synchronize the project and navigate back to `BookmarkContent.kt` file.

With none of the Compose imports marked red, it means the project can now resolve its Compose dependencies.

Updating Your Shared UI Dependencies

Now that **shared-ui** contains your app UI, it's time to add the missing libraries. Open the **build.gradle.kts** file from this module and look for `commonMain/dependencies`. Update it to include:

```
api(project(":shared"))
```

When prompted, click to synchronize the project, so it connects to both libraries.

Using Third-Party Libraries

Although Compose Multiplatform is taking its first steps, the community is following closely, releasing libraries that help make the bridge between Android and desktop apps.

Fetching Images

In the Android app, you were using Coil (<https://github.com/coil-kt/coil>) to fetch images. Unfortunately, it currently doesn't fully support Multiplatform (<https://github.com/coil-kt/coil/issues/842>), so you'll migrate this logic to a new one: Compose ImageLoader (<https://github.com/qdsfdhv/compose-imageloader>).

Compose ImageLoader uses Ktor (you can read more about this library in Chapter 12, "Networking") to fetch media. This API is similar to Coil, so you won't need to make many changes.

Open the **build.gradle.kts** file from **shared-ui** and in the `commonMain/dependencies` section, add:

```
implementation(libs.image.loader)
```

Synchronize the project.

In the **shared-ui/components** directory, open **ImagePreview.kt**. This file contains the logic required to fetch an image from the network and handles the request state: *success*, *loading*, and *error*.

The **AddImagePreview** Composable first checks if the `url` is empty. If it isn't, it will create a request to download the image via `rememberAsyncImagePainter`.

Compose ImageLoader API is similar to Coil. To fetch an image, you just need to make a couple of changes:

1. Update the above call to use `rememberImagePainter`:

```
val resource = painterResource(R.drawable.ic_brand)

val painter = rememberImagePainter(
    url = url,
    placeholderPainter = { resource },
    errorPainter = { resource }
)
```

The `placeholderPainter` and `errorPainter` correspond to the image that should be shown during the process of fetching an image and when this operation fails, respectively. For now, you won't be able to resolve both `painterResource` and the `R` class. You're going to see in the "Handling Resources" section how to address this.

2. Replace the existing Coil imports with:

```
import com.seiko.imageloader.rememberImagePainter
```

Using LiveData and ViewModels

`learn` was built using **LiveData** and **ViewModels** that are available in Android through the **runtime-livedata** library. Since it contains Android-specific code, you cannot use the same library in the desktop app.

Fortunately, there's a strong community around Kotlin Multiplatform and Compose that tries to reduce the gap between Android, desktop (and now iOS), and creates libraries that you can use on both platforms. One of these libraries is **PreCompose** (<https://github.com/Tlaster/PreCompose>) and it supports the Android Jetpack Lifecycle, ViewModel, LiveData and Navigation components in Multiplatform.

Now that you're familiar with **precompose**, open the **build.gradle.kts** file from the **shared-ui** module, and on **commonMain/dependencies**, add:

```
api(libs.precompose)
api(libs.precompose.viewmodel)
```

Synchronize your project. Once this operation ends, you'll need to update your app **ViewModels**. Open the **BookmarkViewModel.kt** file from the **shared-ui** module, and remove the imports that you no longer need:

```
import androidx.lifecycle.MutableLiveData
```

```
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
```

To import ViewModel() and the viewModelScope, you'll need to add the **precompose** version of both classes:

```
import moe.tlaster.precompose.viewmodel.ViewModel
import moe.tlaster.precompose.viewmodel.viewModelScope
```

The **MutableLiveData** class from this library is slightly different from the one in Android. Remove the `_items` variable, and update the `items` declaration to:

```
val items: MutableState<List<KodecoEntry>> =
mutableStateOf(emptyList())
```

Add the following imports for `MutableState`:

```
import androidx.compose.runtime.MutableState
import androidx.compose.runtime.mutableStateOf
```

Now, open **FeedViewModel.kt**. You'll have to make similar changes.

Remove the imports to Android libraries:

```
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
```

And add the ones from **precompose** for `ViewModel()` and `viewModelScope`:

```
import moe.tlaster.precompose.viewmodel.ViewModel
import moe.tlaster.precompose.viewmodel.viewModelScope
```

Finally, remove the `_profile` declaration and replace `profile` with:

```
val profile: MutableState<GravatarEntry> =
mutableStateOf(GravatarEntry())
```

And update its usage on `onMyGravatarData` to:

```
profile.value = item
```

And replace the import:

```
import androidx.lifecycle.MutableLiveData
```

With:

```
import androidx.compose.runtime.MutableState
import androidx.compose.runtime.mutableStateOf
```

With both view models updated, navigate to the **androidApp** and open the **MainActivity.kt** file. Here, look for their declaration and update it to:

```
private lateinit var bookmarkViewModel: BookmarkViewModel
private lateinit var feedViewModel: FeedViewModel
```

There's no support to call `by viewModel()` on **precompose**. Instead, you need to initialize them inside a Composable function. This is why they're set as `lateinit`. Inside `setContent`, add:

```
feedViewModel = viewModel {
    FeedViewModel()
}

bookmarkViewModel = viewModel {
    BookmarkViewModel()
}
```

When prompted, add:

```
import moe.tlaster.precompose.ui.viewModel
```

And move the view model's `fetch` calls to be after its initialization.

Finally, remove the call to `observeAsState()` since it's no longer necessary.

Don't forget to delete the now-unnecessary imports:

```
import androidx.activity.viewModels
import androidx.compose.runtime.livedata.observeAsState
```

Handling Navigation

The **precompose** library also handles navigation between different screens. In case of **learn**, the user can change between the tabs on the bottom navigation bar.

The desktop app already uses **precompose**, so there's nothing that you need to do there. However, Android was using different libraries, so you'll need to make a few changes here.

Open the **MainActivity.kt** file inside **androidApp**, and replace the class the activity extends with:

```
class MainActivity : PreComposeActivity()
```

You'll also have to remove the `androidx.*` imports:

```
import androidx.activity.ComponentActivity
import androidx.compose.setContent
```

And, add the ones from **precompose**:

```
import moe.tlaster.precompose.lifecycle.PreComposeActivity
import moe.tlaster.precompose.lifecycle.setContent
```

That's it on the app side. Now, navigate back to the **shared-ui** module and make a few more updates.

The bottom navigation bar on Android uses the **NavHost**, which isn't available for Multiplatform. Fortunately, **precompose** has a similar feature called **Navigator**. You'll need to replace the current implementation that uses `NavController` with this one.

Open the **main/MainBottomBar.kt** file and replace the type of the `NavController` to **Navigator**. You need to make this change on **MainBottomBar** and **AppBottomNavigation** functions.

Once that's done, don't forget to remove the imports:

```
import androidx.navigation.NavHostController
```

Now that you've updated **MainBottomBar**, you'll have to make similar changes on **MainContent.kt**. Open this file, and once again replace the `NavController` type on the different functions with **Navigator**.

In **MainScreenNavigationConfigurations**, you also have to import the **NavHost** from **precompose** and set it as `navigator`, and replace `startDestination` with `initialRoute`:

```
NavHost(
    navigator = navController,
    initialRoute = DEFAULT_SCREEN.route
)
```

Afterward, replace the multiple composable calls with `scene`.

Finally, remove the `androidx.*` imports:

```
import androidx.navigation.NavHostController  
import androidx.navigation.compose.NavHost  
import androidx.navigation.compose.composable
```

The last change required is in **MainScreen.kt** when `navController` is defined:

```
val navController = rememberNavigator()
```

And, remove:

```
navController.enableOnBackPressed(false)
```

Which is currently not supported.

Finally, remove the import:

```
import androidx.navigation.compose.rememberNavController
```

Handling Resources

All platforms handle resources quite differently. Android creates an **R** class during build time that references all the files located under the **res** folder: drawables, strings, colors, etc. Although this gives you easy access to the application resource files, it won't work on another platform.

There are currently two libraries you can use to handle resources:

- **resources** (<https://github.com/JetBrains/compose-multiplatform/tree/master/components/resources>): developed by JetBrains, it's currently in an experimental state and, for now, it doesn't support string sharing.
- **moko-resources** (<https://github.com/icerockdev/moko-resources>): developed by IceRock Development (<https://moko.icerock.dev/>), supports sharing strings, images, and fonts across multiple platforms: JVM, Native, and JS.

In this section you're going to use **moko-resources** since sharing strings is one of the features that you will need to share your UI across all the targets. It's also worth mentioning, that this library has been available for some time now and being used in several projects, which indirectly makes it more stable than **resources** at this time.

Note: By default (<https://github.com/icerockdev/moko-resources/issues/530>), you can't use both libraries at the same time. **resources** currently doesn't run if it detects that your project has the moko plugin added.

Configuring moko-resources

Start by opening **libs.versions.toml** file, located inside the **gradle** folder. Inside the **[versions]** section at the latest moko-resources version:

```
moko-resources = "0.23.0"
```

Scroll down to **[libraries]** group and add both libraries:

```
moko-resources = { module = "dev.icerock.moko:resources",
version.ref = "moko-resources" }
moko-resources-compose = { module = "dev.icerock.moko:resources-
compose", version.ref = "moko-resources" }
```

The second library is to use **moko** with Compose.

Finally, go to the **[plugins]** set and add:

```
moko-multiplatform-resources = { id =
"dev.icerock.mobile.multiplatform-resources", version.ref =
"moko-resources" }
```

Now that both libraries and plugins are defined, it's time to include them in the project. Open the **build.gradle.kts** file located in the root directory. In the **plugins** section, at the end of the list, add:

```
alias(libs.plugins.moko.multiplatform.resources) apply false
```

This will add the **multiplatform-resources** to the project. Now, open the **build.gradle.kts** file, but this time the one from **shared-ui** and add its plugin:

```
alias(libs.plugins.moko.multiplatform.resources)
```

With this, you need to set the app package name for **moko-resources** to use. After the **plugins** declaration, add:

```
multiplatformResources {
    multiplatformResourcesPackage = "com.kodeco.learn.ui"
}
```

Now you need to add the libraries to the `commonMain/dependencies` section:

```
api(libs.moko.resources)
api(libs.moko.resources.compose)
```

There's currently an issue (<https://github.com/icerockdev/moko-resources/issues/510>) copying resources from a module to the app project, so you need to manually set the resources source directory for each one of the platforms that you're targeting. After `getByName("commonTest")` add:

```
getByName("desktopMain") {
    resources.srcDirs("build/generated/moko/desktopMain/src")
}

getByName("iosX64Main") {
    resources.srcDirs("build/generated/moko/iosX64Main/src")
}

getByName("iosArm64Main") {
    resources.srcDirs("build/generated/moko/iosArm64Main/src")
}

getByName("iosSimulatorArm64Main") {
    resources.srcDirs("build/generated/moko/iosSimulatorArm64Main/
src")
}
```

For Android, you have to scroll down to the end of the file, and inside the `android` section add:

```
sourceSets["main"].java.srcDirs("build/generated/moko/
androidMain/src")
```

Click **Sync Now** and wait for the project to load these new libraries.

Loading Local Images

You'll write the logic to load local images in Kotlin Multiplatform. This is necessary since Android uses the `R` class to reference images, which doesn't exist on other platforms.

It's also worth mentioning that all platforms can use different formats for images. Although Android and desktop supports vector drawables, it's currently not available for iOS using **moko-resources**.

Nevertheless, you can use PNGs, JPGs, or SVGs on all platforms. With this in mind, and that SVGs are vector-based images, which means that they can be resized without losing quality, you're going to use this format for sharing images.

Open **shared-ui/commonMain** and start by creating a new resources folder. You can easily create it by right-clicking on this folder and selecting **New ▶ Directory ▶ resources**. Repeat the process, but this time click on **resources** and enter **MR/images**.

This **MR** folder is required by **moko**. All the resources that you're going to share across multiple platforms need to be located in it.

The SVG files that you will use are located in the **assets** folder in this chapter's materials. Copy-paste the six files into **MR/images**, and remove the correspondent **.xml** files from **androidApp/res/drawable**, which won't be needed anymore.

With all the resources set, you'll need to make quite a few updates to replace the current calls to the **R** class with this new implementation.

Since **moko-resources** is going to generate an **MR** class, available for all platforms, similar to **R** with the reference to all the resources on **shared-ui**, before making any update, you need to first build the project. For that, go to **Build ▶ Make Project** and wait for this operation to end.

Starting alphabetically, you'll need to make the following changes in the **commonMain** files:

common/EntryContent

In the **AddEntryContent** Composable, start by changing the import of **painterResource**. Instead of using **androidx.compose** you have to use the function from **moko.resources.compose**:

```
import dev.icerock.moko.resources.compose.painterResource
```

Now, remember the **R** class is Android-specific, so you'll use the **MR** generated by **moko** instead:

```
val resource = painterResource(MR.images.ic_more)
```

And import:

```
import com.kodeco.learn.ui.MR
```

You can now remove the other imports:

```
import com.kodeco.learn.R
import androidx.compose.ui.res.painterResource
```

components/ImagePreview

Both in the AddImagePreview and AddImagePreviewEmpty Composables, replace the call to R.drawable.ic_brand with:

```
val resource = painterResource(MR.images.ic_brand)
```

Add the import to painterResource from moko.compose and the MR class:

```
import com.kodeco.learn.ui.MR
import dev.icerock.moko.resources.compose.painterResource
```

And remove the imports:

```
import androidx.compose.ui.res.painterResource
import com.kodeco.learn.R
```

main/BottomNavigationScreens

Similar as before, start by importing the painterResource function and the MR class:

```
import com.kodeco.learn.ui.MR
import dev.icerock.moko.resources.compose.painterResource
```

And remove the R class and painterResource from androidx.compose:

```
import androidx.compose.ui.res.painterResource
import com.kodeco.learn.R
```

Now, look for the data objects that are created in this class and replace the R.drawable.* with the equivalent reference from MR.images.*:

- Home :

```
painter = painterResource(MR.images.ic_home),
```

- Bookmark:

```
painter = painterResource(MR.images.ic_bookmarks),
```

- Latest:

```
painter = painterResource(MR.images.ic_latest),
```

- Search:

```
painter = painterResource(MR.images.ic_search),
```

There are still a couple of errors here that are related to the app strings. You'll see how to update this logic in detail in the "Sharing Strings" section of this appendix.

search/SearchContent

In the AddSearchField Composable, replace the painterResource call in leadingIcon with:

```
val resource = painterResource(MR.images.ic_search)
```

Import the corresponding classes:

```
import com.kodeco.learn.ui.MR
import dev.icerock.moko.resources.compose.painterResource
```

And remove the unnecessary imports:

```
import androidx.compose.ui.res.painterResource
import com.kodeco.learn.R
```

All done! A couple more sections to go, and you'll have your app's UI completely shared.

Using Custom Fonts

The font that the three apps use is OpenSans. Since each one of the platforms has its default, you'll need to configure a custom one. You'll use, once again, **moko-resources** to load the new font.

Start by creating the **fonts** folder inside **shared-ui/commonMain/resources/MR** and move the files from **androidApp/resources/font** there. To use a font with **moko** it needs to follow a specific naming:

```
<fontFamily>-<fontStyle>
```

So you'll have to rename all the OpenSans fonts to obey this rule:

```
OpenSans-Bold.ttf  
OpenSans-ExtraBold.ttf  
OpenSans-Light.ttf  
OpenSans-Regular.ttf  
OpenSans-SemiBold.ttf
```

To update the generated **MR** file, go to **Build > Make Project**. Once this operation ends, you can go to **shared-ui/build/generated/moko/commonMain/../MR** and search for fonts. Here, you've got the five different types that you've just added to the project.

You can access any of these fonts via:

```
fontFamilyResource(MR.fonts.OpenSans.regular)
```

Or:

```
MR.fonts.OpenSans.regular.asFont()
```

But implementations need to be called from Composable functions. Therefore, you'll have to use these fonts directly from the **Typography** property that's on **Type.kt** file.

Before updating all the **Text** Composable's with these new typographies, you'll need to remove the references to the **R** class from **Type.kt**. Open this file and remove:

```
import androidx.compose.ui.text.font.Font  
import androidx.compose.ui.text.font.FontFamily  
import com.kodeco.learn.android.R  
  
private val OpenSansFontFamily = FontFamily(  
    Font(R.font.opensans_bold, FontWeight.Bold),  
    Font(R.font.opensans_extrabold, FontWeight.ExtraBold),  
    Font(R.font.opensans_light, FontWeight.Light),  
    Font(R.font.opensans_regular, FontWeight.Normal),  
    Font(R.font.opensans_semidbold, FontWeight.SemiBold),  
)
```

Now that there's no more **OpenSansFontFamily**, you must remove this call from all the **fontFamily** properties. Afterward, you need to manually update all the **Text styles**, since it's not possible to reference the **FONTS** that you've created above from **Typography**.

When prompted, import:

```
import com.kodeco.learn.ui.MR
import dev.icerock.moko.resources.compose.fontFamilyResource
```

Starting alphabetically on **commonMain/ui**, navigate to:

- **common/EmptyContent**: On the Text declaration, set the `fontFamily` argument to:

```
fontFamily = fontFamilyResource(MR.fonts.OpenSans.regular)
```

- **common/EntryContent**: On the AddEntryContent Composable, look for four Text usages and add:

```
fontFamily = fontFamilyResource(MR.fonts.OpenSans.regular)
```

- **home/HomeContent**: Scroll down to the end of this file, and on Text add:

```
fontFamily = fontFamilyResource(MR.fonts.OpenSans.regular)
```

- **home/HomeSheetContent**: Search for the two Text calls and add:

```
fontFamily = fontFamilyResource(MR.fonts.OpenSans.regular)
```

- **latest/LatestContent**: Set the `fontFamily` on the Text declarations on AddNewPage and AddNewPageEntry:

```
fontFamily = fontFamilyResource(MR.fonts.OpenSans.regular)
```

- **main/MainBottomBar**: When defining the BottomNavigationItem, on Text add:

```
fontFamily = fontFamilyResource(MR.fonts.OpenSans.regular)
```

- **main/MainTopAppBar**: Update the Text to contain the `fontFamily` argument:

```
fontFamily = fontFamilyResource(MR.fonts.OpenSans.regular)
```

- **search/SearchContent**: Finally, when defining the placeholder set the `fontFamily` in Text:

```
fontFamily = Fonts.BitterFontFamily(),
```

Sharing Strings

Once again, you're going to use the **moko-resources** library to share strings across all platforms.

The strings on the desktop app are currently hardcoded. This is enough for a simple app, but if you keep adding new features that use strings, having them located in a single file is easier to maintain. Moreover, if you want to add support for internationalization, you'll need to have multiple strings files, so the OS can know which one to load.

You'll reuse the Android **strings.xml** file as the shared strings across both platforms.

In order for **moko-resources** to work, the string files need to be in a specific path: **commonMain/resources/MR/base**. Create the **base** directory and move **strings.xml** from **androidApp/res** to this new location.

Note: If your app supports internationalization, you should create a folder inside **MR** with the language country code, then move the corresponding **strings.xml** file to that location.

Build the project. **moko-resources** will generate a couple of Multiplatform files (Android, desktop, iOS, and common) that contain the strings your app will use. You can find them at **shared-ui/build/generated/moko/**

The changes needed for strings is similar to the one that you've done previously for images. You need to go through all the classes and update the references from **R** to **MR** class, and use the **stringResource** function from **moko**.

Starting alphabetically on **commonMain/ui**, navigate to:

- **bookmark/BookmarkContent**: On **BookmarkContent Composable**, update the **stringResource** call to:

```
text = stringResource(MR.strings.empty_screen_bookmarks)
```

Add the imports to:

```
import com.kodeco.learn.ui.MR
import dev.icerock.moko.resources.compose.stringResource
```

And remove the previous ones:

```
import androidx.compose.ui.res.stringResource
import com.kodeko.learn.android.R
```

- **common/EntryContent.kt**: Locate all the calls to R class, and, orderly, update them to use the equivalent MR reference. Starting with R.string.app_kodeko. Update to:

```
text = stringResource(MR.strings.app_kodeko),
```

And import:

```
import com.kodeko.learn.ui.MR
import dev.icerock.moko.resources.compose.stringResource
```

Finally, change the access to description_more to:

```
val description = stringResource(MR.strings.description_more)
```

And remove the import:

```
import androidx.compose.ui.res.stringResource
```

- **components/ImagePreview.kt**: You only need to make one change. Scroll down to AddImagePreviewEmpty and update the description property that accesses the R class to:

```
val description =
    stringResource(MR.strings.description_preview_error)
```

Import stringResource from moko:

```
import dev.icerock.moko.resources.compose.stringResource
```

And remove the now-unused import:

```
import androidx.compose.ui.res.stringResource
```

- **home/HomeSheetContent.kt**: Look for the accesses to the R class. The first one is the result of an if condition used to decide which text should be displayed. Replace this code block with:

```
val text = if (item.value.bookmarked) {
    stringResource(MR.strings.action_remove_bookmarks)
```

```
    } else {
        stringResource(MR.strings.action_add_bookmarks)
    }
```

And, as usual, import:

```
import com.kodeco.learn.ui.MR
import dev.icerock.moko.resources.compose.stringResource
```

At the end of the file, there's another reference to **R**. Replace this call with:

```
text = stringResource(MR.strings.action_share_link),
```

And remove the imports of:

```
import androidx.compose.ui.res.stringResource
import com.kodeco.learn.android.R
```

- **latest/LatestContent.kt**: On LatestContent Composable, update the strings call to:

```
AddEmptyScreen(stringResource(MR.strings.empty_screen_loading))
```

Add the imports:

```
import com.kodeco.learn.ui.MR
import dev.icerock.moko.resources.compose.stringResource
```

And, as always, remove the unnecessary ones:

```
import androidx.compose.ui.res.stringResource
import com.kodeco.learn.android.R
```

- **main/BottomNavigationScreens.kt**: `@StringRes` is a string reference specific to the Android platform. Since you're sharing this class with a desktop, and an iOS app, you need to update this parameter to a common type — which will be **StringResource**. Change `stringResId` to:

```
val title: StringResource,
```

With that, you need to update all the objects declared in this class.

For the home object, update the `stringResId` and the `contentDescription`, respectively, to:

```
title = MR.strings.navigation_home,  
  
contentDescription = stringResource(MR.strings.navigation_home)
```

And add the corresponding imports:

```
import com.kodeko.learn.ui.MR  
import dev.icerock.moko.resources.StringResource  
import dev.icerock.moko.resources.compose.stringResource
```

The same applies to the bookmark object:

```
title = MR.strings.navigation_bookmark,  
  
contentDescription =  
    stringResource(MR.strings.navigation_bookmark)
```

And to latest:

```
title = MR.strings.navigation_latest,  
  
contentDescription =  
    stringResource(MR.strings.navigation_latest)
```

Finally, for search:

```
title = MR.strings.navigation_search,  
  
contentDescription =  
    stringResource(MR.strings.navigation_search)
```

Remove the now-unnecessary imports:

```
import androidx.annotation.StringRes  
import androidx.compose.ui.res.stringResource
```

- **main/MainBottomBar.kt**: With the previous change, you have to update the `BottomNavigationItem` in the `MainBottomBar`. Replace the `stringResource` from `androidx.compose` to:

```
import dev.icerock.moko.resources.compose.stringResource
```

And remove its import:

```
import androidx.compose.ui.res.stringResource
```

- **main/MainTopAppBar.kt**: Replace the `stringResource` call with:

```
text = stringResource(MR.strings.app_name),
```

And import:

```
import com.kodeco.learn.ui.MR
import dev.icerock.moko.resources.compose.stringResource
```

Now scroll down to where the Icon contentDescription is set, and update it to:

```
contentDescription =
stringResource(MR.strings.description_profile)
```

Finally, remove the imports:

```
import androidx.compose.ui.res.stringResource
import com.kodeco.learn.android.R
```

- **search/SearchContent.kt**: This is the last file that needs to be updated! Scroll down to `AddSearchField` and locate the two calls to `stringResource`. The first one is where you're defining the placeholder and needs to be updated to:

```
text = stringResource(MR.strings.search_hint),
```

The second one is for `leadingIcon`, and you have to change the description to:

```
val description = stringResource(MR.strings.description_search)
```

Don't forget to add the imports:

```
import com.kodeco.learn.ui.MR
import dev.icerock.moko.resources.compose.stringResource
```

And, as always, remove the ones you're no longer using:

```
import androidx.compose.ui.res.stringResource
```

What's Missing?

With all of these changes done, you're almost done. Open the **desktopApp** project and:

- Remove the **ui**, **components**, and **utils** folders.
- From **resources**, delete the **font** and **images** directory. You should only have here your app icons.

The entry point of your desktop is the **Main.kt** file.

Now, open its **build.gradle.kts** and include the **shared-ui** dependency you've created throughout this appendix. To avoid having unnecessary implementations, you can replace all the libraries in this section with:

```
implementation(project(":shared-ui"))
implementation(project(":shared-action"))

implementation(compose.desktop.currentOs)
```

Do the same for **androidApp**. Open its **build.gradle.kts** and replace the dependencies section with:

```
implementation(project(":shared-ui"))
implementation(project(":shared-action"))

implementation(libs.android.material)
```

The strings' namespace changed to **com.kodeco.learn.ui**, you'll need to make this update in **MainActivity.kt**. Open this file and replace, the existing import:

```
import com.kodeco.learn.R
```

With the new one:

```
import com.kodeco.learn.ui.R
```

There's one more change that you need to do. Open **Theme.kt** in **commonMain/./ui/theme**. If you look at the **KodecoTheme**, you can see that there's a set of operations that are going to update the status and navigation bars which are Android-specific.

Remove the following code block:

```
val view = LocalView.current
if (!view.isInEditMode) {
    SideEffect {
        val window = (view.context as Activity).window
        window.statusBarColor = colorScheme.surface.toArgb()
        WindowCompat.getInsetsController(window,
            view).isAppearanceLightStatusBars = !darkTheme
    }
}
```

And then also remove its imports:

```
import android.app.Activity
import androidx.compose.runtime.SideEffect
import androidx.compose.ui.platform.LocalView
import androidx.core.view.WindowCompat
```

Now return to **MainActivity.kt** in **androidApp** and update the **KodecoTheme** call to:

```
val darkTheme = isSystemInDarkTheme()
KodecoTheme(
    darkTheme = darkTheme
) {

    val view = LocalView.current
    val colorScheme = MaterialTheme.colorScheme
    if (!view.isInEditMode) {
        SideEffect {
            val window = (view.context as Activity).window
            window.statusBarColor = colorScheme.surface.toArgb()
            WindowCompat.getInsetsController(window,
                view).isAppearanceLightStatusBars = !darkTheme
        }
    }
}
```

Add the needed imports. You can refer to the imports you removed in the previous step.

Synchronize your project and — finally — compile and run your desktop and Android apps.

You'll see screens like these:

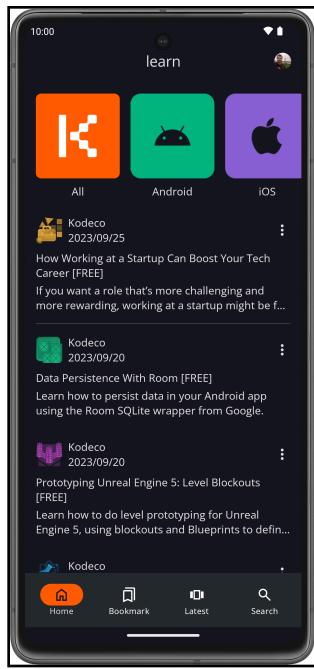


Fig. B.6 – Feed in Android App



Fig. B.7 – Feed in Desktop App

Now, for iOS, you'll need to make additional changes. Start by opening **build.gradle.kts** file from **shared-ui** and in the **kotlin** section, update the framework declaration to:

```
it.binaries.framework {
    baseName = "SharedUIKit"
    linkerOpts.add("-lsqlite3")
}
```

This way the framework name follows Apple guidelines, and additionally you need to set this flag, which is required by SQLDelight.

With the configuration done, go to **shared-ui/iosMain/..ui** and create a **Main.ios.kt** file, and add the following code:

```
package com.kodeco.learn.ui

import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.material.Surface
import androidx.compose.ui.Modifier
import com.kodeco.learn.data.model.KodecoEntry
import com.kodeco.learn.ui.bookmark.BookmarkViewModel
import com.kodeco.learn.ui.home.FeedViewModel
import com.kodeco.learn.ui.main.MainScreen
import com.kodeco.learn.ui.theme.KodecoTheme
import moe.tlaster.precompose.PreComposeApplication
import moe.tlaster.precompose.viewmodel.viewModel
import platform.Foundation.NSLog
import platform.Foundation.URL
import platform.UIKit.UIApplication

private lateinit var bookmarkViewModel: BookmarkViewModel
private lateinit var feedViewModel: FeedViewModel

private lateinit var bookmarkViewModel: BookmarkViewModel
private lateinit var feedViewModel: FeedViewModel

fun MainViewController() = PreComposeApplication {
    Surface(modifier = Modifier.fillMaxSize()) {

        bookmarkViewModel = viewModel(BookmarkViewModel::class) {
            BookmarkViewModel()
        }

        feedViewModel = viewModel(FeedViewModel::class) {
            FeedViewModel()
        }

        feedViewModel.fetchAllFeeds()
        feedViewModel.fetchMyGravatar()
        bookmarkViewModel.getBookmarks()
    }
}
```

```
    val items = feedViewModel.items
    val profile = feedViewModel.profile
    val bookmarks = bookmarkViewModel.items

    KodecoTheme {
        MainScreen(
            profile = profile.value,
            feeds = items,
            bookmarks = bookmarks,
            onUpdateBookmark = { updateBookmark(it) },
            onShareAsLink = {},
            onOpenEntry = { openLink(it) }
        )
    }
}

private fun updateBookmark(item: KodecoEntry) {
    if (item.bookmarked) {
        removeFromBookmarks(item)
    } else {
        addToBookmarks(item)
    }
}

private fun addToBookmarks(item: KodecoEntry) {
    bookmarkViewModel.addAsBookmark(item)
    bookmarkViewModel.getBookmarks()
}

private fun removeFromBookmarks(item: KodecoEntry) {
    bookmarkViewModel.removeFromBookmark(item)
    bookmarkViewModel.getBookmarks()
}

private fun openLink(url: String) {
    val application = UIApplication.sharedApplication
    val nsurl = NSURL(string = url)
    if (!application.canOpenURL(nsurl)) {
        NSLog("Unable to open url: $url")
        return
    }

    application.openURL(nsurl)
}
```

If you look at `MainActivity.kt` or `Main.kt` from the `desktopApp`, you can see that the code is identical.

Now open Xcode, and go to **iOSApp** ▶ **Build Phases** ▶ **Compile Kotlin** and update the existing script to compile a framework from `shared-ui` instead:

```
cd "$SRCROOT/.."  
./gradlew :shared-ui:embedAndSignAppleFrameworkForXcode
```

You also need to update the path location where Xcode is going to look for the framework for the project. Now go to the **Build Settings** section and scroll down to **Linking - General** and look for **Other Linker Flags** setting. Or, you can just search for it. Now double-click on its value, and update the current SharedKit to SharedUIKit.

Once done, search for **Framework Search Paths**, which is inside the **Search Paths** section, and once again, double-click on its value: <Multiple values>. Scroll horizontally on the path to the **shared** framework, and update it to be **shared-ui**.

Finally, open **ContentView.swift** and add the SharedUIKit import to the list:

```
import SharedUIKit
```

And `makeUIViewController` instead of loading an empty Controller should now import the one that you've created in `Main.ios.kt`:

```
func makeUIViewController(context: Context) -> UIViewController  
{  
    Main_iosKt.MainViewController()  
}
```

One last change, if you scroll down to the end of this file, you see there's an `.ignoresSafeArea` invocation. Update it to:

```
.ignoresSafeArea(.all, edges: .all)
```

Otherwise, the status and navigation bars won't have the same color as the background.

Now compile and run your iOS app!

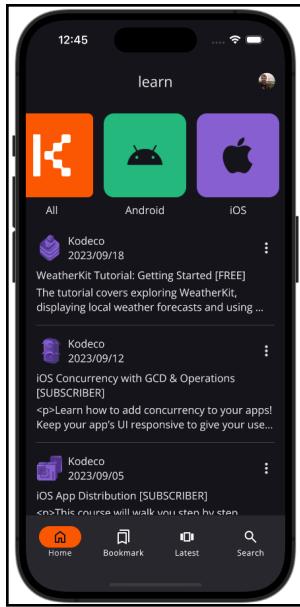


Fig. B.8 – Feed in iOS App (dark mode)

Want to see something amazing? It also supports light mode. :]

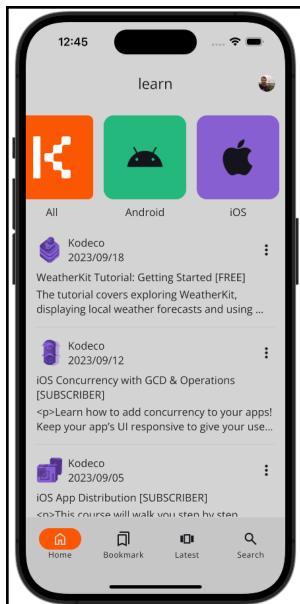


Fig. B.9 – Feed in iOS App (light mode)

Where to Go From Here?

Congratulations! You just finished Kotlin Multiplatform by Tutorials. What a ride! Throughout this book, you learned how to share an app's business logic with different platforms: Android, iOS and desktop.

Now that you've mastered KMP, perhaps you're interested in learning more about Jetpack Compose (<https://www.kodeco.com/books/jetpack-compose-by-tutorials>) and SwiftUI (<https://www.kodeco.com/books/swiftui-by-tutorials>). These books are the perfect starting point!