

SwiftUI Cookbook

Second Edition

A guide to solving the most common problems and learning best practices while building SwiftUI apps

Giordano Scalzo | Edgar Nzokwe



SwiftUI Cookbook

Second Edition

A guide to solving the most common problems and learning best practices while building SwiftUI apps

Giordano Scalzo

Edgar Nzokwe

Packt

BIRMINGHAM—MUMBAI

SwiftUI Cookbook

Second Edition

Copyright © 2021 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

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

Group Product Manager: Rohit Rajkumar

Publishing Product Manager: Ashitosh Gupta

Senior Editor: Hayden Edwards

Content Development Editor: Rashi Dubey

Technical Editor: Joseph Aloocaran

Copy Editor: Safis Editing

Project Coordinator: Rashika Ba

Proofreader: Safis Editing

Indexer: Manju Arasan

Production Designer: Vijay Kamble

First published: October 2020

Second edition: October 2021

Production reference: 1291021

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-80323-445-8

www.packtpub.com

We wrote this book standing on the shoulders of giants.

Thanks to Dave Verwer, Majid Jabrayilov, John Sundell, Gui Rambo, Gio Lodi, Antoine van der Lee, Donny Wals, Vincent Pradelles, Mark Moeykens, Paul Hudson, and all the fantastic Swift community for the support and knowledge.

- Giordano Scalzo

To my childhood mentor, Mr. Tatuh N. Atem Formin, who introduced me to the amazing world of computing.

- Edgar Nzokwe

Contributors

About the authors

Giordano Scalzo started his journey back in the days of ZX Spectrum. He has worked with Swift, Objective-C, C/C++, Java, .NET, Ruby, Python, and a ton of other languages that he has forgotten the names of. He is currently a tech lead consultant in London, where he leads mobile digital transformations through his company, Effective Code Ltd.

To my bright future, Mattia and Luca, for giving me lots of smiles and hugs when I needed them.

To my better half, Valentina, who lovingly supports me in everything I do: without you, none of this would have been possible.

Edgar Nzokwe is a software engineer with 6+ years of experience building web and mobile applications. His areas of expertise include SwiftUI, UIKit, Kotlin, Python, and a few JavaScript frameworks. Edgar is dedicated to advancing the knowledge base of SwiftUI because he believes it empowers developers to design and build dazzling cross-platform apps with minimal development time.

I want to thank my wife, Mabelle, for encouraging me along this journey.

Thanks to my kids, Neil and Zoe, for making me smile when the going got tough.

Finally, I want to give special thanks to my editors, Hayden Edwards and Rashi Dubey, as well as the rest of the Packt team who helped put this book together.

About the reviewers

Chris Barker is an iOS developer and principal software manager for Jaguar Land Rover, where he heads up the mobile app team.

Chris started his career developing .NET applications for online retailer dabs.com (now BT Shop) before he made his move into mobile app development with digital agency Openshadow (now MyStudioFactory Paris). There, he worked on mobile apps for clients such as Louis Vuitton and L'Oréal Paris.

He has since worked for online fashion retailer N Brown (JD Williams, SimplyBe, Jacamo) as an iOS tech lead.

Chris often attends and speaks at local iOS developer meetups and conferences such as NSManchester, Malaga Mobile, and CodeMobile.

Danny Bolella is an iOS developer lead at a major trading company. With over a decade of experience under his belt, he's worked on everything from full-stack web to mobile apps in a variety of industries, including financial, energy, and medical devices. He also enjoys writing articles and was the technical reviewer for *SwiftUI Cookbook, First Edition*.

Danny thanks his amazing wife and children, who give him the motivation and drive to always be better.

Marc Aupont is a first-generation American born to Haitian immigrant parents. After 12 years of working in fintech, he transitioned into mobile software development when Swift was announced in 2015. His passion for technology led him to move from Orlando, FL to NYC in 2017, where he started multiple meetups teaching programming. In addition to his work as a senior iOS engineer at Nike, Marc has been a champion for inclusion and diversity within developer communities for many years. Marc has worked to create opportunities for technical growth in the Swift community by organizing meetups for beginners, sharing his own experience and expertise through mentorship programs, working collaboratively with HBCU students on app development, and more. His hobbies include working on side projects involving electronics and hardware, serving as a music director at his local church, and going on weekend road trips to random destinations with his family.

Table of Contents

Preface

1

Using the Basic SwiftUI Views and Controls

Technical requirements	2	Beyond buttons – using advanced pickers	23
Laying out components	3	Getting ready	23
Getting ready	3	How to do it...	23
How to do it...	3	How it works...	26
How it works...	6	There's more...	27
There's more...	7	Applying groups of styles using ViewModifier	27
Dealing with text	7	Getting ready	27
Getting ready	7	How to do it...	27
How to do it...	7	How it works...	29
How it works...	10	There's more...	30
There's more...	11	See also	30
See also	11	Separating presentation from content with ViewBuilder	30
Using images	11	Getting ready	30
Getting ready	11	How to do it...	30
How to do it...	12	How it works...	32
How it works...	15	See also	33
See also	18	Adding buttons and navigating with them	33
Getting ready	18	Simple graphics using SF Symbols	33
How to do it...	18	Getting ready	33
How it works...	22	How to do it...	33
See also	23	How it works...	35
		See also	36

Integrating UIKit into SwiftUI – the best of both worlds	36	Getting ready	39
Getting ready	37	How to do it...	40
How to do it...	37	How it works...	43
How it works...	39		
See also	39	Exploring more views and controls (iOS 14+)	43
		Getting ready	43
Adding SwiftUI to an existing app	39	How to do it...	43
		How it works...	47

2

Going Beyond the Single Component with Lists and Scroll Views

Technical requirements	50	How it works...	66
Using scroll views	50	There's more...	66
Getting ready	50	Creating an editable List view	67
How to do it...	51	Getting ready	67
How it works...	52	How to do it...	67
See also	53	How it works...	68
		There's more...	68
Creating a list of static items	53		
Getting ready	53	Moving the rows in a List view	69
How to do it...	53	Getting ready	69
How it works...	56	How to do it...	69
		How it works...	71
Using custom rows in a list	57		
Getting ready	57	Adding sections to a list	71
How to do it...	57	Getting ready	71
How it works...	61	How to do it...	71
		How it works...	73
Adding rows to a list	62		
Getting ready	62	Creating editable Collections	74
How to do it...	62	Getting ready	74
How it works...	64	How to do it...	74
		How it works...	75
Deleting rows from a list	64		
Getting ready	64	Using Searchable lists	76
How to do it...	65	Getting ready	76

How to do it...	77	There's more...	80
How it works...	79		

3

Exploring Advanced Components

Technical requirements	82	Displaying hierarchical content in expanding lists	95
Using LazyHStack and LazyVStack	82	Getting ready	95
Getting ready	82	How to do it...	95
How to do it...	82	How it works...	98
How it works...	86	There's more...	99
There's more...	86		
Displaying tabular content with LazyHGrid and LazyVGrid	86	Using disclosure groups to hide and show content	99
Getting ready	86	Getting ready	99
How to do it...	87	How to do it...	100
How it works...	89	How it works...	102
Creating SwiftUI widgets	103		
Scrolling programmatically with ScrollViewReader	90	Getting ready	103
Getting ready	90	How to do it...	103
How to do it...	91	How it works...	111
How it works...	95	See also	112

4

Viewing while Building with SwiftUI Preview

Technical requirements	114	How to do it	118
Previewing a layout in dark mode	114	How it works	124
Getting ready	114	See also	124
Previewing a layout in a NavigationView	124		
Getting ready	114		
How to do it	114		
How it works	116	Getting ready	124
Previewing a layout at different dynamic type sizes	117	How to do it	125
Getting ready	117	How it works	128

Previewing a layout on different devices	128	How to do it	131
		How it works	135
Getting ready	128	See also	135
How to do it	128		
How it works	130	Using mock data for previews	135
		Getting ready	136
Using previews in UIKit	131	How to do it	136
Getting ready	131	How it works	140
		There's more	140

5

Creating New Components and Grouping Views with Container Views

Technical requirements	142	How to do it...	151
Showing and hiding sections in forms	142	How it works...	154
Getting ready	142	There's more...	154
How to do it...	142	See also	155
How it works...	146		
There's more...	146	Navigating between multiple views with TabView	155
See also	147	Getting ready	155
Disabling and enabling items in forms	147	How to do it...	155
Getting ready	148	How it works...	160
How to do it...	148	There's more...	161
How it works...	150	See also	161
Filling out forms easily using Focus and Submit	151	Using gestures with TabView	161
Getting ready	151	Getting ready	162
		How to do it...	162
		How it works...	163

6

Presenting Extra Information to the User

Technical requirements	166	Getting ready	166
Presenting alerts	166	How to do it	167
		How it works	169

See also	170	Getting ready	176
Adding actions to alert buttons	170	How to do it	176
Getting ready	170	How it works	181
How to do it	171	See also	181
How it works	173	Creating context menus	182
Presenting confirmation dialogs	173	Getting ready	182
Getting ready	173	How to do it	182
How to do it	173	How it works	184
How it works	176	See also	184
Presenting new views using sheets	176	Displaying popovers	184
See also	176	Getting ready	184
		How to do it	184
		How it works	187
		See also	188

7

Drawing with SwiftUI

Technical requirements	190	How it works...	203
Using SwiftUI's built-in shapes	190	There's more...	203
Getting ready	190	Implementing a progress ring	203
How to do it...	191	Getting ready	203
How it works...	192	How to do it...	204
Drawing a custom shape	193	How it works...	207
Getting ready	193	Implementing a Tic-Tac-Toe game in SwiftUI	208
How to do it...	194	Getting ready	208
How it works...	195	How to do it...	208
Drawing a curved custom shape	196	How it works...	213
Getting ready	197	There's more...	213
How to do it...	197	Rendering a gradient view in SwiftUI	214
How it works...	199	Getting ready	214
Drawing using the Canvas API	200	How to do it...	214
Getting ready	200	How it works...	218
How to do it...	201		

Building a bar chart	219	Building a pie chart	224
Getting ready	219	Getting ready	224
How to do it...	219	How to do it...	224
How it works...	223	How it works	232
There's more...	223		

8

Animating with SwiftUI

Technical requirements	236	How to do it...	253
Creating basic animations	236	How it works...	256
Getting ready	237	Applying multiple animations to a view	256
How to do it...	237	Getting ready	256
How it works...	241	How to do it...	256
There's more...	241	How it works...	258
See also	242		
Transforming shapes	242	Creating custom view transitions	259
Getting ready	242	Getting ready	259
How to do it...	242	How to do it...	260
How it works...	244	How it works...	263
Creating a banner with a spring animation	246	Creating a hero view transition with .matchedGeometryEffect	263
Getting ready	246	Getting ready	263
How to do it...	246	How to do it...	264
How it works...	248	How it works...	270
Applying a delay to a view modifier animation to create a sequence of animations	249	Lottie animations in SwiftUI	271
Getting ready	249	Getting ready	271
How to do it...	249	How to do it...	274
How it works...	252	How it works...	277
Applying a delay to a withAnimation function to create a sequence of animations	252	Implementing a stretchable header in SwiftUI	277
Getting ready	252	Getting ready	278
		How to do it...	278
		How it works...	281

Creating floating hearts in SwiftUI	282	Implementing a swipeable stack of cards in SwiftUI	291
Getting ready	282	Getting ready	292
How to do it...	283	How to do it...	292
How it works...	289	How it works...	298
See also	291		

9

Driving SwiftUI with Data

Technical requirements	301	How to do it...	312
Using @State to drive a View's behavior	301	How it works...	316
Getting ready	301	Using @StateObject to preserve the model's life cycle	317
How to do it...	301	Getting ready	317
How it works...	304	How to do it...	317
See also	305	How it works...	320
Using @Binding to pass a state variable to child Views	305	Sharing state objects with multiple Views using @EnvironmentObject	321
Getting ready	305	Getting ready	322
How to do it...	306	How to do it...	323
How it works...	310	How it works...	329
Implementing a CoreLocation wrapper as @ObservedObject	310	See also	330
Getting ready	311		

10

Driving SwiftUI with Combine

Technical requirements	333	Getting ready	346
Introducing Combine in a SwiftUI project	333	How to do it...	346
Getting ready	334	How it works...	350
How to do it...	335	See also	351
How it works...	342	Validating a form using Combine	352
See also	345	Getting ready	352
Managing the memory in Combine to build a timer app	345	How to do it...	352
		How it works...	362

There's more...	362	on Combine	376
Fetching remote data using Combine and visualizing it in SwiftUI	363	Getting ready	376
Getting ready	363	How to do it...	377
How it works...	374	How it works...	381
There's more...	376	There's more...	381
Debugging an app based		Unit testing an app based on Combine	382
		Getting ready	382
		How to do it...	383
		How it works...	390

11

SwiftUI Concurrency with `async await`

Technical requirements	392	How to do it...	403
Integrating SwiftUI and an <code>async</code> function	392	How it works...	407
Getting ready	393	Converting a completion block function to <code>async await</code>	407
How to do it...	393	Getting ready	407
How it works...	395	How to do it...	408
Fetching remote data in SwiftUI	397	How it works...	411
Getting ready	397	See also	413
How to do it...	397	Implementing infinite scrolling with <code>async await</code>	413
How it works...	401	Getting ready	413
Pulling and refreshing data asynchronously in SwiftUI	402	How to do it...	418
Getting ready	402	How it works...	422
		See also	422

12

Handling Authentication and Firebase with SwiftUI

Technical requirements	425	Getting ready	426
Implementing Sign in with Apple in a SwiftUI app	425	How to do it...	426
		How it works...	432

Integrating Firebase into a SwiftUI project	433	How to do it...	448
Getting ready	434	How it works...	459
How to do it...	435		
How it works...	446		
There's more...	446		
Using Firebase to sign in using Google	446		
Getting ready	447		
Implementing a distributed Notes app with Firebase and SwiftUI	461		
Getting ready	462		
How to do it...	463		
How it works...	474		
There's more...	476		

13

Handling Core Data in SwiftUI

Technical requirements	478	Filtering Core Data requests using a predicate	499
Integrating Core Data with SwiftUI	478	Getting ready	500
Getting ready	479	How to do it...	500
How to do it...	479	How it works...	503
How it works...	484		
There's more...	484	Deleting Core Data objects from a SwiftUI view	504
Showing Core Data objects with @FetchRequest	484	Getting ready	504
Getting ready	485	How to do it...	504
How to do it...	485	How it works...	506
How it works...	492		
Adding Core Data objects from a SwiftUI view	492	Presenting data in a sectioned list with @SectionedFetchRequest	507
Getting ready	492	Getting ready	507
How to do it...	492	How to do it...	507
How it works...	498	How it works...	511

14

Creating Cross-Platform Apps with SwiftUI

Technical requirements	514	How to do it...	525
Creating an iOS app in SwiftUI	514	How it works...	534
Getting ready	514	Creating the watchOS version of the iOS app	534
How to do it...	515	Getting ready	534
How it works...	524	How to do it...	535
Creating the macOS version of the iOS app	524	How it works...	542
Getting ready	525		

15

SwiftUI Tips and Tricks

Technical requirements	546	There's more...	567
Snapshot testing SwiftUI views	546	Implementing SwiftUI views using Playground	567
Getting ready	547	Getting ready	567
How to do it...	549	How to do it...	567
How it works...	554	How it works...	569
Unit testing SwiftUI with ViewInspector	556	Using custom fonts in SwiftUI	569
Getting ready	556	Getting ready	570
How to do it...	558	How to do it...	571
How it works...	563	How it works...	574
See also	563		
Showing a PDF in SwiftUI	564	Implementing a Markdown editor with Preview	574
Getting ready	564	Getting ready	574
How to do it...	564	How to do it...	574
How it works...	566	How it works...	578

Other Books You May Enjoy

Index

Preface

SwiftUI provides an innovative and simple way to build user interfaces across all Apple platforms. SwiftUI code is easy to read and write because it uses Swift's declarative programming syntax.

This book covers the foundations of SwiftUI, as well as the new features of SwiftUI 3.0 introduced in iOS 15. Particular attention is given to illustrate the interaction between SwiftUI and the rest of an app's code, such as Combine, Core Data, Firebase, and network services.

By the end of this book, you'll have simple, direct solutions to common problems found in building SwiftUI apps, and you'll know how to build visually compelling apps using SwiftUI, Combine, and async await code.

Who this book is for

This book is for mobile developers who want to learn SwiftUI as well as experienced iOS developers transitioning from UIKit to SwiftUI. The book assumes knowledge of the Swift programming language. Knowledge of object-oriented design and data structures will be useful but is not necessary. You'll also find this book to be a helpful resource if you're looking for reference material regarding the implementation of various features in SwiftUI.

What this book covers

Chapter 1, Using the Basic SwiftUI Views and Controls, explains how to implement various SwiftUI layout components, such as HStack, VStack, ZStack, LazyHStack, LazyVStack, LazyHGrid, LazyVGrid, and other layout components. It also covers how to capture various user gestures and how to use form fields. This chapter covers the basic building blocks for creating SwiftUI apps.

Chapter 2, Going Beyond the Single Component with Lists and Scroll Views, explains how to implement Lists, ScrollViews, and editable and searchable lists.

Chapter 3, Exploring Advanced Components, explains how to use Lazy Stacks and Lazy Grids to improve performance when displaying large datasets. Also, you'll learn how to programmatically scroll with ScrollViewReader, display hierarchical data in an expanding list, and hide and show content with DisclosureGroups. Finally, you'll learn how to display glanceable content using SwiftUI Widgets.

Chapter 4, Viewing While Building with SwiftUI Preview, explains how to unleash the power and capabilities of SwiftUI previews to speed up UI development time.

Chapter 5, Creating New Components and Grouping Views in Container Views, explains how to group views, use container views, and implement architectural views such as NavigationView and TabView.

Chapter 6, Presenting Extra Information to the User, provides various ways of presenting extra information to the user, such as alerts, modals, context menus, and popovers.

Chapter 7, Drawing with SwiftUI, explains how to implement drawings in SwiftUI by using built-in shapes and drawing custom paths and polygons.

Chapter 8, Animating with SwiftUI, explains how to implement basic animations, spring animations, and implicit and delayed animations, as well as how to combine transitions, create custom transitions, and create asymmetric transitions.

Chapter 9, Driving SwiftUI with Data, explains how to use the SwiftUI binding mechanism to populate and change views when the bounded data changes.

Chapter 10, Driving SwiftUI with Combine, explains how to integrate Combine to drive the changes of the SwiftUI views. You'll explore how to validate forms, fetch data asynchronously from the network, and test Combine-based apps.

Chapter 11, SwiftUI Concurrency with async await, explains how to implement Swift concurrency with async await, fetching from a network resource, and creating an infinitely scrolling page.

Chapter 12, Handling Authentication and Firebase with SwiftUI, explains how to implement authentication in your app and store data in cloud network storage.

Chapter 13, Handling Core Data in SwiftUI, explains how to implement persistence using SwiftUI and Core Data, saving, deleting, and modifying objects in a Core Data local database.

Chapter 14, Creating Cross-Platform Apps with SwiftUI, explains how to create a cross-platform SwiftUI app that works on iOS, macOS, and watchOS.

Chapter 15, SwiftUI Tips and Tricks, covers several SwiftUI tips and tricks that will help you solve a number of common problems, such as testing the views, using custom fonts, and using Markdown text.

To get the most out of this book

You will need Xcode 13 and iOS 15 installed on your Mac, if possible. All code examples have been tested using iOS 15. However, they should work with future version releases, too:

Software/hardware covered in the book	Operating system requirements
Xcode 13	macOS

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

Download the example code files

You can download the example code files for this book from GitHub at <https://github.com/PacktPublishing/SwiftUI-Cookbook-2nd-Edition>. If there's an update to the code, it will be updated in the GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots and diagrams used in this book. You can download it here: https://static.packt-cdn.com/downloads/9781803234458_ColorImages.pdf.

Conventions used

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

Code in text: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Let's start by creating a new SwiftUI app called `StaticList`."

A block of code is set as follows:

```
struct Todo: Identifiable {  
    let id = UUID()  
    let description: String  
    var done: Bool  
}
```

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

```
struct ContentView: View {  
    @State private var tabSelected = 0  
    ...  
}
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For instance, words in menus or dialog boxes appear in **bold**. Here is an example: "Tap on the + button a few times and then on the **Refresh** button."

Tips or important notes

Appear like this.

Sections

In this book, you will find several headings that appear frequently (*Getting ready*, *How to do it...*, *How it works...*, *There's more...*, and *See also*).

To give clear instructions on how to complete a recipe, use these sections as follows:

Getting ready

This section tells you what to expect in the recipe and describes how to set up any software or any preliminary settings required for the recipe.

How to do it...

This section contains the steps required to follow the recipe.

How it works...

This section usually consists of a detailed explanation of what happened in the previous section.

There's more...

This section consists of additional information about the recipe in order to make you more knowledgeable about the recipe.

See also

This section provides helpful links to other useful information for the recipe.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, email us at customercare@packtpub.com and mention the book title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/support/errata and fill in the form.

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

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

1

Using the Basic SwiftUI Views and Controls

SwiftUI was launched during Apple's **Worldwide Developer Conference (WWDC)** in June 2019. Since then, its popularity has kept increasing as it has been adopted more into the Apple developer community. Apple also releases updates every year, adding new and exciting capabilities to SwiftUI.

SwiftUI ditches UIKit concepts such as Auto Layout for an easier-to-use declarative programming model. It allows the fast and easy creation of applications that work across Apple platforms (iOS, iPadOS, and macOS).

Are you still wondering whether you should start learning SwiftUI? Here are a few reasons to convince you:

- **SwiftUI apps can work alongside UIKit apps:** You can slowly convert your app's **user interface (UI)** to SwiftUI, one screen at a time.
- **Industry adoption:** SwiftUI adoption among most companies will take time. Getting a head start and learning SwiftUI now improves your future marketability. SwiftUI today is like the early days of Objective-C versus Swift, where most companies still used Objective-C; now, almost everyone has switched to Swift.

- **Low learning curve:** SwiftUI offers a low learning curve for people who have used declarative programming before. It is also a great way to start learning declarative programming for those with little-to-no experience.
- **Live previews increase speed:** SwiftUI live previews provide an instant preview of your UI. You can quickly prototype apps and make any changes required by clients. This greatly improves the speed of UI development.

This book is designed to be your SwiftUI reference material. Each project focuses on a single concept so that you can understand each concept thoroughly, and then combine multiple concepts to build amazing applications.

In this chapter, we will learn about views and controls, SwiftUI's visual building blocks for app interfaces. The following recipes will be covered:

- Laying out components
- Dealing with text
- Using images
- Adding buttons and navigating with them
- Beyond buttons – using advanced pickers
- Applying groups of styles using `ViewModifier`
- Separating presentation from content with `ViewBuilder`
- Simple graphics using **San Francisco Symbols (SF Symbols)**
- Integrating UIKit into SwiftUI—the best of both worlds
- Adding SwiftUI to an existing app
- Exploring more views and controls (iOS 14+)

Technical requirements

The code in this chapter is based on Xcode 13.0. You can download and install Xcode from the App Store.

All the code examples for this chapter can be found on GitHub at <https://github.com/PacktPublishing/SwiftUI-Cookbook-2nd-Edition/tree/main/Chapter01-Using-the-basic-SwiftUI-Views-and-Controls>.

Laying out components

SwiftUI uses three basic layout components, `VStack`, `HStack`, and `ZStack`. Use the `VStack` view to arrange components in a vertical axis, `HStack` to arrange components in a horizontal axis, and—you guessed it right—use the `ZStack` to arrange components along the vertical and horizontal axis.

In this recipe, we will also look at spaces and adjust the position used to position elements. We will also look at how `Spacer` and `Divider` can be used for layout.

Getting ready

Let's start by creating a new SwiftUI project called `TheStacks`. Use the following steps:

1. Start Xcode (**Finder -> Applications -> Xcode**).
2. Click on **Create a new Xcode project** from the left pane.
Select **iOS**, and then **App**. Click **Next**.
3. In the **Interface** menu, make sure **SwiftUI** is selected instead of storyboard.
4. Enter the **Product Name**, `TheStacks`.
5. Select the folder location to store the project and click **Next**.

How to do it...

Let's implement the `VStack`, `HStack`, and `ZStack` within a single screen to better understand how each works and the differences between them. The steps are given here:

1. Select the `ContentView/ContentView.swift` file on the navigation pane (left side of Xcode).
2. Replace the single text in the body view with a `VStack` and some text:

```
 VStack{  
     Text("VStack Item 1")  
     Text("VStack Item 2")  
     Text("VStack Item 3")  
 } .background(Color.blue)
```

3. Press *Cmd + Option + Enter* if the canvas is not visible, then click on the **Resume** button above the canvas window to display the resulting view:

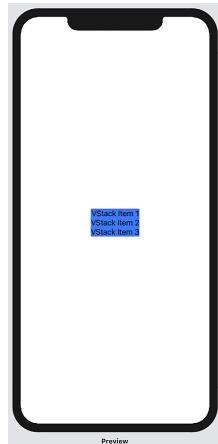


Figure 1.1 – VStack with three items

4. Add a `Spacer()` and a `Divider()` between "VStack Item 2" and "VStack Item 3":

```
Spacer()  
Divider()
```

The content expands and covers the screen's width and height, as follows:

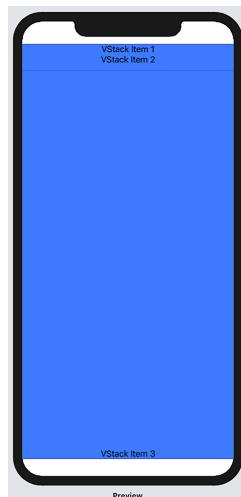


Figure 1.2 – VStack + Spacer + Divider

5. Add an HStack and a ZStack below "VStack Item 3":

```
HStack{  
    Text("Item 1")  
    Divider().background(Color.black)  
    Text("HStack Item 2")  
    Divider()  
    .background(Color.black)  
    Spacer()  
    Text("HStack Item 3")  
}  
.background(Color.red)  
ZStack{  
    Text("ZStack Item 1")  
    .padding()  
    .background(Color.green)  
    .opacity(0.8)  
    Text("ZStack Item 2")  
    .padding()  
    .background(Color.green)  
    .offset(x: 80, y: -400)  
}
```

The preview should now look like this:

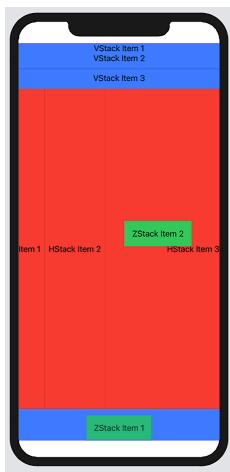


Figure 1.3 – VStack, HStack, and ZStack

This concludes our recipe on using stacks. Going forward, we'll make extensive use of `vStack` and `HStack` to position various components in our views.

How it works...

A new SwiftUI project starts with a single `Text` view located at the center of the screen. The body view only returns a single view. To add other views vertically, we replace the initial `Text` view with a `VStack`, and then add some content. SwiftUI stacks determine how to display content by using the following steps:

1. Figure out its internal spacing and subtract that from the size proposed by its parent view.
2. Divide the remaining space into equal parts.
3. Process the size of its least flexible view.
4. Divide the remaining unclaimed space by the unallocated space, and then repeat *Step 2*.
5. The stack then aligns its content and chooses its own size to exactly enclose its children.

Adding the `Spacer()` forces the view to use the maximum amount of vertical space. This is because the `Spacer()` is the most flexible view—it fills the remaining space after all other views have been displayed.

The `Divider()` component is used to draw a horizontal line across the width of its parent view. That is why adding a `Divider()` view stretched the `VStack` background from just around the `Text` views to the entire width of the `VStack`.

By default, the divider line does not have a color. To set the divider color, we add the `.background(Color.black)` modifier. Modifiers are methods that can be applied to a view to return a new view. In other words, it applies changes to a view. Examples include `.background(Color.black)`, `.padding()`, and `.offsets(...)`.

The `HStack` is like the `VStack` but its contents are displayed horizontally from left to right. Adding a `Spacer()` in an `HStack` thus causes it to fill all available horizontal space, and a divider draws a vertical line between components in the `HStack`.

The `ZStack` is like `HStack` and `VStack` but overlays its content on top of existing items.

There's more...

You can also use the `.frame` modifier to adjust the width and height of a component. Try deleting the `Spacer()` and `Divider()` from the `HStack` and then apply the following modifier to the `HStack`:

```
.frame(  
    maxWidth: .infinity,  
    maxHeight: .infinity,  
    alignment: .topLeading  
)
```

Dealing with text

The most basic building block of any application is text, which we use to provide or request information from a user. Some text requires special treatment, such as password fields, which must be hidden.

In this recipe, we will implement different types of SwiftUI `Text` views. A `Text` view is used to display one or more lines of read-only text on the screen. A `TextField` view is used to display multiline editable text, and a `SecureField` view is used to request private information that should be hidden, such as passwords.

Getting ready

Create a new SwiftUI project named `FormattedText`.

How to do it...

We'll implement multiple types of text-related views and modifiers. Each step in this section applies minor changes to the view, so note the UI changes that occur after each step. Let's get started:

1. Enclose the initial `ContentView` text in a `VStack`:

```
struct ContentView: View {  
    var body: some View {  
        VStack{  
            Text("Hello World")  
        }  
    }  
}
```

2. Add the `.fontWeight(.medium)` modifier to the text and observe the text weight change in the canvas preview:

```
Text("Hello World")
    .fontWeight(.medium)
```

3. Add two state variables to the `ContentView.swift` file: `password` and `someText`. Place the values below the `ContentView` struct declaration. These variables will hold the content of the user's password and `Textfield` inputs:

```
struct ContentView: View {
    @State var password = "1234"
    @State var someText = "initial text"
    var body: some View {
        ...
    }
}
```

4. Now, add `SecureField` and a `Text` view to the `VStack`. The `Text` view displays the value entered in `SecureField`:

```
SecureField("Enter a password", text: $password)
    .padding()
Text("password entered: \(password)")
    .italic()
```

5. Add `TextField` and a `Text` view to display the value entered in `TextField`:

```
TextField("Enter some text", text: $someText)
    .padding()
Text("\(someText)")
    .font(.largeTitle)
    .underline()
```

6. Now, let's add some other text and modifiers to the list:

```
Text("Changing text color and make it bold")
    .foregroundColor(Color.blue)
    .bold()
Text("Use kerning to change space between lines
    of text")
    .kerning(7)
```

```
Text("Changing baseline offset")
    .baselineOffset(100)
Text("Strikethrough")
    .strikethrough()
Text("This is a multiline text implemented in
SwiftUI. The trailing modifier was added
to the text. This text also implements
multiple modifiers")
    .background(Color.yellow)
    .multilineTextAlignment(.trailing)
    .lineSpacing(10)
```

7. Run the code in a simulator or click the **Play** button in the Canvas **Preview**. Enter some text in the `SecureField` and `TextField`. The resulting preview should look like this:

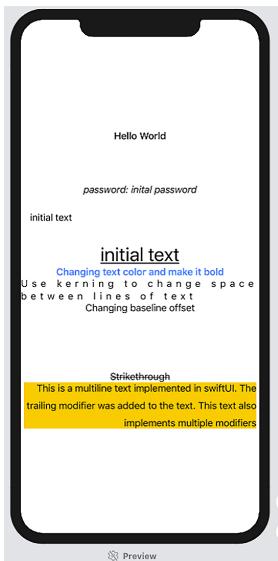


Figure 1.4 – FormattedText preview

Text entered in the `SecureField` will be masked, while text in the `TextField` will be in cleartext.

How it works...

Text views have several modifiers for font, spacing, and other formatting requirements. When in doubt, type . after the Text view and choose from a list of possible options that will appear. This is shown in the following example:

The screenshot shows a portion of Xcode's code editor with the following code:

```
14     Text("Hello World").  
15     M         Text baselineOffset(baselineOffset: CGFloat)  
16     M         Text bold()  
17     M         Text color(Color?)  
18     M         Text font(Font?)  
19     M         Text fontWeight(Font.Weight?)  
20     M         Text foregroundColor(Color?)  
21     M         Text italic()  
22     M         Text italicColor(Color?)  
23     M         Text kerning(CGFloat)  
24     M         Text self  
25  
26             Sets the baseline offset for the text.  
27             .strikeThrough()  
28             Text("Underline")  
29                 .underline()  
30             Text("Italized text")  
31                 .italic()
```

A callout bubble is positioned over the line `Text baselineOffset(baselineOffset: CGFloat)`, listing various modifier methods starting with 'M'. The description for `baselineOffset` is visible in the callout.

Figure 1.5 – Using Xcode autocomplete to view formatting options

Unlike regular Text views, TextField and SecureField require state variables to store the value entered by the user. State variables are declared using the `@State` keyword. SwiftUI manages the storage of properties declared by using `@State` and refreshes the body each time the value of the state variable changes.

Values entered by the user are stored using the process of binding. In this recipe, we have state variables bound to the SecureField and TextField input parameter. The `$` symbol is used to bind a state variable to the field. Using the `$` symbol ensures that the state variable's value is changed to correspond to the value entered by the user, as shown in the following example:

```
TextField("Enter some text", text: $someText)
```

Binding also notifies other views of state changes and causes the views to be redrawn on state change.

The content of bound state variables is displayed without using the `$` symbol because no binding is required when displaying content, as shown in the following code snippet:

```
Text("\(someText)")
```

There's more...

Try adding an eleventh element to the `VStack`. Xcode displays an error message because SwiftUI views can hold a maximum of 10 elements. To add an eleventh element, you would have to enclose some elements in a `Group` view such that the total number of child views in the `VStack` is equal to or less than 10. A `Group` view can be used as shown in the following code snippet:

```
Group {  
    Text("Item 1")  
    ...  
    Text("Item 10")  
}  
Text("Item 11")
```

Items in a `Group` are considered as one view. The preceding code snippet thus allows us to display eleven items as two views—the `Group` view and the `Text` view at the bottom.

See also

Apple documentation regarding SwiftUI `Text` view: <https://developer.apple.com/documentation/swiftui/text>

Using images

In this recipe, we will learn how to add an image to a view, use an already existing `UIImage` put an image in a frame, and use modifiers to present beautiful images. The images in this section were obtained from <https://unsplash.com/>, so special thanks to jf-brou, Kieran White, and Camilo Fierro.

Getting ready

Let's start by creating a new SwiftUI project called `ImageApp`.

How to do it...

Let's add some images to our SwiftUI project and introduce the modifiers used to style them. The steps are given here:

1. Replace the initial Text view with a VStack.
2. Download the project images from the GitHub link at <https://github.com/PacktPublishing/SwiftUI-Cookbook-2nd-Edition/tree/main/Resources/Chapter01/recipe3>.
3. Drag and drop the downloaded images for this recipe into the project's Assets.xcassets (or Assets) folder, as shown in the following screenshot:

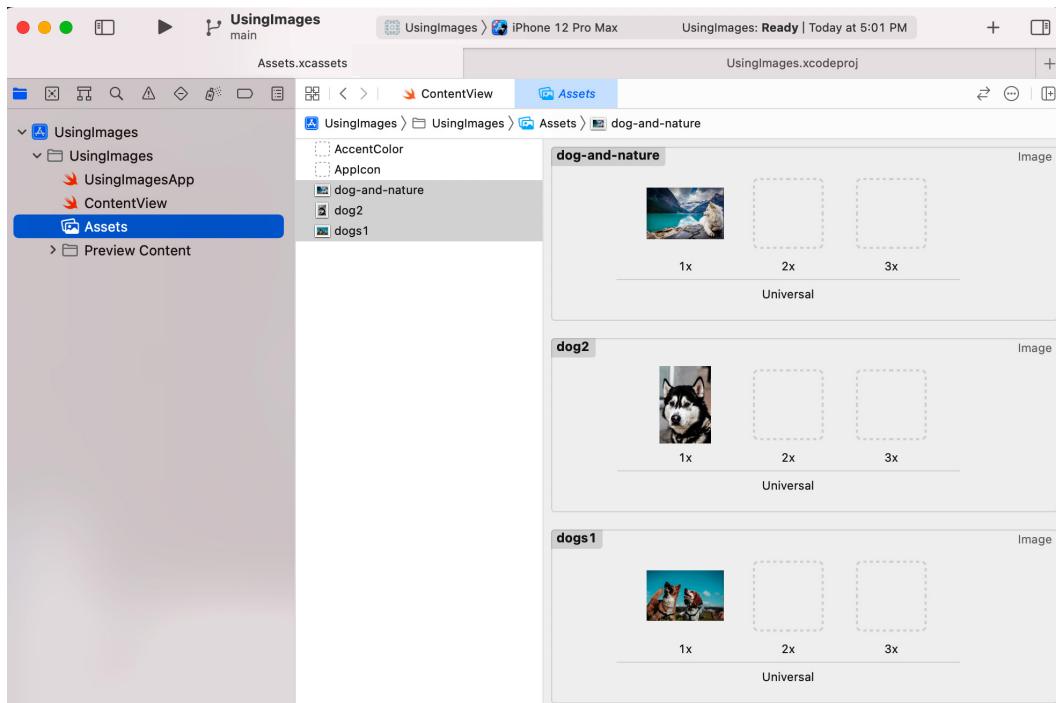


Figure 1.6 – Assets.xcassets folder in Xcode

4. Add an Image view to VStack:

```
Image("dogs1")
```

Observe the result in the canvas preview.

5. Add a `.resizable()` modifier to the image and allow SwiftUI to adjust the image such that it fits the screen space available:

```
Image("dogs1")
    .resizable()
```

6. The `.resizable()` modifier causes the full image to fit on the screen, but the proportions are distorted. That can be fixed by adding the `.aspectRatio(contentMode: .fit)` modifier:

```
Image("dogs1")
    .resizable()
    .aspectRatio(contentMode: .fit)
```

7. Add the dog-and-nature image to `VStack`:

```
Image("dog-and-nature")
    .resizable()
    .aspectRatio(contentMode: .fit)
    .frame(width:300, height:200)
    .clipShape(Circle())
    .overlay(Circle().stroke(Color.blue, lineWidth: 6))
    .shadow(radius: 10)
```

8. To use a `UIImage` class as input for images, create a `getImageFromUIImage(image: String)` function that accepts an image name and returns a `UIImage`:

```
func getImageFromUIImage(image:String) -> UIImage {
    guard let img = UIImage(named: image) else {
        fatalError("Unable to load image")
    }
    return img
}
```

9. Use `getImageFromUIImage(image: String)` to display a `UIImage` within the `VStack`. The resulting code should look like this:

```
struct ContentView: View {
    var body: some View {
        VStack{
            Image("dogs1")
                .resizable()
                .aspectRatio(contentMode: .fit)
            Image("dog-and-nature")
                .resizable()
                .aspectRatio(contentMode: .fit)
                .frame(width:300, height:200)
                .clipShape(Circle())
                .overlay(Circle().stroke(Color.blue,
                    lineWidth: 6))
                .shadow(radius: 10)
            Image(uiImage: getImageFromUIImage(image:
                "dog2"))
                .resizable()
                .frame(width: 200, height: 200)
                .aspectRatio(contentMode: .fit)

        }
    }
}

func getImageFromUIImage(image:String) -> UIImage {
    guard let img = UIImage(named: image) else {
        fatalError("Unable to load image")
    }
    return img
}
```

The completed application should then look like this:



Figure 1.7 – ImageApp preview

How it works...

Adding the `Image` view to SwiftUI displays the image in its original proportions. The image might be too small or too big for the device's display. For example, without any modifiers, the dog-and-nature image fills up the full iPhone 11 Pro Max screen:



Figure 1.8 – dog-and-nature image without the resizable modifier

To allow an image to shrink or enlarge to fit the device screen size, add the `.resizable()` modifier to the image. Adding the `.resizable()` modifier causes the image to fit within its view, but it may be distorted due to changes in proportion:



Figure 1.9 – Image with resizable modifier

To address the issue, add the `.aspectRatio(contentMode: .fit)` modifier to the image:



Figure 1.10 – Image with AspectRatio set

To specify the width and height of an image, add the `.frame(width, height)` modifier to the view and set the width and height: `.frame(width: 200, height: 200)`.

Images can be clipped to specific shapes. The `.clipShape(Circle())` modifier changes the image shape to a circle:



Figure 1.11 – Image with the `clipShape(Circle())` modifier

The `.overlay(Circle().stroke(Color.blue, lineWidth: 6))` and `.shadow(radius: 10)` modifiers were used to draw a blue line around the image circle and add a shadow to the circle:



Figure 1.12 – Stroke and shadow applied to image

Important Note

The order in which the modifiers are added matters. Adding the `.frame()` modifier before the `.resizable()` modifier may lead to different results or cause an error.

See also

Apple documentation regarding SwiftUI images: <https://developer.apple.com/documentation/swiftui/image>

Adding buttons and navigating with them

In this recipe, we will learn how to use various buttons available in SwiftUI. We will use a `Button` view to trigger the change of a count when clicked and implement a `NavigationView` to move between various SwiftUI views and an `EditButton` to remove items from a list. We will also briefly discuss the `MenuButton` and `PasteButton` only available in macOS.

Getting ready

Let's start by creating a new SwiftUI project called `ButtonsApp`.

How to do it...

Let's create a home screen with buttons for each of the items we want to go over. Once clicked, we'll use SwiftUI's navigation view to go to the view that implements the clicked concept. The steps are given here:

1. Add a new SwiftUI view file called `ButtonView` to the project: **File | New | File** (or press the shortcut keys `⌘ + N`).
2. Select **SwiftUI View** from the UI templates.
3. In the **Save As** field of the pop-up menu, enter the filename `ButtonView`.
4. Repeat *Step 1* and enter the filename `EditButtonView`.
5. Repeat *Step 1* and enter the filename `PasteButtonView`.
6. Repeat *Step 1* and enter the filename `MenuButtonView`.

Important Note

Avoid using the `MenuButton` view because this is deprecated and only available in macOS 10.14-12.0. For similar functionality, use the `Menu` view instead.

7. Open the ContentView.swift file and create a NavigationView to navigate between the SwiftUI views we added to the project:

```
NavigationView {
    VStack{
        NavigationLink(destination: ButtonView()) {
            Text("Buttons")
        }

        NavigationLink(destination:EditButtonView()) {
            Text("EditButtons")
            .padding()
        }

        NavigationLink(destination:MenuButtonView()) {
            Text("MenuButtons")
            .padding()
        }

        NavigationLink(destination:PasteButtonView()) {
            Text("PasteButtons")
            .padding()
        }

        NavigationLink(destination:
            Text("Very long text that should not be displayed in a
                  single line because it is not good design")
            .padding()
            .navigationBarTitle(Text("Detail"))
        ) {
            Text("details about text")
            .padding()
        }

    }.navigationBarTitle(Text("Main View"), displayMode:
        .inline)
}
```

Upon completion, the `ContentView` preview should look like this:

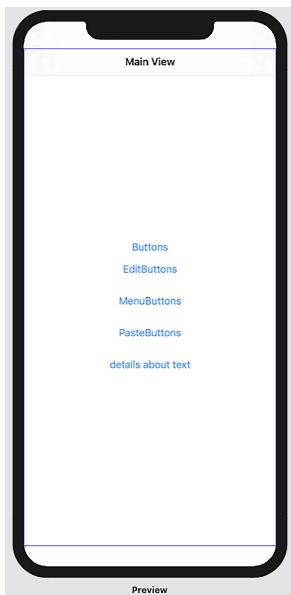


Figure 1.13 – ButtonsApp ContentView

8. Open the `EditButtonView.swift` file in the project navigator and add the following code that implement an `EditButton`:

```
@State private var animals = ["Cats", "Dogs", "Goats"]
var body: some View {
    NavigationView{
        List{
            ForEach(animals, id: \.self){ animal in
                Text(animal)
            }.onDelete(perform: removeAnimal)
        }
        .navigationBarItems(trailing: EditButton())
        .navigationBarTitle(Text("EditButtonView"),
                           displayMode: .inline)
    }
}
func removeAnimal(at offsets: IndexSet) {
    animals.remove(atOffsets: offsets)
}
```

-
9. Open the `MenuButtonView.swift` file and add the code for MenuButtons:

```
var body: some View {
    Text("MenuButtons are currently available on MacOS
        currently")
    .padding()
    .navigationBarTitle("MenuButtons", displayMode:
        .inline)
    /*
    MenuButton("country +") {
        Button("USA") { print("Selected USA") }
        .background(Color.accentColor)
        Button("India") { print("Selected India") }
    }
    */
}
```

10. Open the `PasteButtonView.swift` file and implement the text regarding PasteButtons:

```
@State var text = String()
var body: some View {
    VStack{
        Text("PasteButton controls how you paste in macOS but
            is not available in iOS. For more information,
            check the \"See also\" section of this recipe")
        .padding()
    }.navigationBarTitle("PasteButton",
        displayMode:.inline)
}
```

Go back to `ContentView` and run the code in the canvas preview or simulator and play around with it to see what the results look like.

How it works...

A `NavigationLink` must be placed in a `NavigationView` prior to being used.

In this recipe, we use a `NavigationLink` with two parameters—`destination` and `label`. The `destination` parameter represents the view that would be displayed when the `label` is clicked, while the `label` parameter represents the text to be displayed within `NavigationLink`.

`NavigationLink` buttons can be used to move from one SwiftUI view file to another—for example, moving from `ContentView` to `EditButtonView`. They can also be used to display text details without creating a SwiftUI view file, such as in the last `NavigationLink`, where a click just presents a long piece of text with more information. This is made possible because the `Text` struct conforms to the `view` protocol.

The `.navigationBarTitle(Text("Main View"), displayMode:.inline)` modifier adds a title to the `ContentView` screen. The first argument passed to the modifier represents the title to be displayed, and the `displayMode` argument controls the style for displaying the navigation bar.

The `.navigationBarTitle()` modifier is also added to `EditButtonView` and other views. Since these views do not contain `NavigationView` structs, the titles would not be displayed when viewing the page directly from the preview but would show up when running the code and navigating from `ContentView.swift` to the view provided in `NavigationLink`.

The `EditButton` view is used in conjunction with `List` views to make lists editable. We will go over `List` and `Scroll` views in *Chapter 2, Going Beyond the Single Component with List and Scroll Views*, but `EditButtonView` provides a peek into how to create an editable list.

`MenuButtons` and `PasteButtons` are only available on macOS. This recipe provides some sample code for menu buttons. Refer to the *See also* section of this recipe for code on how the `PasteButton` is implemented.

See also

The code for implementing PasteButtons can be found here: <https://gist.github.com/sturdysturge/79c73600cfb683663c1d70f5c0778020#file-swiftuidocumentationpaste>.

More information regarding NavigationLink buttons can be found here: <https://developer.apple.com/documentation/swiftui/navigationlink>.

Beyond buttons – using advanced pickers

In this recipe, we will learn how to implement pickers—namely, Picker, Toggle, Slider, Stepper, and DatePicker. Pickers are typically used to prompt the user to select from a set of mutually exclusive values. Toggle views are used to switch between on/off states. Slider views are used for selecting a value from a bounded linear range of values. As with Slider views, Stepper views also provide a UI for selecting from a range of values. However, steppers use the + and – signs to allow users to increment the desired value by a certain amount. Finally, DatePicker views are used for selecting dates.

Getting ready

Create a new SwiftUI project named `PickersApp`.

How to do it...

Let's create a SwiftUI project that implements various pickers. Each picker will have a `@State` variable to hold the current value of the picker. The steps are given here:

1. In the `ContentView.swift` file, create `@State` variables that will hold the values selected by the pickers and other controls. Place the variables between the `ContentView` struct and the body:

```
@State var choice = 0
@State var showText = false
@State var transitModes = ["Bike", "Car", "Bus"]
@State var sliderVal: Float = 0
@State var stepVal = 0
@State var gameTime = Date()
```

2. Create a Form view within the body view of the ContentView struct. Then, add a Section view and a Picker view to the form:

```
Form{  
    Section{  
        Picker("Transit Modes", selection: $choice){  
            ForEach( 0 ..< transitModes.count) { index  
                in  
                    Text("\(self.transitModes[index])")  
            }  
        }.pickerStyle(SegmentedPickerStyle())  
        Text("Current choice:  
            \(transitModes[choice])")  
    }  
}
```

3. Add an additional Section view and a Toggle view:

```
Section{  
    Toggle(isOn: $showText){  
        Text("Show Text")  
    }  
    if showText {  
        Text("The Text toggle is on")  
    }  
}
```

4. Add a Section view and a Slider view:

```
Section{  
    Slider(value: $sliderVal, in: 0...10,  
        step: 0.001)  
    Text("Slider current value  
        \(sliderVal, specifier: "%.1f")")  
}
```

5. Add a Section view and a Stepper view:

```
Section {  
    Stepper("Stepper", value: $stepVal,  
            in: 0...5)  
    Text("Stepper current value  
        \(stepVal)")  
}
```

6. Add a Section view and a DatePicker view:

```
Section {  
    DatePicker("Please select a date",  
              selection: $gameTime)  
}
```

7. Add a Section view and a slightly modified DatePicker view that only accepts future dates:

```
Section {  
    DatePicker("Please select a date",  
              selection: $gameTime, in: Date()...)  
}
```

The result should be a beautiful format, similar to what is shown here:

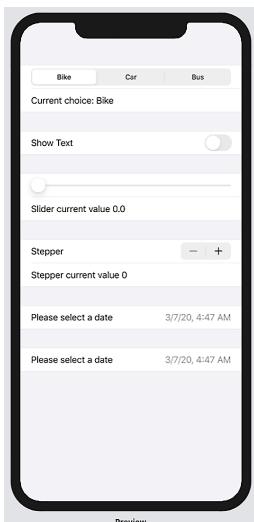


Figure 1.14 – SwiftUI Form with Pickers

How it works...

Form views group controls are used for data entry, and Section views create hierarchical view content. Form and Section views can be coupled to generate the gray *padding* area between each component.

Picker views are used for selecting from a set of mutually exclusive values. In the following example, a picker is used to select a transit mode from our `transitModes` state variable:

```
Picker(selection: $choice, label:Text("Transit Mode")) {  
    ForEach( 0 ..< transitModes.count) { index in  
        Text("\\"(self.transitModes[index])")  
    }  
}.pickerStyle(SegmentedPickerStyle())
```

As shown in the preceding example, a Picker view takes two parameters, a string describing its function, and a state variable that holds the value selected. The state variable should be of the same type as the range of values to select from. In this case, the `ForEach` loop iterates through the `transitModes` array indices. The value selected would be an `Int` within the range of `transitModes` indices. The transit mode located in the selected index can then be displayed using `Text("\\"(self.transitModes[index])")`.

Toggle views are controls that switch between "on" and "off" states. The state variable for holding the toggle selection should be of the `Bool` type. The section with the Toggle view also contains some text. The current value of the toggle's `@State` property stores the current value of the Toggle.

Creating a slider requires three arguments:

- `value`: The `@State` variable to bind the user input to
- `in`: The range of the slider
- `step`: By how much the slider should change when the user moves it

In the sample code, our slider moves can hold values between 0 and 10, with a step of 0.001.

Steppers take three arguments too—a string for the label, `value`, and `in`. The `value` argument holds the `@State` variable that binds the user input, and the `in` argument holds the range of values for the stepper.

In this recipe, we also demonstrate two applications of a date picker. The first from the top shows a date picker whose first argument is the label of `DatePicker`, and the second argument holds the state variable that binds the user input. Use it in situations where the user is allowed to pick any date without restriction. The other date picker contains a third parameter, `in`. This parameter represents the date range the user can select.

Important Note

The `@State` variables should be of the same type as the data to be stored.

For example, the `gameTime` state variable is of the `Date` type.

Picker styles change based on its ancestor. The default appearance of a picker may be different when placed within a form or list instead of a `VStack` or some other view. Styles can be overridden using the `.pickerStyle(. . .)` modifiers.

Applying groups of styles using ViewModifier

SwiftUI comes with built-in modifiers such as `background()` and `fontWeight()`, among others. It also gives you the ability to create your own custom modifiers. You can use custom modifiers to combine multiple existing modifiers into one.

In this section, we will create a custom modifier that adds rounded corners and a background to a `Text` view.

Getting ready

Create a new SwiftUI project named `UsingViewModifiers`.

How to do it...

Let's create a view modifier and use a single line of code to apply it to a `Text` view. The steps are given here:

1. Change the string in the `ContentView` view to "Perfect":

```
Text ("Perfect")
```

2. In the `ContentView.swift` file, create a struct that conforms to the `ViewModifier` protocol, accepts a parameter of type `Color`, and applies styles to the view's body:

```
struct BackgroundStyle: ViewModifier {  
    var bgColor: Color  
    func body(content: Content) -> some View{  
        content  
            .frame(width:UIScreen.main.bounds.width * 0.3)  
            .foregroundColor(Color.black)  
            .padding()  
            .background(bgColor)  
            .cornerRadius(CGFloat(20))  
    }  
}
```

3. Add a custom style to the text using the `modifier()` modifier:

```
Text ("Perfect") .modifier(BackgroundStyle(bgColor:  
    .blue))
```

4. To apply styles without using a modifier, create an extension to the `View` protocol:

```
extension View {  
    func backgroundStyle(color: Color) -> some View{  
        self.modifier(BackgroundStyle(bgColor: color))  
    }  
}
```

5. Remove the modifier on the `Text` view and add your custom styles using the `backgroundStyle()` modifier that you just created:

```
Text ("Perfect")  
    .backgroundStyle(color: Color.red)  
}
```

The result should look like this:

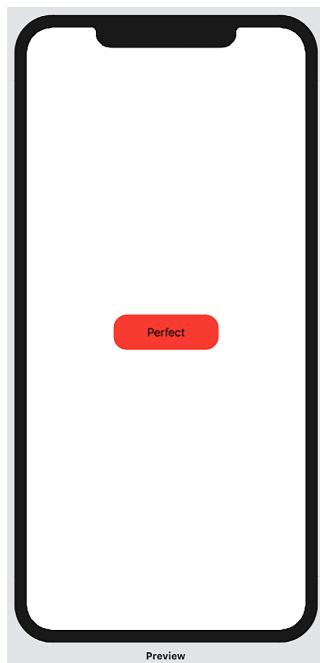


Figure 1.15 – Custom view modifier

This concludes the section on view modifiers. View modifiers promote clean coding and reduce repetition.

How it works...

A `ViewModifier` creates a new view by altering the original view to which it is applied. We create a new view modifier by creating a struct that conforms to the `view` protocol and applied our styles to the struct's body view. You can make the `ViewModifier` customizable by requiring input parameters/properties that would be used when applying styles.

In the example here, the `bgColor` property is used in our `BackGroundStyle` struct that alters the background color of the content passed to the `body` function.

At the end of *Step 2*, we have a functioning `ViewModifier` but decide to make it easier to use by creating a `View` extension and adding in a function that calls our struct:

```
extension View {  
    func backgroundStyle(color: Color) -> some View{  
        self.modifier(BackgroundStyle(bgColor: color))  
    }  
}
```

We are thus able to use `.backgroundStyle(color: Color)` directly on our views instead of `.modifier(BackgroundStyle(bgColor:Color))`.

See also

Apple documentation on view modifiers: <https://developer.apple.com/documentation/swiftui/viewmodifier>

Separating presentation from content with `ViewBuilder`

Apple defines `ViewBuilder` as "a custom parameter attribute that constructs views from closures". `ViewBuilder` can be used to create custom views that can be used across an application with minimal or no code duplication. In this recipe, we will create a SwiftUI view, `BlueCircle`, that applies a blue circle to the right of its content.

Getting ready

Let's start by creating a new SwiftUI project called `UsingViewBuilder`.

How to do it...

We'll create our `ViewBuilder` in a separate swift file and then apply it to items that we'll create in the `ContentView.swift` file. The steps are given here:

1. With our `UsingViewBuilder` app opened, let's create a new SwiftUI file by going to **File | New | File**.
2. Select SwiftUI view from the menu and click **Next**.
3. Name the file `BlueCircle` and click **Create**.
4. Delete the `BlueCircle_Previews` struct from the file.

-
5. Add the `BlueCircle` `ViewModifier` to the file:

```
struct BlueCircle<Content: View>: View {  
    let content: Content  
    init(@ViewBuilder content: () -> Content) {  
        self.content = content()  
    }  
    var body: some View {  
        HStack {  
            content  
            Spacer()  
            Circle()  
                .fill(Color.blue)  
                .frame(width:20, height:30)  
        }.padding()  
    }  
}
```

6. Open the `ContentView.swift` file and try out the `BlueCircle` `ViewBuilder`:

```
var body: some View {  
    VStack {  
        BlueCircle {  
            Text("some text here")  
            Rectangle()  
                .fill(Color.red)  
                .frame(width: 40, height: 40)  
        }  
        BlueCircle {  
            Text("Another example")  
        }  
    }  
}
```

The resulting preview should look like this:

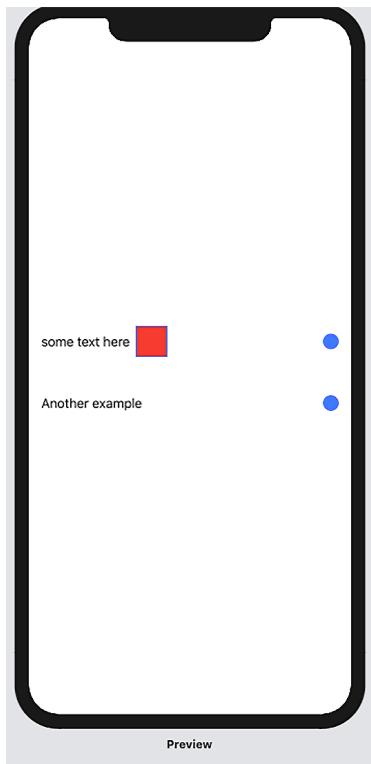


Figure 1.16 – ViewBuilder result preview

How it works...

We use the `ViewBuilder` struct to create a view template that can be used anywhere in the project without duplicating code. The `ViewBuilder` struct must contain a `body` property since it extends the `View` protocol.

Within the `body` property/`view`, we update the `content` property with the components we want to use in our custom view. In this case, we use a `BlueCircle`. Notice the location of the `content` property. This determines the location where the element added to our `ViewBuilder` would be placed.

See also

Apple documentation on ViewBuilder: <https://developer.apple.com/documentation/swiftui/viewbuilder>

Simple graphics using SF Symbols

The SF Symbols provides a set of over 3,200 free consistent and highly configurable symbols.

You can download and browse through a list of SF symbols using the macOS app available for download here: <https://devimages-cdn.apple.com/design/resources/download/SF-Symbols-3.dmg>.

In this recipe, we will use SF symbols in labels and images. We'll also apply various modifiers that will add a punch to your design.

Getting ready

Let's start by creating a new SwiftUI project called `UsingSFSymbols`.

How to do it...

Let's create an app where we use different combinations of SF Symbols and modifiers. The steps are given here:

1. Open the `ContentView.swift` file and replace the initial `Text` view with a `VStack`, `HStack`, and some SF Symbols:

```
    VStack {
        HStack{
            Image(systemName: "c")
            Image(systemName: "o")
            Image(systemName: "o")
            Image(systemName: "k")

        }.symbolVariant(.fill.circle)
            .foregroundStyle(.yellow, .blue)
            .font(.title)
```

2. Add an HStack with SF Symbols for the word book:

```
HStack{  
    Image(systemName: "b.circle.fill")  
    Image(systemName: "o.circle.fill")  
        .foregroundStyle(.red)  
    Image(systemName: "o.circle.fill")  
        .imageScale(.large)  
    Image(systemName: "k.circle.fill")  
        .accessibility(identifier: "Letter K")  
}.foregroundColor(.blue)  
    .font(.title)  
    .padding()
```

3. Let's add another HStack with more SF Symbols:

```
HStack{  
    Image(systemName: "allergens")  
    Image(systemName: "ladybug")  
}.symbolVariant(.fill)  
    .symbolRenderingMode(.multicolor)  
    .font(.largeTitle)
```

4. Finally, let's add a Toggle view that changes the color of the Wi-Fi SF Symbol based on the toggle state:

```
Toggle(isOn: $wifi_on) {  
    Label("Wifi", systemImage: "wifi")  
}.foregroundStyle(wifi_on ? .blue :  
    .secondary)  
.padding()
```

The resulting preview should look like this:



Figure 1.17 – SF Symbols in action

How it works...

SF Symbols defines several design variants such as enclosed, fill, and slash. These different variants can be used to convey different information—for example, a slash variant on a Wi-Fi symbol lets the user know if the Wi-Fi is unavailable.

In our first `HStack` we use the `.symbolVariant(.fill.circle)` modifier to apply the `.fill` and `.circle` variants to all the items in the `HStack`. This could also be accomplished using the following code:

```
HStack{  
    Image(systemName: "c.circle.fill")  
    Image(systemName: "o.circle.fill ")  
    Image(systemName: "o.circle.fill ")  
    Image(systemName: "k.circle.fill ")  
}
```

However, the preceding code is too verbose and would require too many changes if we decided that we didn't need either the `.circle` or `.fill` variant, or both.

We also notice something new in our first `HStack` —the `.foregroundStyle(...)` modifier. The `.foregroundStyle` modifier can accept one, two, or three parameters corresponding to the primary, secondary, and tertiary colors. Some symbols may have all three levels of colors, or only primary and secondary, or primary and tertiary. For symbols without all three layers, only the ones that pertain to them are applied to the symbol. For example, a tertiary color applied to an SF Symbol with only primary and secondary layers will have no effect on the symbol.

The second `HStack` also uses the `.symbolVariant` modifier with one variant. It also introduces a new modifier, `.symbolRenderingMode()`. Rendering modes can be used to control how color is applied to symbols. The multicolor rendering mode renders symbols as multiple layers with their inherited styles. Adding `.multicolor` rendering mode is enough to present a symbol with its default layer colors. Other rendering modes include `hierarchical`, `monochrome`, and `palette`.

Finally, we create a toggle for a Wi-Fi label where we change the foreground based on the status of the `wifi_on` state variable. The toggle reads the state variable and changes the `wifi` symbol color to blue when turned on or gray when off.

See also

More about WWDC SwiftUI SF Symbols can be found here: <https://developer.apple.com/videos/play/wwdc2021/10349>.

Integrating UIKit into SwiftUI – the best of both worlds

SwiftUI was announced at WWDC 2019 and is only available on devices running iOS 13 and above. Due to its relative immaturity, SwiftUI may lack broad API coverage compared to UIKit. For example, as of July 2021, at the time of this writing, pictures and movies can only be handled using UIKit's `UIImagePickerController`. There is, therefore, a need to implement certain UIKit APIs in SwiftUI.

In this recipe, we'll look at how to integrate UIKit APIs in SwiftUI. We will create a project that wraps instances of `UIActivityIndicatorView` to display an indicator in SwiftUI.

Getting ready

Open Xcode and create a SwiftUI project named `UIKitToSwiftUI`.

How to do it...

We can display UIKit views in SwiftUI by using the `UIViewRepresentable` protocol. Follow these steps to implement the `UIActivityIndicatorView` in SwiftUI:

1. Within the Xcode menu, click **File** | **New** | **File** and select **Swift File**. Name the view `ActivityIndicator`.
2. Replace the `import Foundation` statement with `import SwiftUI`:

```
import SwiftUI
```

3. Modify the code in `ActivityIndicator` to use the `UIViewRepresentable` protocol:

```
struct ActivityIndicator: UIViewRepresentable {  
    var animating: Bool  
  
    func makeUIView(context: Context) ->  
        UIActivityIndicatorView {  
        return UIActivityIndicatorView()  
    }  
  
    func updateUIView(_ activityIndicator:  
        UIActivityIndicatorView, context: Context) {  
        if animating {  
            activityIndicator.startAnimating()  
        } else {  
            activityIndicator.stopAnimating()  
        }  
    }  
}
```

- Let's open the `ContentView.swift` file and add the following code to make use of the `ActivityIndicator` instance that we just created. Let's also add a `Toggle` control to turn the indicator on or off:

```
struct ContentView: View {  
    @State var animate = true  
    var body: some View {  
        VStack{  
            ActivityIndicator(animating:  animate)  
            HStack{  
                Toggle(isOn: $animate) {  
                    Text("Toggle Activity")  
                }  
            }.padding()  
        }  
    }  
}
```

The resulting `ContentView` preview should look like this:

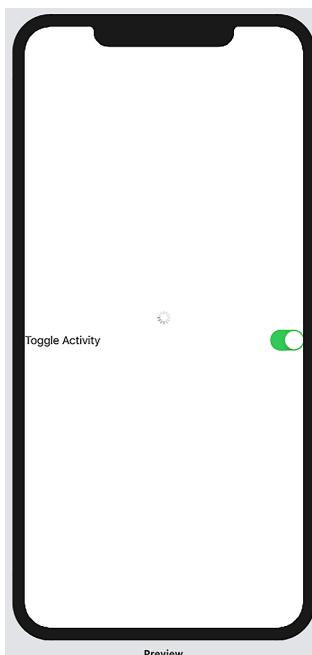


Figure 1.18 – UIKit `ActivityIndicator` in SwiftUI

How it works...

UIKit views can be implemented in SwiftUI by using the `UIViewRepresentable` protocol to wrap the UIKit views. In this recipe, we make use of a `UIActivityIndicatorView` by first wrapping it with a `UIViewRepresentable`.

In our `ActivityIndicator.swift` file, we implement a struct that conforms to the `UIViewRepresentable` protocol. This requires us to implement both the `makeUIView` and `updateUIView` functions. The `makeUIView` function creates and prepares the view, while the `updateUIView` function updates the `UIView` when the animation changes.

Important Note

You can implement the preceding features in iOS 14+ apps by using SwiftUI's `ProgressView`.

See also

Check out the *Exploring more views and controls* recipe at the end of this chapter for more information on `ProgressView`.

Adding SwiftUI to an existing app

In this recipe, we will learn how to navigate from a UIKit view to a SwiftUI view while passing a secret text to our SwiftUI view.

We'll be making use a UIKit storyboard, a visual representation of the UI in UIKit. The `Main.storyboard` file is to UIKit what the `ContentView.swift` file is to SwiftUI. They are both the default home views that are created when you start a new project.

We start off this project with simple UIKit project that contains a button.

Getting ready

Get the following ready before starting out with this recipe:

1. Clone or download the code for this book from GitHub: <https://github.com/PacktPublishing/SwiftUI-Cookbook-2nd-Edition-/tree/main/Chapter01-Using-the-basic-SwiftUI-Views-and-Controls/10-Adding-SwiftUI-to-UIKit>.
2. Open the `StartingPoint` folder and double-click on `AddSwiftUIToUIKit.xcodeproj` to open the project in Xcode.

How to do it...

We will add a `NavigationController` to the UIKit `ViewController` that allows the app to switch from the UIKit to the SwiftUI view when the button is clicked:

1. Open the `Main.storyboard` file in Xcode by clicking on it. The `Main.storyboard` looks like this:

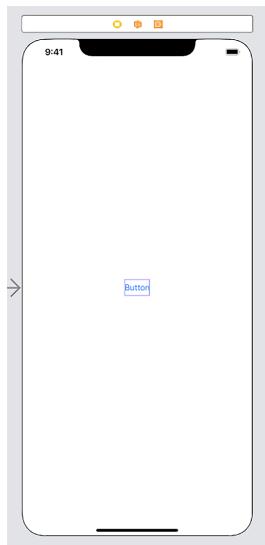


Figure 1.19 – UIKit ViewController

2. Click anywhere in the `ViewController` to select it.
3. In the Xcode menu, click **Editor | Embed in | Navigation Controller**.
4. Add a new `ViewController` to the project:
 - a. Click the `+` button at the top left of the Xcode window.
 - b. In the pop-up menu, type `hosting` and select **Hosting View Controller**:

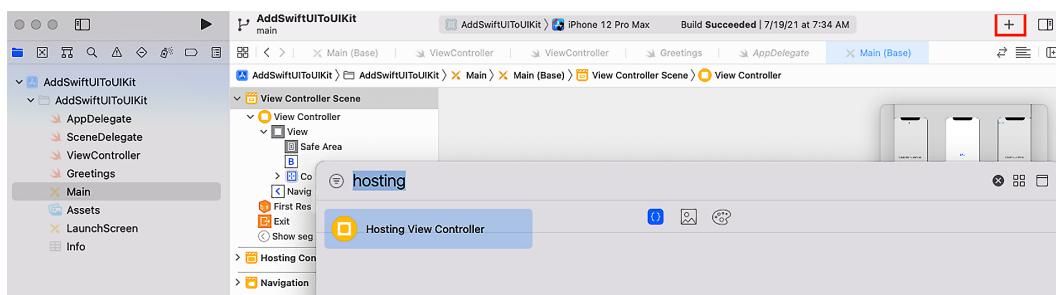


Figure 1.20 – Creating a UIKit ViewController

5. Hold down the *Ctrl* key, and then click and drag from the **ViewController** button to the new **Hosting View Controller** that we added.
6. In the pop-up menu, for the **Action Segue** option, select **Show**.
7. Click the *Adjust Editor Options* button:



Figure 1.21 – Adjust Editor Options button

8. Click **Assistant**. This splits the view into two panes, as shown here:

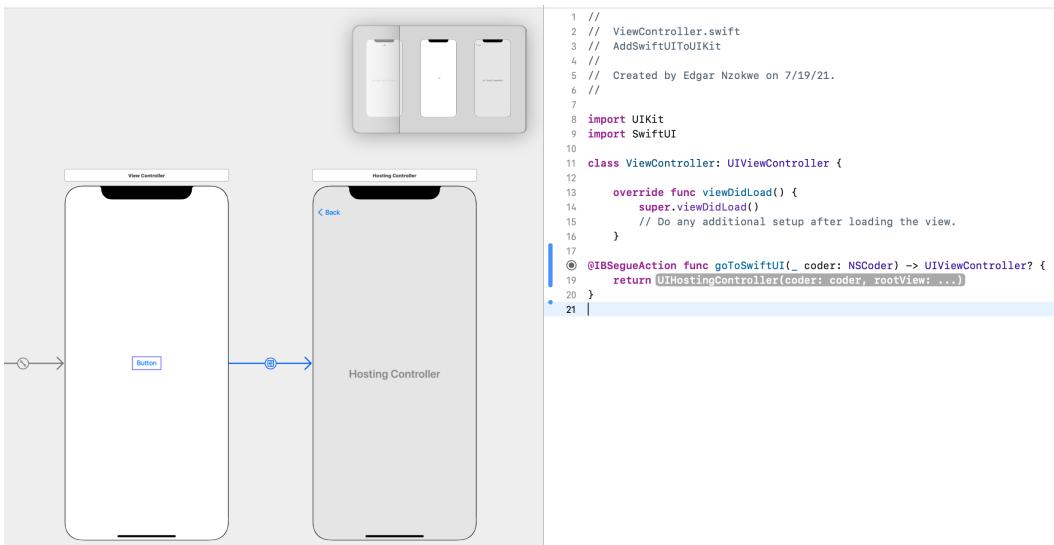


Figure 1.22 – View with Assistant opened

9. To create a segue action, hold the *Ctrl* key, then click and drag from the segue button (item in the middle of the blue arrow in *Figure 1.22*) to the space after the `viewDidLoad` function in the `ViewController.swift` file.
10. In the pop-up menu, enter the name `goToSwiftUI` and click **Connect**. The following code will be added to the `ViewController.swift` file:

```

@IBAction func goToSwiftUI(_ coder: NSCoder)
-> UIViewController? {
    return UIHostingController(coder: coder,
    rootView: rootView)
}

```

11. Add a statement to import SwiftUI at the top of the `ViewController` page, next to `Import UIKit`:

```
Import SwiftUI
```

12. Within the `goToSwiftUI` function, create a "*secret*" text that will be passed to our SwiftUI view. Also, create a `rootView` variable that specifies the SwiftUI view that you would like to reach. Finally, return the `ViewController`, which is used to display the SwiftUI view. The resulting code should look like this:

```
@IBSegueAction func goToSwiftUI(_ coder: NSCoder)
-> UIViewController? {
    let greetings = "Hello From UIKit"
    let rootView = Greetings(textFromUIKit:
        greetings)
    return UIHostingController(coder: coder,
        rootView: rootView)
}
```

At this point, the code will not compile because we have not yet implemented a `Greetings` view. Let's resolve that now.

13. Create a SwiftUI view to display a message:
 - a. Click **File | New | File** and select **SwiftUI View**.
 - b. Name the View `Greetings.swift`.
14. Add a `View` component that displays some text passed to it:

```
struct Greetings: View {
    var textFromUIKit = ""
    var body: some View {
        Text(textFromUIKit)
    }
}
```

Run the project in the simulator, click on the `UIKit` button, and watch the SwiftUI page get displayed.

How it works...

To host SwiftUI views in an existing app, you need to wrap the SwiftUI hierarchy in a `ViewController` or `InterfaceController`.

We start by performing core UIKit concepts, such as adding a `NavigationView` controller to the storyboard and adding a **Hosting View Controller** as a placeholder for our SwiftUI view.

Lastly, we create an `IBSegueAction` to present our SwiftUI view upon clicking the UIKit button.

Exploring more views and controls (iOS 14+)

In this section, we introduce some views and controls that did not clearly fit in any of the earlier created recipes. We'll look at the `ProgressView`, `ColorPicker`, `Link`, and `Menu` views.

`ProgressView` are used to show the degree of completion of a task. There are two types of `ProgressView`: indeterminate progress views show a spinning circle till a task is completed, while determinate progress views show a bar that gets filled up to show the degree of completion of a task.

`ColorPicker` views allow users to select from a wide range of colors, while `Menu` views present a list of items that users can choose from to perform a specific action.

Getting ready

Let's start by creating a new SwiftUI project called `MoreViewsAndControls`.

How to do it...

Let's implement some views and controls in the `ContentView.swift` file. We will implement each item in a separate `Section` within a `List`. The steps are given here:

1. Just below the `ContentView` struct, add the state variables that we'll be using for various components:

```
@State private var progress = 0.5
@State private var color = Color.red
@State private var secondColor = Color.yellow
@State private var someText = "Initial value"
```

2. Add a List view in the body with a Section view and ProgressView components:

```
List{  
    Section(header: Text("ProgressViews")) {  
        ProgressView("Indeterminate progress  
view")  
        ProgressView("Downloading", value:  
            progress, total:2)  
        Button("More") {  
            if(progress < 2) {  
                progress += 0.5  
            }  
        }  
    }  
}
```

3. Let's add another section that implements two labels:

```
Section(header: Text("Labels")) {  
    Label("Slow ", systemImage:  
        "tortoise.fill")  
    Label{  
        Text ("Fast")  
            .font(.title)  
    }  
    icon: {  
        Circle()  
            .fill(Color.orange)  
            .frame(width: 40, height:  
                20, alignment: .center)  
            .overlay(Text("F"))  
    }  
}
```

4. Now, add a new section that implements a ColorPicker:

```
Section(header: Text("ColorPicker")) {  
    ColorPicker(selection: $color) {  
        Text("Pick my background")  
            .background(color)  
            .padding()  
    }  
    ColorPicker("Picker", selection:  
        $secondColor)  
}
```

5. Next, add a TextEditor:

```
Section(header: Text("TextEditor")) {  
    TextEditor(text: $someText)  
    Text("current editor  
text:\n\\(someText)")  
}
```

6. Then, add a Menu:

```
Section(header: Text("Menu")) {  
    Menu("Actions") {  
        Button("Set TextEditor text to  
'magic') {  
            someText = "magic"  
        }  
        Button("Turn first picker green") {  
            color = Color.green  
        }  
    }  
    Menu("Actions") {  
        Button("Set TextEditor text to  
'magic') {  
            someText = "magic"  
        }  
        Button("Turn first picker  
green") {  
            color = Color.green  
        }  
    }  
}
```

```
        }
```

```
    }
```

```
}
```

```
}
```

Finally, let's improve the style of all the content by applying a `listStyle` modifier on the `List`:

```
List {
```

```
...
```

```
}.listStyle(GroupedListStyle())
```

The resulting view app preview should look like this:

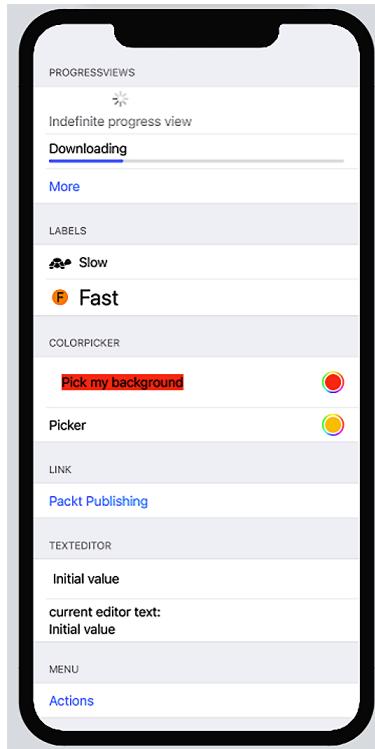


Figure 1.23 – More Views and Controls app preview

How it works...

We've implemented multiple views in this recipe. Let's look at each one and discuss how they work.

Indeterminate `ProgressView` require no parameters:

```
ProgressView("Indeterminate progress view")
ProgressView()
```

Determinate `ProgressView` components, on the other hand, require a `value` parameter that takes a state variable and displays the level of completion:

```
ProgressView("Downloading", value: progress, total:2)
```

The `total` parameter in the `ProgressView` component is optional and defaults to `1.0` if not specified.

`Label` views were mentioned earlier in the *Simple graphics using SF Symbols* recipe. Here, we introduce a second option for implementing labels where we customize the design of the label text and icon:

```
Label{
    Text ("Fast")
        .font(.title)
}
icon: {
    Circle()
        .fill(Color.orange)
        .frame(width: 40, height: 20,
               alignment: .center)
        .overlay(Text("F"))
}
```

Let's move on to the `ColorPicker` view. Color pickers let you display a palette for users to pick colors from. We create a two-way binding using the `color` state variable so that we can store the color selected by the user:

```
ColorPicker(selection: $color ){
    Text("Pick my background")
        .background(color)
        .padding()
```

```
        }
        ColorPicker("Picker", selection:
            $secondColor )
```

Link views are used to display clickable links:

```
Link("Packt Publishing", destination: URL(string:
    "https://www.packtpub.com/")!)
}
```

Finally, the Menu view provides a convenient way of presenting a user with a list of actions to choose from and can also be nested, as seen here:

```
Menu("Actions") {
    Button("Set TextEditor text to
        'magic') {
        someText = "magic"
    }
    Button("Turn first picker green") {
        color = Color.green
    }
    Menu("Actions") {
        Button("Set TextEditor text to
            'nested magic') {
            someText = "nested magic"
        }
        Button("Turn first picker red") {
            color = Color.red
        }
    }
}
```

You can add one or more buttons to a menu, each performing a specific action. Although menus can be nested, this should be done sparingly as too much nesting may decrease usability.

2

Going Beyond the Single Component with Lists and Scroll Views

In this chapter, we'll learn how to display lists in SwiftUI. List views are like UITableViews in UIKit but are significantly simpler to use. For example, no storyboards or prototype cells are required, and we do not need to remember how many rows or columns we created. Furthermore, SwiftUI's lists are modular so that you can build more significant apps from smaller components.

This chapter will also discuss new and exciting features introduced in WWDC 2021, such as lists with editable text and searchable lists. By the end of this chapter, you will understand how to display lists of static or dynamic items, add or remove items from lists, edit lists, add sections to List views, and much more.

In this chapter, we'll be covering the following recipes:

- Using scroll views
- Creating a list of static items
- Using custom rows in a list
- Adding rows to a list
- Deleting rows from a list
- Creating an editable List view
- Moving the rows in a List view
- Adding sections to a list
- Creating editable Collections
- Creating Searchable lists

Technical requirements

The code in this chapter is based on Xcode 13 and iOS 15.

You can find the code for this book in this book's GitHub repository: <https://github.com/PacktPublishing/SwiftUI-Cookbook-2nd-Edition/tree/main/Chapter02-Lists-and-ScrollViews>.

Using scroll views

You can use SwiftUI scroll views when the content to be displayed cannot fit in its container. Scroll views create scrolling content where users can use gestures to bring new content into the section of the screen where it can be viewed. Scroll views are vertical by default but can be made to scroll horizontally or vertically.

In this recipe, we will learn how to use horizontal and vertical scroll views.

Getting ready

Let's start by creating a SwiftUI project called `WeScrollView`.

Optional: If you don't have it yet, download the **San Francisco Symbols (SF Symbols)** app here: <https://developer.apple.com/sf-symbols/>.

As we mentioned in *Chapter 1, Using the Basic SwiftUI Views and Controls*, SF Symbols is a set of over 3,200 symbols provided by Apple.

How to do it...

Let's learn how scroll views work by implementing horizontal and vertical scroll views that display SF symbols for alphabet characters A - P. Here are the steps:

1. Add an array variable to our ContentView struct that contains the letters a to p:

```
let letters =
    ["a", "b", "c", "d", "e", "f", "g", "h",
     "i", "j", "k", "l", "m", "n", "o", "p"]
```

2. Replace the original text view with a VStack, a ScrollView, and a ForEach struct:

```
var body: some View {
    VStack{
        ScrollView {
            ForEach(self.letters, id: \.self) {
                letter in
                Image(systemName: letter)
                    .font(.largeTitle)
                    .foregroundColor(Color.yellow)
                    .frame(width: 50, height: 50)
                    .background(Color.blue)
                    .symbolVariant(.circle.fill)
            }
        }
        .frame(width:50, height:200)

        ScrollView(.horizontal, showsIndicators:
            false) {
            HStack{
                ForEach(self.letters, id: \.self) {
                    name in
                    Image(systemName: name)
                        .font(.largeTitle)
                        .foregroundColor(Color.yellow)
                        .frame(width: 50, height: 50)
                        .background(Color.blue)
                }
            }
        }
    }
}
```

```
        .symbolVariant(.circle.fill)
    }
}
}
}
}
```

3. Run/resume the Xcode preview from the canvas window. It should look as follows:

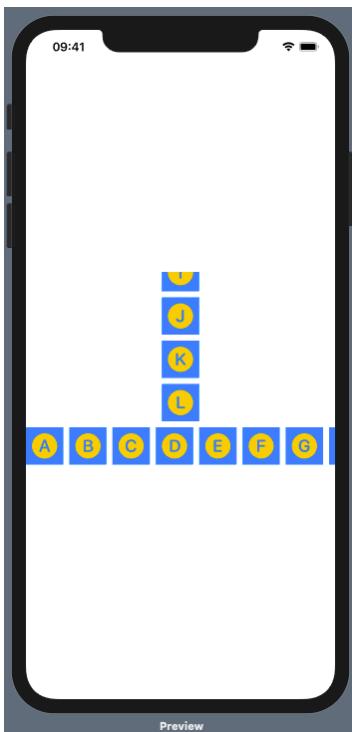


Figure 2.1 – The WeScroll app with horizontal and vertical scroll views

How it works...

By default, scroll views display items vertically. Therefore, our first scroll view displays its content along the vertical axis without requiring us to specify the axis.

In this recipe, we also introduce the `ForEach` structure, which computes views on-demand based on an underlying collection of identified data. In this case, the `ForEach` structure iterates over a static array of alphabet characters and displays the `SF Symbols` of the said characters.

We provided two arguments to the `ForEach` structure: the collection we want to iterate over and an `id`. This `id` helps us distinguish between the items in the collection and should be unique. Using `\.self` as `id`, we indicated that the alphabet characters we are using are unique and will not be repeated in this `List`. We used unique items because SwiftUI expects each row to be uniquely identifiable and will not run as expected otherwise.

You can use the `ForEach` structure without specifying the `id` argument if your collection conforms to the `Identifiable` protocol.

Moving on to the second scroll view, it uses two arguments: `axis` and `showIndicators`. The `.horizontal` axis's enum indicates we want the content to scroll horizontally, while the `.showIndicators: false` argument prevents the scrollbar indicator from appearing in the view.

See also

Apple's documentation on scroll views: <https://developer.apple.com/documentation/swiftui/scrollview>

Creating a list of static items

`List` views are like scroll views in that they are used to display a collection of items. However, `List` views are better for dealing with larger datasets because they do not load the entirety of the datasets in memory.

In this recipe, we will create an app the uses static lists to display sample weather data for various cities.

Getting ready

Let's start by creating a new SwiftUI app called `StaticList`.

How to do it...

We'll create a struct to hold weather information and an array of several cities' weather data. We'll then use a `List` view to display all the content. The steps are as follows:

1. Open the `ContentView.swift` file and add the `WeatherInfo` struct right above the `ContentView` struct:

```
struct WeatherInfo: Identifiable {  
    var id = UUID()
```

```
    var image: String  
    var temp: Int  
    var city: String  
}
```

2. Add the `weatherData` property to the `ContentView` struct. `weatherData` contains an array of `WeatherInfo` items:

```
let weatherData: [WeatherInfo] = [  
    WeatherInfo(image: "snow", temp: 5, city:"New  
        York"),  
    WeatherInfo(image: "cloud", temp:5, city:"Kansas  
        City"),  
    WeatherInfo(image: "sun.max", temp: 80, city:"San  
        Francisco"),  
    WeatherInfo(image: "snow", temp: 5,  
        city:"Chicago"),  
    WeatherInfo(image: "cloud.rain", temp: 49,  
        city:"Washington DC"),  
    WeatherInfo(image: "cloud.heavyrain", temp: 60,  
        city:"Seattle"),  
    WeatherInfo(image: "sun.min", temp: 75,  
        city:"Baltimore"),  
    WeatherInfo(image: "sun.dust", temp: 65,  
        city:"Austin"),  
    WeatherInfo(image: "sunset", temp: 78,  
        city:"Houston"),  
    WeatherInfo(image: "moon", temp: 80,  
        city:"Boston"),  
    WeatherInfo(image: "moon.circle", temp: 45,  
        city:"denver"),  
    WeatherInfo(image: "cloud.snow", temp: 8,  
        city:"Philadelphia"),  
    WeatherInfo(image: "cloud.hail", temp: 5,  
        city:"Memphis"),  
    WeatherInfo(image: "cloud.sleet", temp:5,  
        city:"Nashville")]
```

```
WeatherInfo(image: "sun.max", temp: 80, city:"San
Francisco"),
WeatherInfo(image: "cloud.sun", temp: 5,
city:"Atlanta"),
WeatherInfo(image: "wind", temp: 88, city:"Las
Vegas"),
WeatherInfo(image: "cloud.rain", temp: 60,
city:"Phoenix"),
]
```

3. Add the `List` view to the `ContentView` body and use the `ForEach` structure to iterate over our `weatherData` collection. Add some font and padding modifiers to improve the styling too:

```
List {
    ForEach(self.weatherData){ weather in
        HStack {
            Image(systemName: weather.image)
                .frame(width: 50, alignment:
                    .leading)
            Text("\(weather.temp) °F")
                .frame(width: 80, alignment:
                    .leading)
            Text(weather.city)
        }
        .font(.system(size: 25))
        .padding()
    }
}
```

The resulting preview should look as follows:

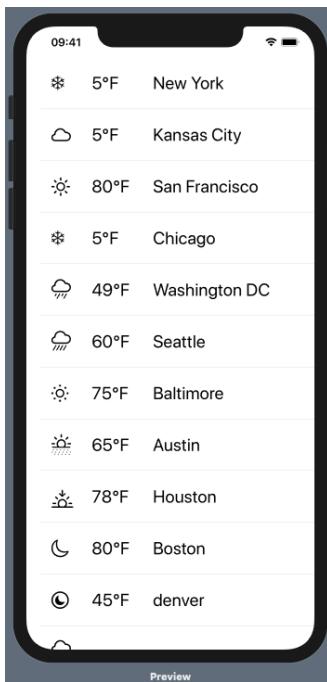


Figure 2.2 – Implementing static lists

How it works...

First, we created the `WeatherInfo` struct, which contains properties we'd like to use, such as images, temperature (`temperate`), and `city`. Notice that the `WeatherInfo` struct implements the `Identifiable` protocol. Making the struct conform to the `Identifiable` protocol allows us to use the data in a `ForEach` structure without specifying an `id` parameter. To conform to the `Identifiable` protocol, we added a unique property to our struct called `id`, a property whose value is generated by the `UUID()` function.

The basic form of a static list is composed of a `List` view and some other views, as shown here:

```
List {  
    Text("Element one")  
    Text("Element two")  
}
```

In this recipe, we went a step further and used the `ForEach` struct to iterate through an array of identifiable elements stored in the `weatherData` variable. We wanted to display the data in each list item horizontally, so we displayed the contents in an `HStack`. Our image, temperature, and city are displayed using image and text views.

The weather image names are SF Symbol variants, so using them with an `Image` view `systemName` parameter displays the corresponding SF Symbol. You can read more about SF Symbols in *Chapter 1, Using the Basic SwiftUI Views and Controls*.

Using custom rows in a list

The number of lines of code required to display items in a `List` view row could vary from one to several lines of code. Repeating the code several times or in several places increases the chance of an error occurring and potentially becomes very cumbersome to maintain. One change would require updating the code in several different locations or files.

A custom list row can be used to solve this problem. This custom row can be written once and used in several places, thereby improving maintainability and encouraging reuse.

Let's find out how to create custom list rows.

Getting ready

Let's start by creating a new SwiftUI app named `CustomRows`.

How to do it...

We will reorganize the code in our static lists to make it more modular. We'll create a separate file to hold the `WeatherInfo` struct, a separate SwiftUI file for the custom view, `WeatherRow`, and finally, we'll implement the components in the `ContentView.swift` file. The steps are as follows:

1. Create a new Swift file called `WeatherInfo` by going to **File | New | File | Swift File** (or by using the *Command* (⌘) + *N* keys).
2. Create a `WeatherInfo` struct within the newly created file:

```
struct WeatherInfo: Identifiable {  
    var id = UUID()  
    var image: String  
    var temp: Int  
    var city: String  
}
```

3. Also, add a `weatherData` variable that holds an array of `WeatherInfo`:

```
let weatherData: [WeatherInfo] = [
    WeatherInfo(image: "snow", temp: 5, city:"New
        York"),
    WeatherInfo(image: "cloud", temp:5, city:"Kansas
        City"),
    WeatherInfo(image: "sun.max", temp: 80, city:"San
        Francisco"),
    WeatherInfo(image: "snow", temp: 5,
        city:"Chicago"),
    WeatherInfo(image: "cloud.rain", temp: 49,
        city:"Washington DC"),
    WeatherInfo(image: "cloud.heavyrain", temp: 60,
        city:"Seattle"),
    WeatherInfo(image: "sun.min", temp: 75,
        city:"Baltimore"),
    WeatherInfo(image: "sun.dust", temp: 65,
        city:"Austin"),
    WeatherInfo(image: "sunset", temp: 78,
        city:"Houston"),
    WeatherInfo(image: "moon", temp: 80,
        city:"Boston"),
    WeatherInfo(image: "moon.circle", temp: 45,
        city:"denver"),
    WeatherInfo(image: "cloud.snow", temp: 8,
        city:"Philadelphia"),
    WeatherInfo(image: "cloud.hail", temp: 5,
        city:"Memphis"),
    WeatherInfo(image: "cloud.sleet", temp:5,
        city:"Nashville"),
    WeatherInfo(image: "sun.max", temp: 80, city:"San
        Francisco"),
    WeatherInfo(image: "cloud.sun", temp: 5,
        city:"Atlanta"),
    WeatherInfo(image: "wind", temp: 88, city:"Las
        Vegas"),
```

```
    WeatherInfo(image: "cloud.rain", temp: 60,  
    city:"Phoenix"),  
]
```

4. Create a new SwiftUI file by selecting **File | New | File | SwiftUI View** from the Xcode menu or by using the *Command (⌘) + N* key combination. Name the file `WeatherRow`.
5. Add the following weather row design to our new SwiftUI view:

```
struct WeatherRow: View {  
    var weather: WeatherInfo  
    var body: some View {  
        HStack {  
            Image(systemName: weather.image)  
                .frame(width: 50, alignment: .leading)  
            Text("\(weather.temp) °F")  
                .frame(width: 80, alignment: .leading)  
            Text(weather.city)  
        }  
        .font(.system(size: 25))  
        .padding()  
    }  
}
```

6. To preview or update the row design, add a sample `WeatherInfo` instance to the `WeatherRow_Previews` function:

```
struct WeatherRow_Previews: PreviewProvider {  
    static var previews: some View {  
        WeatherRow(weather: WeatherInfo(image: "snow",  
            temp: 5, city:"New York"))  
    }  
}
```

The resulting `WeatherRow.swift` canvas preview should look as follows:



Figure 2.3 – WeatherRow row preview

7. Open the `ContentView.swift` file and create a list to display data using the `WeatherRow` component:

```
struct ContentView: View {  
    var body: some View {  
        List{  
            ForEach(weatherData){weather in  
                WeatherRow(weather: weather)  
            }  
        }  
    }  
}
```

The resulting canvas preview should look as follows:

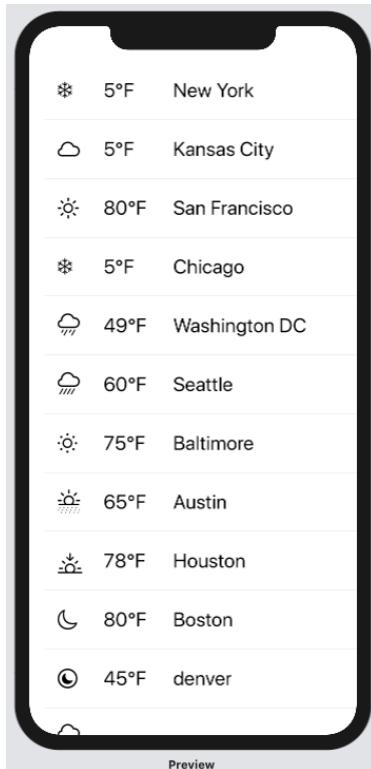


Figure 2.4 – CustomRow App preview

Run the app on a device or run a live preview to scroll through and test the app's functionality.

How it works...

`WeatherInfo.swift` is the model file containing a blueprint of what we want each instance of our `weatherInfo` struct to contain. We also instantiated an array of the `WeatherInfo` struct, `weatherData`, that can be used in other parts of the project previewing and testing areas as we build.

The `WeatherRow` SwiftUI file is our focus for this recipe. By using this file, we can extract the design of a list row into a separate file and reuse the design in other sections of our project. We added a `weather` property to our `WeatherRow` that will hold the `WeatherInfo` arguments that are passed to our `WeatherRow` view.

As in the previous recipe, we want the content of each row to be displayed horizontally next to each other, so we enclosed the components related to our `weather` variable in an `HStack`.

Important Note

The `weatherData` array is only necessary during development and should be removed before deployment if such data is obtained at runtime through API calls.

Adding rows to a list

The most common actions users might want to be able to perform on a list include adding, editing, or deleting items.

In this recipe, we'll go over the process of implementing those actions on a SwiftUI list.

Getting ready

Create a new SwiftUI project and call it `ListRowAdd`.

How to do it...

Let's create a list with a button at the top that can be used to add new rows to the list. The steps are as follows:

1. Create a state variable in the `ContentView` struct that holds an array of numbers:

```
@State var numbers = [1,2,3,4]
```

2. Add a `NavigationView` struct and a `List` view to the `ContentView` struct's body:

```
NavigationView{  
    List{  
        ForEach(self.numbers, id:\.self){  
            number in  
            Text("\(number)")  
        }  
    }  
}
```

3. Add a `.navigationBarTitle` modifier to the list with a title:

```
.navigationBarTitle("Number List", displayMode:  
    .inline)
```

4. Add a `navigationBarItems` modifier to the list with a function to trigger an element being added to the row:

```
.navigationBarItems(trailing: Button("Add", action:  
    addItemToRow) )
```

5. Implement the `addItemToRow` function and place it immediately after the `body` view's closing brace:

```
private func addItemToRow() {  
    self.numbers.append(Int.random(in: 5 ..< 100))  
}
```

The preview should look as follows:

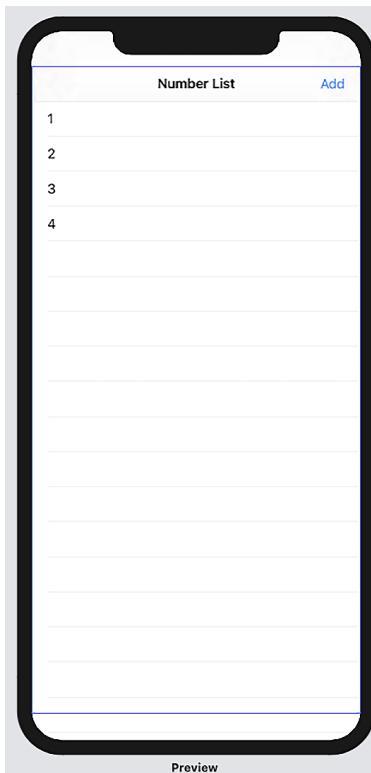


Figure 2.5 – ListRowAdd preview

You can now run the preview and click the **Add** button to add new items to the list.

How it works...

Our state variable, `numbers`, holds an array of numbers. We made it a state variable so that the view that's created by our `ForEach` struct gets updated each time a new number is added to the array.

The `.navigationBarTitle ("Number List," displayMode: .inline)` modifier adds a title to the top of the list and within the standard bounds of the navigation bar. The display mode is optional, so you could remove it to display the title more prominently. Other display modes include `automatic`, to inherit from the previous navigation item, and `large`, to display the title within an expanded navigation bar.

The `.navigationBarItems (...)` modifier adds a button to the trailing end of the navigation section. The button calls the `addItemToRow` function when clicked.

Finally, the `addItemToRow` function generates a random number between 0-99 and appends it to the `numbers` array. The view gets automatically updated since the `numbers` variable is a state variable and a change in its state triggers a view refresh.

Important Note

In our list's `ForEach` struct, we used `\.self` as our `id` parameter. However, we may end up with duplicate numbers in our list as we generate more items. Identifiers should be unique, so using values that could be duplicated may lead to unexpected behaviors. Remember to ONLY use unique identifiers for apps meant to be deployed to users.

Deleting rows from a list

So far, we've learned how to add new rows to a list. Now, let's find out how to use a swipe motion to delete items one at a time.

Getting ready

Create a new SwiftUI app called `ListRowDelete`.

How to do it...

We will create a list of items and use the list view's `onDelete` modifier to delete rows. The steps are as follows:

1. Add a state variable to the `ContentView` struct called `countries` and initialize it with an array of country names:

```
@State var countries = ["USA", "Canada",
    "England", "Cameroon", "South Africa", "Mexico" ,
    "Japan", "South Korea"]
```

2. Within the `body` variable, add a `NavigationView` and a `List` view that displays our array of countries. Also, include the `onDelete` modifier at the end of the `ForEach` structure:

```
NavigationView{
    List {
        ForEach(countries, id: \.self) {
            country in
            Text(country)
        }
        .onDelete(perform: self.deleteItem)
    }
    .navigationBarTitle("Countries",
        displayMode: .inline)
}
```

3. Below the `body` variable's closing brace, add the `deleteItem` function:

```
private func deleteItem(at indexSet: IndexSet) {
    self.countries.remove(atOffsets: indexSet)
}
```

The resulting preview should look as follows:

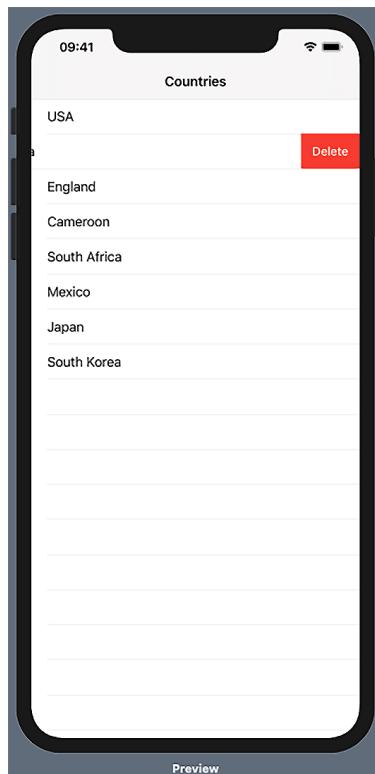


Figure 2.6 – ListRowDelete in action

Run the canvas preview and swipe right to left on a list row. The **Delete** button will appear and can be clicked to delete an item from the list.

How it works...

In this recipe, we introduced the `.onDelete` modifier, whose `perform` parameter takes a function that will be executed when clicked. In this case, deleting an item triggers the execution of our `deleteItem` function.

The `deleteItem` function takes a single parameter, `IndexSet`, which is the index of the row to be deleted. The `onDelete` modifier automatically passes the index of the item to be deleted.

There's more...

Deleting an item from a `List` view can also be performed by embedding the list navigation view and adding an `EditButton` component.

Creating an editable List view

Adding an edit button to a List view is very similar to adding a delete button, as seen in the previous recipe. An edit button offers the user the option to quickly delete items by clicking a minus sign to the left of each list row.

Getting ready

Create a new SwiftUI project named ListRowEdit.

How to do it...

The steps for adding an edit button to a List view are similar to the steps we used when adding a delete button. The process is as follows:

1. Replace the ContentView struct with the following content from the DeleteRowFromList app:

```
struct ContentView: View {  
    @State var countries = ["USA", "Canada",  
    "England", "Cameroon", "South Africa", "Mexico" ,  
    "Japan", "South Korea"]  
    var body: some View {  
        NavigationView{  
            List {  
                ForEach(countries, id: \.self) {  
                    country in  
                    Text(country)  
                }  
                .onDelete(perform: self.deleteItem)  
            }  
            .navigationBarTitle("Countries",  
                displayMode: .inline)  
        }  
    }  
    private func deleteItem(at indexSet: IndexSet){  
        self.countries.remove(atOffsets: indexSet)  
    }  
}
```

2. Add a `.navigationBarItems(trailing: EditButton())` modifier to the `List` view, just below the `.navigationBarTitle` modifier.
3. Run the preview and click on the **Edit** button at the top-right corner of the emulated device's screen. A minus (-) sign in a red circle will appear to the left of each list item, as shown in the following preview:

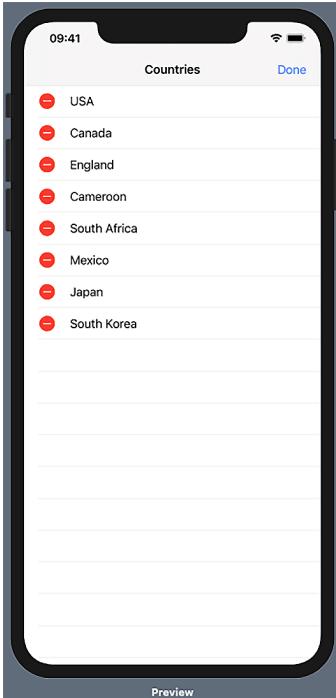


Figure 2.7 – ListRowEdit app preview during execution

Click on the circle to the left of any list item to delete it.

How it works...

The `.navigationBarItems(trailing: EditButton())` modifier adds an **Edit** button to the top-right corner of the display. Once clicked, it triggers the appearance of a minus sign to the left of each item in the modified List. Clicking on the minus sign executes the function in our `.onDelete` modifier and deletes the related item from the row.

There's more...

To display the **Edit** button on the left-hand side of the navigation bar, change the modifier to `.navigationBarItems(leading: EditButton())`.

Moving the rows in a List view

In this recipe, we'll create an app that implements a List view that allows users to move and reorganize rows.

Getting ready

Create a new SwiftUI project named `MovingListRows`.

How to do it...

To make the List view rows movable, we'll add a modifier to the List view's `ForEach` struct, and then we'll embed the List view in a navigation view that displays a title and an edit button. The steps are as follows:

1. Add a `@State` variable to the `ContentView` struct that holds an array of countries:

```
@State var countries = ["USA", "Canada",
    "England", "Cameroon", "South Africa", "Mexico" ,
    "Japan", "South Korea"]
```

2. Replace the body variable's text view with a `NavigationView`, a `List`, and modifiers for navigating. Also, notice that the `.on Move` modifier is applied to the `ForEach` struct:

```
NavigationView{
    List {
        ForEach(countries, id: \.self) {
            country in
            Text(country)
        }
        .on Move(perform: moveRow)
    }
    .navigationBarTitle("Countries",
        displayMode: .inline)
    .navigationBarItems(trailing:
        EditButton())
}
```

3. Now, let's add the function that gets called when we try to move a row. The `moveRow` function should be located directly below the closing brace of the `body` view:

```
private func moveRow(source: IndexSet,  
                     destination: Int){  
    countries.move(fromOffsets: source, toOffset:  
                    destination)  
}
```

Let's run our application in the canvas or a simulator and click on the edit button. If everything was done right, the preview should look as follows. Now, click and drag on the hamburger menu symbol at the right of each country to move it to a new location:

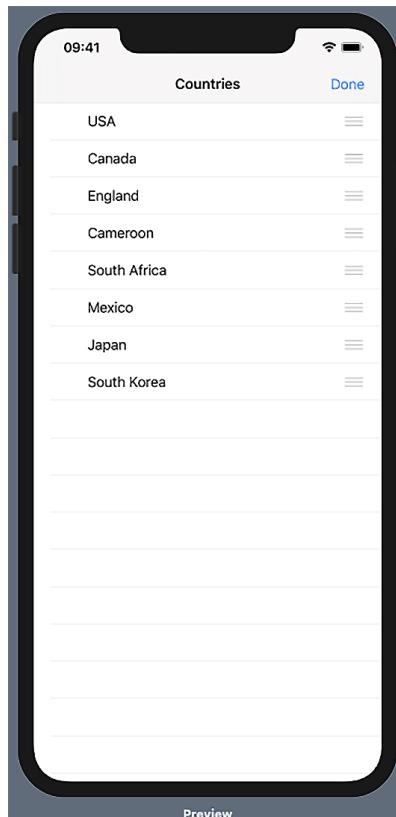


Figure 2.8 – MovingListRows

How it works...

To move list rows, you need to wrap the list in a `NavigationView`, add the `.onMove(perform:)` modifier to the `ForEach` struct, and add a `.navigationBarItems(...)` modifier to the list. The `onMove` modifier calls the `moveRow` function when clicked, while `.navigationBarItems` displays the button that starts the "move mode," where list row items become movable.

The `moveRow(source: Int, destination: Int)` function takes two parameters, `source` and `IndexSet`, which represent the current index of the item to be moved and its destination index, respectively.

Adding sections to a list

In this recipe, we will create an app that implements a static list with sections. The app will display a list of countries grouped by continent.

Getting ready

Let's start by creating a new SwiftUI app in Xcode named `ListWithSections`.

How to do it...

We will add a `Section` view to our `List` to separate groups of items by section titles. Proceed as follows:

1. (Optional) Open the `ContentView.swift` file and replace the `Text` view with a `NavigationView`. Wrapping the `List` in a `NavigationView` allows us to add a title and navigation items to the view:

```
NavigationView{  
}
```

2. Add a list and section to `NavigationView` (or body view if you skipped optional Step 1). Also, add a `listStyle` and `navigationBarTitle` modifier:

```
List {  
    Section(header: Text("North America")) {  
        Text("USA")  
        Text("Canada")  
        Text("Mexico")  
        Text("Panama")  
    }  
}
```

```
        Text ("Anguilla")
    }
}
.listStyle(.grouped)
.navigationBarTitle("Continents and Countries",
    displayMode: .inline)
```

3. Below the initial Section, add more sections representing countries in various continents:

```
List {
...
Section(header: Text("Africa")) {
    Text("Nigeria")
    Text("Ghana")
    Text("Kenya")
    Text("Senegal")
}
Section(header: Text("Europe")) {
    Text("Spain")
    Text("France")
    Text("Sweden")
    Text("Finland")
    Text("UK")
}
}
```

Your canvas preview should resemble the following:

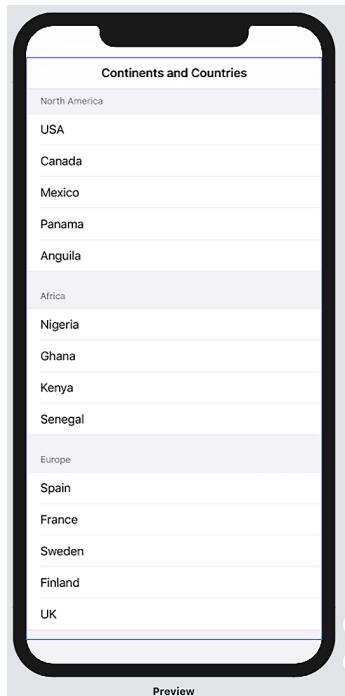


Figure 2.9 – ListWithSections preview

Looking at the preview, you can see the continents where each country is located by reading the section titles.

How it works...

SwiftUI's `Section` views are used to separate items into groups. In this recipe, we used `Section` views to visually group countries by their continents. A `Section` view can be used with a header, as shown in this recipe, or without a header, as follows:

```
Section {  
    Text("Spain")  
    Text("France")  
    Text("Sweden")  
    Text("Finland")  
    Text("UK")  
}
```

You can change section styles by using the `listStyle()` modifier with the `.grouped` style.

Creating editable Collections

Editing lists has always been possible in SwiftUI but before WWDC 2021 and SwiftUI 3, doing so was very inefficient because SwiftUI did not support binding to Collections. Let's use bindings on a collection and discuss how and why it works better now.

Getting ready

Create a new SwiftUI project and name it `EditablesListsFields`.

How to do it...

Let's create a simple to-do list app with a few editable items. The steps are as follows:

1. Add a `TodoItem` struct below the `import SwiftUI` line:

```
struct TodoItem: Identifiable {
    let id = UUID()
    var title: String
    init(_ someTitle:String) {
        title = someTitle
    }
}
```

2. In our `ContentView` struct, let's add a collection of `TodoItem` instances:

```
@State var todos = [
    TodoItem("Eat"),
    TodoItem("Sleep"),
    TodoItem("Code")
]
```

3. Replace the `Text` view in the body with a `List` and a `TextField` view that displays the collection of `todo` items:

```
var body: some View {
    List($todos) { $todo in
        TextField("Number", text: $todo.title)
    }
}
```

Run the preview in canvas. You should be able to edit the text in each row, as shown in the following screenshot:

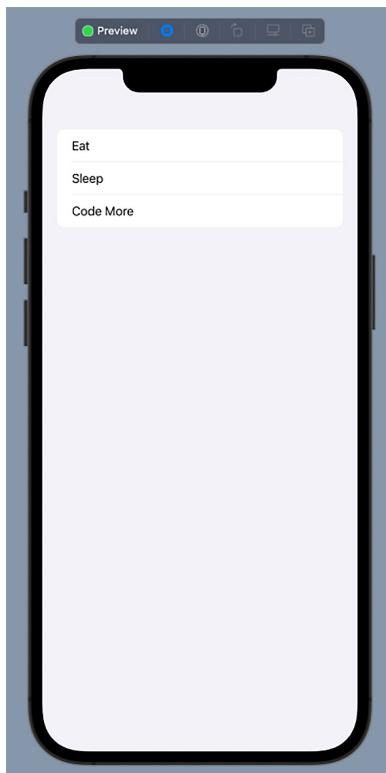


Figure 2.10 – Editable Collections preview

Click on any of the other rows and edit it to your heart's content.

How it works...

Let's start by looking at how editable lists were handled before SwiftUI 3. Before SwiftUI 3, the code for an editable list of items would use list indices to create bindings to a collection, as follows:

```
List(0..<todos.count) { index in
    TextField("Todo", text: $todos[index].title)
}
```

Not only was such code slow, but editing a single item caused SwiftUI to re-render the entire `List` of elements, leading to flickering and slow UI updates.

With SwiftUI 3, we can pass a binding to a collection of elements, and SwiftUI will internally handle binding to the current element specified in the closure. Since the whole of our collection conforms to the `Identifiable` protocol, each of our list items can be uniquely identified by its `id` parameter; therefore, adding or removing items from the list does not change list item indices and does not cause the entire list to be re-rendered.

Using Searchable lists

`List` views can hold from one to an uncountable number of items. As the number of items in a list increases, it is usually helpful to provide users with the ability to search through the list for a specific item without having to scroll through the whole list.

In this recipe, we'll introduce the `.searchable()` modifier and discuss how it can be used to search through items in a list.

Getting ready

Create a new SwiftUI project and name it `SearchableLists`.

The `searchable` modifier is only available in iOS 15+. In your build settings, make sure that your **iOS Deployment Target** is set to iOS 15. Use the following steps to change the deployment target:

1. From the navigation pane, select the project's name (**SearchableLists**).
2. Select **Build settings**.
3. Under **Deployment**, select **iOS Deployment Target**.
4. Select **iOS 15.0** from the popup menu.

These steps are shown in the following screenshot:

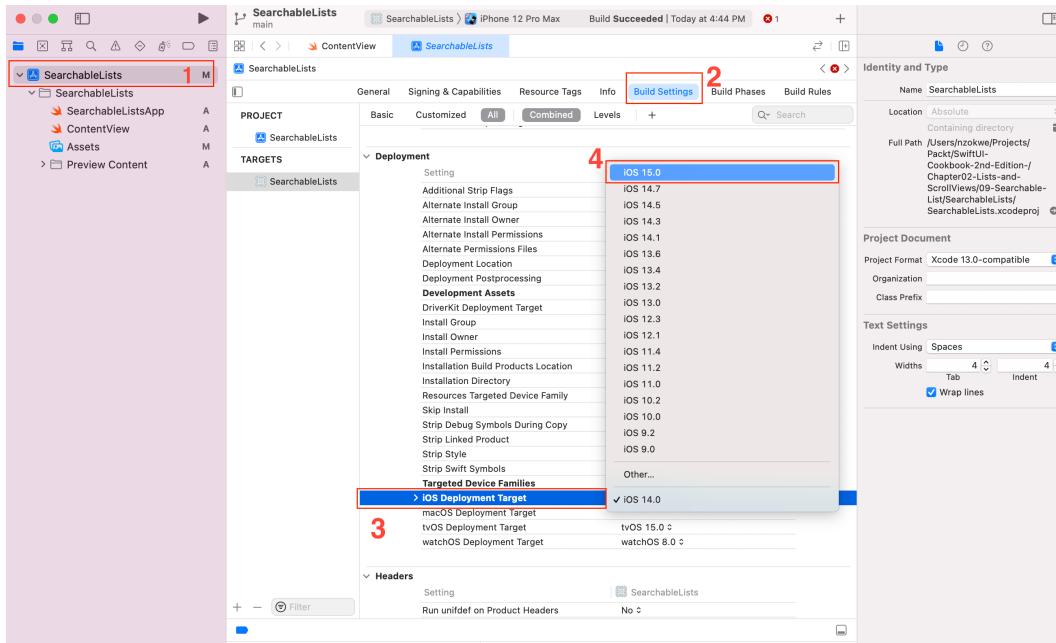


Figure 2.11 – Setting the iOS Deployment Target

How to do it...

Let's create an app to search through possible messages between a parent and their child. The steps are as follows:

1. Before the ContentView struct's body, add a State variable to hold the search text and sample messages:

```
@State private var searchText = ""

let messages = [
    "Dad, can you lend me money?",
    "Nada. Does money grow on trees?",
    "What is money made out of?",
    "Paper",
    "Where does paper come from?",
    "Huh.....",
]
```

2. Add a `NavigationView`, a `List` to display the search results, a `navigationBarTitle` modifier, and a `.searchable` modifier:

```
var body: some View {
    NavigationView {
        List{
            ForEach(searchResults, id: \.self){
                msg in
                Text(msg)
            }
        }
        .searchable(text: $searchText)
        .navigationBarTitle("Order number")
    }
}
```

3. Below the `body` variable, add the `searchResults` computed property, which returns an array of elements representing the result of the search:

```
var searchResults: [String] {
    if searchText.isEmpty {
        return messages
    }else{
        return messages.filter{
            $0.lowercased().contains
            (searchText.lowercased())
        }
    }
}
```

Run the app in canvas mode. The resulting live preview should look as follows:

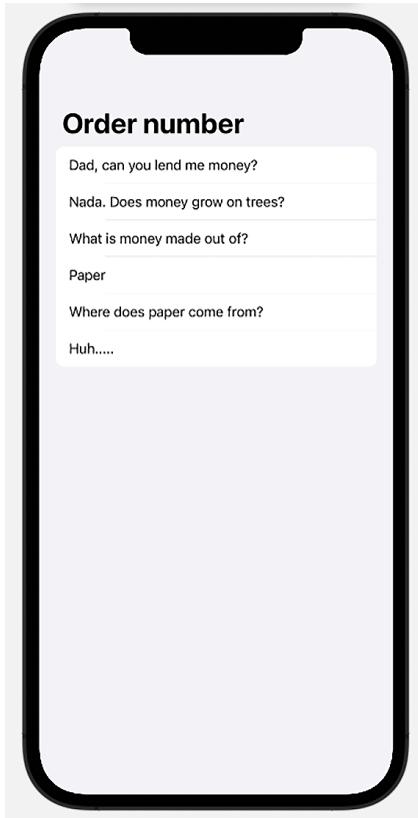


Figure 2.12 – Searchable List live preview

Now, type something within the search field and watch how the content is filtered to match the result of the search text that was entered.

How it works...

The `searchText` state variable holds the value that's being searched for and is passed as an argument to the `.searchable` modifier. Each time the value of `searchText` changes, the computed property, `searchResults`, gets calculated. Finally, the value of `searchResults` is used in the `ForEach` struct to display a filtered list of items based on the search text.

There's more...

You can provide autocomplete information by adding a closure to the `.searchable` modifier, as shown here:

```
.searchable(text: $searchText) {  
    ForEach(searchResults, id: \.self) { result in  
        '    Text((result)) .searchCompletion(result)  
    }  
}
```

The autocomplete feature provides the user with possible suggestions that match the search string they've entered so far. Clicking on one of the suggestions auto-fills the rest of the search text area and displays the results from the search.

3

Exploring Advanced Components

In the previous chapters, we learned about the essential components of SwiftUI and how they can be put together to design great-looking apps. In this chapter, we'll discuss how to display large datasets efficiently by making use of SwiftUI's lazy stacks and lazy grids.

Lazy loaded views only load a subset of their content, the subset of items currently being displayed, and the content immediately preceding or succeeding the displayed subset.

We'll also learn how to present hierarchical data in an expandable list with sections. Finally, we'll introduce one of my favorite iOS features that's only available in SwiftUI Widgets. By the end of this chapter, you will be able to build apps that present large datasets and use collapsible lists and widgets.

In this chapter, we will cover the following recipes:

- Using `LazyHStack` and `LazyVStack`
- Displaying tabular content with `LazyHGrid` and `LazyVGrid`
- Scrolling programmatically with `ScrollViewReader`
- Displaying hierarchical content in expanding lists
- Using disclosure groups to hide and show content
- Creating SwiftUI widgets

Technical requirements

The code in this chapter is based on Xcode 13.

You can find the code for this chapter in this book's GitHub repository at <https://github.com/PacktPublishing/SwiftUI-Cookbook-2nd-Edition/tree/main/Chapter03-Advanced-Components>.

Using LazyHStack and LazyVStack

SwiftUI 2.0 introduced the `LazyHStack` and `LazyVStack` views. They can be used similar to the regular `HStack` and `VStack` views (discussed in *Chapter 1, Using the Basic SwiftUI Views and Controls*) but offer an extra advantage of lazy loading its content. The list's content is loaded just before it becomes visible on the device's screen, allowing the user to seamlessly scroll through large datasets with no perceptible UI lag or long load times. Let's create an app to see how this works in practice.

Getting ready

Create a new SwiftUI app called `LazyStacks`.

How to do it...

We will create an app that applies both the `LazyHStack` and `LazyVStack` views. We will ensure that each list row prints out some text to the console before being loaded. That way, we'll know what is being loaded and when. The steps are as follows:

1. Create a `ListRow` view that has two properties: an `id` and a `type`. The `ListRow` view will print out some information when it is being initialized. Place this `ListRow` struct just above our `ContentView` view:

```
struct ListRow: View {  
    let id: Int  
    let type: String  
    init(id: Int, type: String) {  
        print("Loading \(type) item \(id)")  
        self.id = id  
        self.type = type  
    }  
}
```

```
var body: some View {
    Text("\(type) \(id)").padding()
}
```

2. Add a VStack, a ScrollView, and a LazyHStack to the body view. Use a .frame modifier to limit the view's height:

```
var body: some View {
    VStack{
        ScrollView(.horizontal) {
            LazyHStack {
                ForEach(1...10000, id:\.self) {
                    item in
                    ListRow(id: item, type:
                        "Horizontal")
                }
            }
        }.frame(height: 100, alignment: .center)
    }
}
```

3. Add a ScrollView to VStack, just below the .frame modifier:

```
ScrollView {
    LazyVStack {
        ForEach(1...10000, id:\.self){ item in
            ListRow(id: item, type: "Vertical")
        }
    }
}
```

- Now, let's observe lazy loading in action. First, if the Xcode debug area is not visible, click on **View | Debug Area | Show Debug Area**:

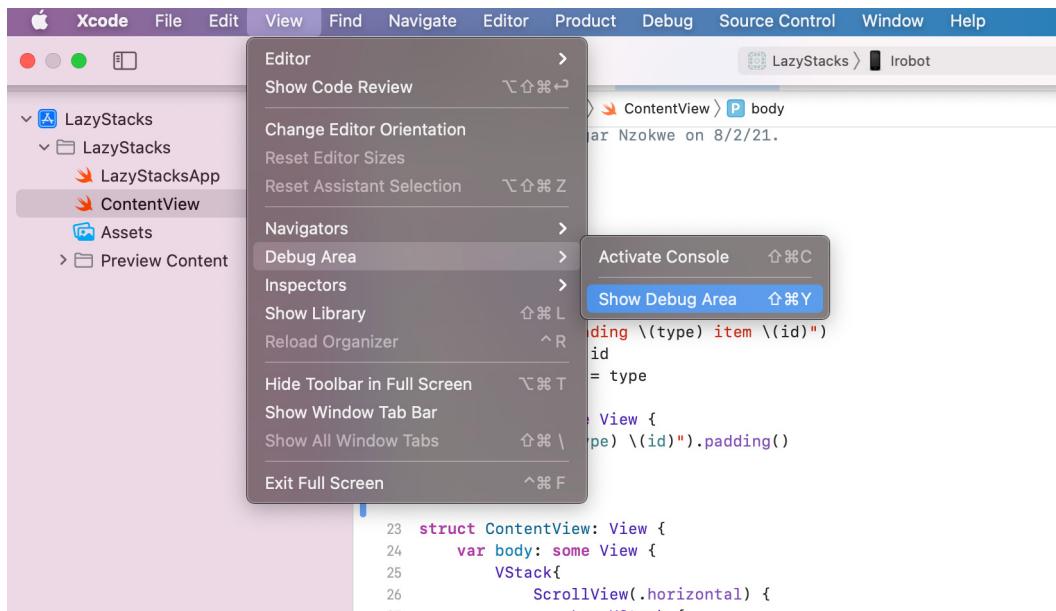


Figure 3.1 – Show Debug Area

- From the Xcode toolbar, select the simulator that you want to use to run the app:



Figure 3.2 – Xcode toolbar with iPhone 12 Pro Max selected

- Click the *play* button above the simulator in the canvas section of Xcode. The resulting view should be as follows:

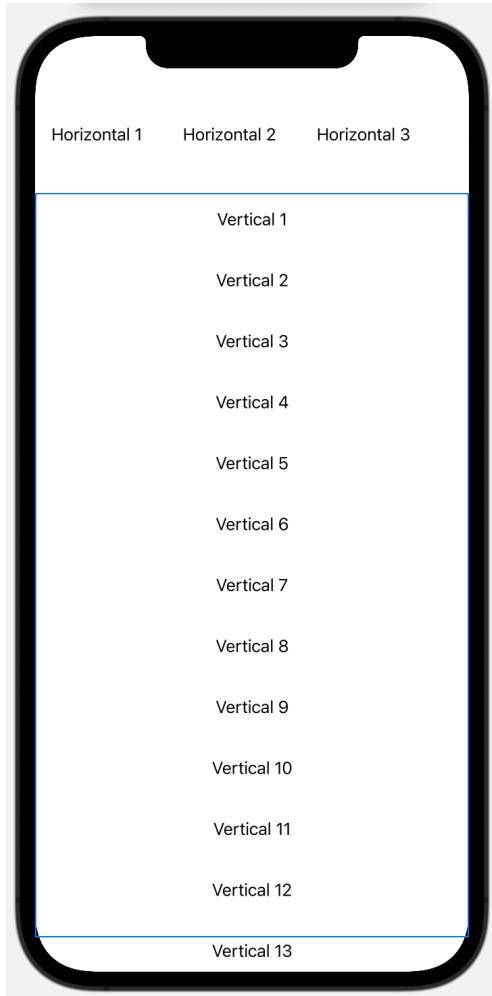


Figure 3.3 – LazyStacks app running in canvas preview

7. Scroll through the items in `LazyHStack`, which is located at the top, and observe the `print` statements in the debug area. You'll notice that each item gets initialized just before it is displayed on the screen.
8. Scroll through the items in `LazyVStack` and observe a similar result to `LazyHStack` in the previous step.

How it works...

As we mentioned in the introduction to this recipe, the main advantage of the `LazyHStack` and `LazyVStack` views over the regular `HStack` and `VStack` views is that the former load items before displaying them while the latter load all items at once during runtime.

We created a `ListRow` view whose `init` function prints the initialized item to observe lazy loading in action. The `ListRow` view's `body` property is a view that displays a `Text` view containing the `type` row and the `id` property of the initialized struct.

In the `ContentView` struct, we replaced the initial `Text` view in the body with a `VStack`. This way, we can vertically display multiple views. Then, we added two `ScrollView`. In the first, we implemented `LazyHStack`, while in the second, we implemented `LazyVStack`.

If you launch the app on a device or an emulator, scroll through the numbers while watching the debug area print statements. You'll notice that only a subset of the `ListRow` views are initialized as you scroll – that's lazy loading at work. Only loading the items that will soon be displayed on the screen will be initialized. That way, the app never spends excessive time loading all the data while the user interface stays empty.

There's more...

Try implementing this recipe with a regular `List` view and observe the performance difference. You'll notice significantly slower loading times because the app initializes all the rows before displaying the list.

Displaying tabular content with `LazyHGrid` and `LazyVGrid`

Like lazy stacks, lazy grids also use lazy loading to display a collection of items. They only initialize only subset of items that will soon be displayed on the screen when the user scrolls. You can display content from top to bottom using a `LazyVGrid` view and from left to right using a `LazyHGrid` view.

In this recipe, we'll use the `LazyVGrid` and `LazyHGrid` views to display an extensive range of numbers embedded in colorful circles.

Getting ready

Create a new SwiftUI app called `LazyGrids`.

How to do it...

We'll use a `ForEach` structure count for numbers from 0 to 999 and display them in a `LazyHGrid`, then repeat similar steps to display the numbers in a `LazyVGrid` view. The steps are as follows:

1. In the `ContentView` struct, just above the `body` variable, create an array of `GridItem` columns. The `GridItem` struct helps configure the layout of the lazy grid:

```
let columnSpec = [
    GridItem(.adaptive(minimum: 100))
]
```

2. Create an array of `GridItem` rows below the columns we just declared in the previous step:

```
let rowSpec = [
    GridItem(.flexible()),
    GridItem(.flexible()),
    GridItem(.flexible())
]
```

3. Create an array of `colors` that will be used later in this recipe to add color around each grid item:

```
let colors: [Color] = [.green, .red, .yellow,
    .blue]
```

4. Replace the initial `Text` view in the body with a `VStack`, a `ScrollView`, a `LazyVGrid`, and a `ForEach` struct that loops through numbers 1 to 999:

```
VStack {
    ScrollView {
        LazyVGrid(columns: columnSpec,
            spacing:20) {
            ForEach(1...999, id:\.self){ index
                in
                    Text("Item \(index)")
                        .padding(EdgeInsets(top: 30,
                            leading: 15,
                            bottom: 30, trailing: 15))
            }
        }
    }
}
```

```
        .background(colors[index %  
                      colors.count])  
        .clipShape(Capsule())  
    }  
}  
}  
}
```

5. Let's add a second `ScrollView` to our `VStack` and add a `LazyHGrid` that displays numbers within a similar range to `LazyVGrid`:

```
ScrollView(.horizontal) {  
    LazyHGrid(rows: rowSpec, spacing:20) {  
        ForEach(1...999, id:\.self){ index  
            in  
                Text("Item \(index)")  
                    .foregroundColor(.white)  
                    .padding(EdgeInsets(top: 30,  
                                      leading: 15, bottom: 30,  
                                      trailing: 15))  
                    .background(colors[index %  
                                      colors.count])  
                    .clipShape(Capsule())  
            }  
        }  
    }
```

If all these steps were executed successfully, the preview should look as follows:

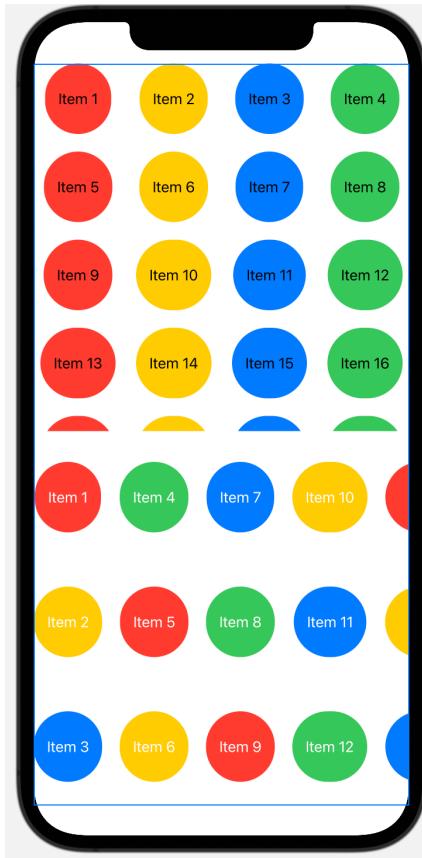


Figure 3.4 – LazyHStack and LazyVStack

Run the app in canvas preview and scroll horizontally or vertically through each grid. Scrolling stay responsive, despite displaying content from a data source with around 200 elements because the content was lazy loaded.

How it works...

In its basic form, a lazy grid can be implemented by embedding a `LazyVGrid` or `LazyHGrid` in a `ScrollView`, as follows:

```
ScrollView {  
    LazyHGrid(columns: columnsSpec) {  
        // Items to be displayed  
    }  
}
```

One of the fundamental concepts about using lazy grids involves understanding how to define the column or rows of the grid. For example, the `LazyVGrid` column variable defines an array containing a single `GridItem` component but causes four columns to be displayed, as seen in *Figure 3.4*. How is that possible? The answer lies in how `GridItem` was defined. Using `GridItem(.adaptive(minimum: 100))`, we told SwiftUI to use at least 100 units of width for each item and place as many items as possible along the same column. Thus, the number of columns changes based on whether the device is in portrait or landscape mode, or if a device with a larger screen size is being used.

If you would rather specify the number of columns in a `LazyVStack` view or rows in a `LazyHStack` view, you can use `GridItem(.flexible())`, as follows:

```
let rowSpec = [
    GridItem(.flexible()),
    GridItem(.flexible()),
    GridItem(.flexible())
]
```

Adding an array of three flexible grid items divides the available space into three equal rows for data to be displayed, with the rest of the space empty. While previewing the app in canvas preview mode, you can delete one of the grid items from `rowSpec` and observe that the number of rows in the preview also decreases.

Scrolling programmatically with ScrollViewReader

`ScrollViewReader` allows you to programmatically scroll to a different index in a list that might or might not be currently visible on the screen. For example, `ScrollViewReader` could be used to programmatically scroll to a newly added item, scroll to the most recently changed item, or scroll based on a custom trigger.

In this recipe, we will create an app that displays a list of characters from A to Q. A button at the top of the screen will programmatically scroll from the top to the end of the list when clicked. Another button at the bottom of the screen will allow us to scroll back up from the bottom to the middle of the list.

Getting ready

Create a new SwiftUI project named `ScrollViewReaders`.

How to do it...

We'll start by creating an `Identifiable` struct that has two properties: a `name` and an `id`.

Then, we'll use this `struct` to create an array of SF Symbols to be displayed on the screen. Finally, we'll implement `ScrollViewReader` and programmatically scroll to different sections of our list.

The steps are as follows:

1. Create a `struct` called `characterInfo`, just above the `ContentView` struct:

```
struct characterInfo: Identifiable {  
    var name: String  
    var id: Int  
}
```

2. In the `ContentView` struct, just before the `body` view, add an array of `characterInfo` called `charArray`. Initialize it with `characterInfo` struct representing the SF Symbols for the characters A-Q:

```
struct ContentView: View {  
    let charArray = [  
        characterInfo(name:"a.circle.fill",id:0),  
        characterInfo(name:"b.circle.fill",id:1),  
        characterInfo(name:"c.circle.fill",id:2),  
        characterInfo(name:"d.circle.fill",id:3),  
        characterInfo(name:"e.circle.fill",id:4),  
        characterInfo(name:"f.circle.fill",id:5),  
        characterInfo(name:"g.circle.fill",id:6),  
        characterInfo(name:"h.circle.fill",id:7),  
        characterInfo(name:"i.circle.fill",id:8),  
        characterInfo(name:"j.circle.fill",id:9),  
        characterInfo(name:"k.circle.fill",id:10),  
        characterInfo(name:"l.circle.fill",id:11),  
        characterInfo(name:"m.circle.fill",id:12),  
        characterInfo(name:"n.circle.fill",id:13),  
        characterInfo(name:"o.circle.fill",id:14),  
        characterInfo(name:"p.circle.fill",id:15),  
        characterInfo(name:"q.circle.fill",id:16),
```

```
    ]
    // body struct and rest of code here
}
```

3. Replace the Text view in the body struct with a ScrollView, ScrollViewReader, and a Button view, which will be used so that we can programmatically navigate to the bottom of the screen:

```
var body: some View {
    ScrollView {
        ScrollViewReader { value in
            Button("Go to letter Q") {
                value.scrollTo(16)
            }
        }
    }
}
```

4. Now, let's use a ForEach struct to iterate over charArray and display its properties using SwiftUI's Image struct:

```
var body: some View {
    ScrollView {
        ScrollViewReader { value in
            Button("Go to letter Q") {
                value.scrollTo(16)
            }
        }
        ForEach(charArray){ image in
            Image(systemName: image.name)
                .id(image.id)
                .font(.largeTitle)
                .foregroundColor(Color.yellow)
                .frame(width: 90, height: 90)
                .background(Color.blue)
                .padding()
        }
    }
}
```

```
    }

}

}
```

5. Now, let's add a button at the bottom of the screen that programmatically scrolls halfway up, to G:

```
var body: some View {
    ScrollView {
        ScrollViewReader { value in
            //ForEach struct ends here
            Button("Go to G") {
                value.scrollTo(6, anchor: .bottom)
            }
        }
    }
}
```

The app should be able to function without issues at this point.

6. Finally, let's add some styling to both the top and bottom buttons:

```
ScrollViewReader { value in
    Button("Go to letter Q") {
        value.scrollTo(16)
    }
    .padding()
    .background(Color.yellow)
    // Some of the code not shown here
    Button("Go to G") {
        value.scrollTo(6, anchor: .bottom)
    }
    .padding()
    .background(Color.yellow)
}
```

The resulting app preview should look as follows:

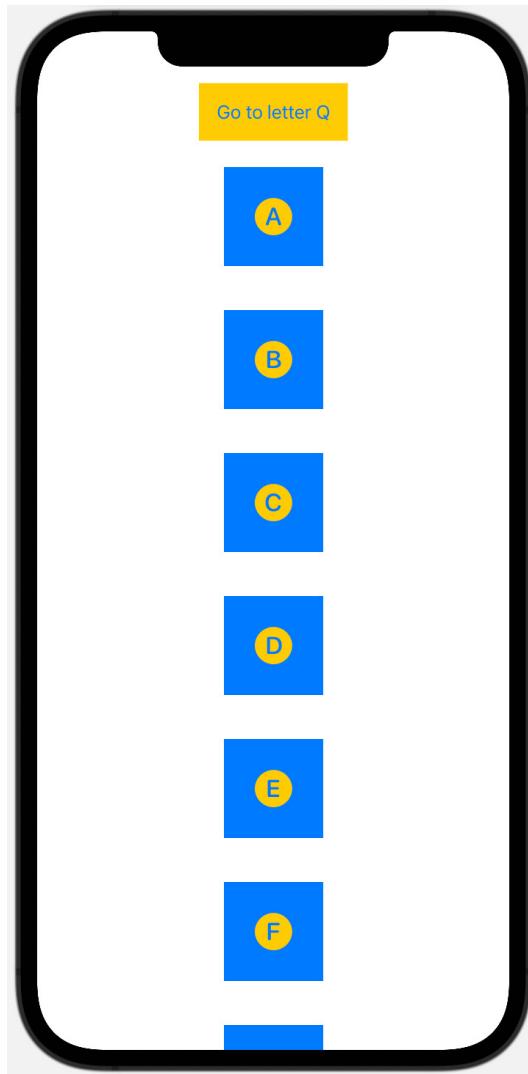


Figure 3.5 – ScrollViewReaders app preview

Run the app and tap on the **Go to letter Q** button to programmatically scroll down to the letter **Q**. Scroll up to the letter **G** by tapping on the button at the bottom of the screen.

How it works...

The `characterInfo` struct conforms to the `Identifiable` protocol and has two properties: a name and an `id`. The `id` parameter is required for `ScrollViewReader` as a reference to the location of each item. Just like a house address provides you with a destination when you're driving, the `id` parameter provides the address to each item in our `ScrollView`. That way, we can specify the destination of our scroll.

We embedded `ScrollViewReader` in a `ScrollView` so that we have access to the `scrollTo()` method. The latter can be used to programmatically scroll to the item whose index has been specified in the method:

```
ScrollViewReader { value in
    Button("Go to letter Q") {
        value.scrollTo(16)
    }
//Some of the code not shown here
}
```

The `scrollTo()` method also has an `anchor` parameter that is used to specify the position of the item we are scrolling to. For example, `scrollTo(6, anchor: .top)` causes the app to scroll until the `characterInfo` item with `id` 6, which is located at the bottom of the view.

Displaying hierarchical content in expanding lists

An expanding list helps display hierarchical content in expandable sections. This expanding ability can be achieved by creating a `struct` that holds some information and an optional array of items. Let's examine how expanding lists work by creating an app that displays the contents of a backpack.

Getting ready

Create a new SwiftUI project named `ExpandingLists`.

How to do it...

We'll start by creating a `Backpack` struct that describes the properties of the data we want to display. The backpack will conform to the `Identifiable` protocol, and each backpack will have a name, an icon, an `id`, and some content of the `Backpack` type.

A struct that represents a backpack is good for demonstrating hierarchies because, in real life, you can put a backpack in another backpack. The steps for this recipe are as follows:

1. At the top of our `ContentView.swift` file, before the `ContentView` struct, define the `Backpack` struct:

```
struct Backpack: Identifiable {
    let id = UUID()
    let name: String
    let icon: String
    var content: [Backpack]?
}
```

2. Below the `Backpack` struct, create three variables representing two different currencies and an array of currencies:

```
let dollar = Backpack(name: "Dollar", icon:
    "dollarsign.circle")
let yen = Backpack(name: "Yen", icon: "yensign.circle")
let currencies = Backpack(name: "Currencies", icon:
    "coloncurrencysign.circle", content: [dollar, yen])
```

3. Create the `pencil`, `hammer`, `paperClip`, and `glass` constants:

```
let pencil = Backpack(name: "Pencil", icon:
    "pencil.circle")
let hammer = Backpack(name: "Hammer", icon: "hammer")
let paperClip = Backpack(name: "Paperclip", icon:
    "paperclip")
let glass = Backpack(name: "Magnifying glass", icon:
    "magnifyingglass")
```

4. Create a `bin` `Backpack` constant that contains `paperClip` and `glass`. Also, create a `tools` constant that holds `pencil`, `hammer`, and the `bin` we just created:

```
let bin = Backpack(name: "Bin", icon: "arrow.up.bin",
    content: [paperClip, glass])
let tools = Backpack(name: "Tools", icon: "folder",
    content: [pencil, hammer, bin])
```

- Within the `ContentView` struct, add an instance property called `bagContents` that contains an array of size two. `bagContents` will store the currencies and tools that we mentioned in the previous step:

```
struct ContentView: View {  
    let bagContents = [currencies, tools]  
    // More code coming here  
}
```

Within the `body` variable, replace the initial `Text` view with a `List` view that displays the contents of our `bagContent` array:

```
var body: some View {  
    List(bagContents, children: \.content) { row in  
        Image(systemName: row.icon)  
        Text(row.name)  
    }  
}
```

When all the sections have been expanded, the resulting Xcode live preview should look as follows:

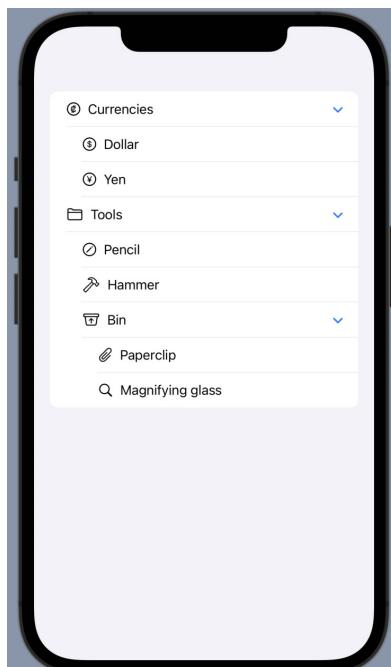


Figure 3.6 – Using expanding lists

Run Xcode live preview and have fun expanding and collapsing sections of the list.

How it works...

The `Backpack` struct was defined in such a way that you can nest elements. Its `content` property is an array of elements of the `Backpack` type.

Since the `Backpack` struct's `content` property is optional, we can create mock items to put in the backpack. These mock items are of the `Backpack` type but have no content. The `dollar` and `yen` variables represent mock items, as shown in the following code:

```
let dollar = Backpack(name: "Dollar", icon:  
    "dollarsign.circle")  
let yen = Backpack(name: "Yen", icon: "yensign.circle")
```

Our currencies can then be stored in a bag called `currencies`, thereby creating a hierarchical structure where the `currencies` variable is the parent and `dollar` and `yen` are the children, as shown in the following code:

```
let currencies = Backpack(name: "Currencies", icon:  
    "coloncurrencysign.circle", content: [dollar, yen])
```

A similar concept was used to add `paperClip` and `glass` to our `bin` variable. We also create a `tools` variable that contains the `bin` variable and a `pencil` and a `hammer`. Finally, in the body section of our project, we created a `bagContent` array that contains the `currencies` and `tools` variables. At this point, our hierarchy looks as follows:

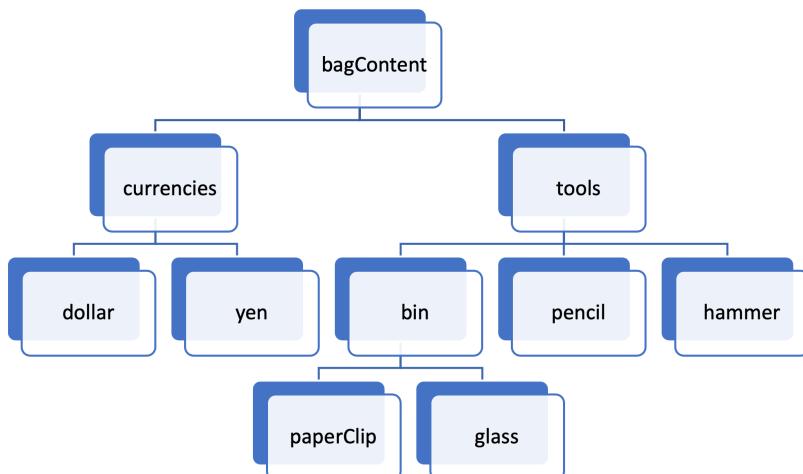


Figure 3.7 – Hierarchical view of `bagContent`

The final touch that makes a list expandable is adding the `children` parameter to our `List` view:

```
List(bagContents, children: \.content){ row in
    Image(systemName: row.icon)
    Text(row.name)
}
```

The `children` parameter of the `List` view expects an array of the same type as the struct being passed to it.

There's more...

The tree structure that we used for our expandable list could also be created in a more compact way using the following code:

```
let tools = Backpack(name: "Tools", icon: "folder",
content:
[Backpack(name: "Pencil", icon: "pencil.circle"),
Backpack(name: "Hammer", icon: "hammer"),
Backpack(name: "Bin", icon: "arrow.up.bin", content:
[Backpack(name: "Paperclip", icon: "paperclip"),
Backpack(name: "Magnifying glass", icon:
"magnifyingglass")
])
])
```

Using disclosure groups to hide and show content

`DisclosureGroup` is a view that's used to show or hide content based on the state of a disclosure control. It takes two parameters: a `label` to identify its content and a binding that controls whether the content is visible or hidden. Let's take a closer look at how it works by creating an app that shows and hides content in a disclosure group.

Getting ready

Create a new SwiftUI project and name it `DisclosureGroups`.

How to do it...

We will create an app that uses the DisclosureGroup view to view some planets in our solar system, continents on Earth, and some surprise text. The steps are as follows:

1. Below the ContentView struct, add a state property called showplanets:

```
struct ContentView: View {  
    @State private var showplanets = true  
    // the rest of the content here  
}
```

2. Replace the Text view in the body struct with a VStack and a DisclosureGroup view that contains two Text views with planet names:

```
var body: some View {  
    VStack {  
        DisclosureGroup("Planets", isExpanded:  
            $showplanets){  
            Text("Mercury")  
            Text("Venus")  
        }  
    }  
}
```

3. Add another DisclosureGroup to the view for planet Earth. This DisclosureGroup contains the list of Earth's continents:

```
var body: some View {  
    VStack {  
        DisclosureGroup("Planets", isExpanded:  
            $showplanets){  
            Text("Mercury")  
            Text("Venus")  
  
            DisclosureGroup("Earth"){  
                Text("North America")  
                Text("South America")  
                Text("Europe")  
                Text("Africa")  
            }  
        }  
    }  
}
```

```
    Text("Asia")
    Text("Antarctica")
    Text("Oceania")
}
}
}
}
```

4. Now, let's use a different way to define our DisclosureGroup view. Below our initial DisclosureGroup view, add another one that reveals surprise text when clicked:

```
DisclosureGroup("Earth") {
    // Content here not shown
}
}
DisclosureGroup{
    Text("Surprise! This is an alternative
        way of using DisclosureGroup")
} label : {
    Label("Tap to reveal", systemImage:
        "cube.box")
        .font(.system(size:25, design:
            .rounded))
        .foregroundColor(.blue)
}
```

The resulting preview should be as follows:

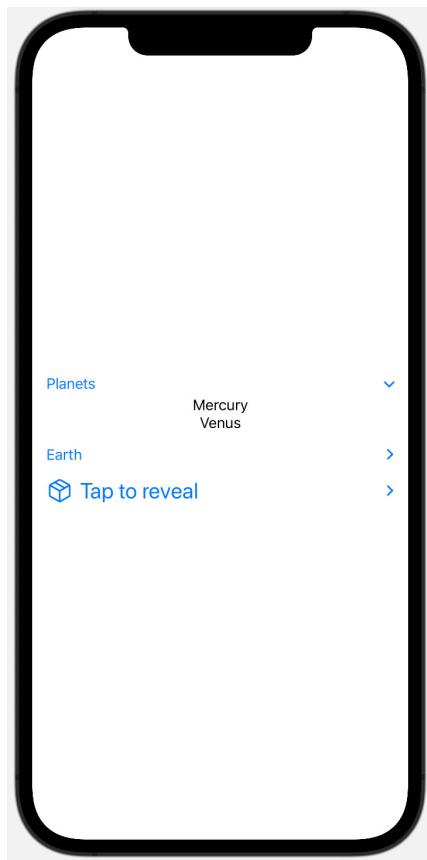


Figure 3.8 – DisclosureGroup preview

Run the Xcode live preview and click around to familiarize yourself with how the DisclosureGroup view works.

How it works...

Our initial DisclosureGroup view presents a list of planets. We can read the state change by passing a binding and knowing whether the DisclosureGroup view is in an open or closed state. Next, we used a DisclosureGroup view without bindings to display the continents on Earth.

Our third DisclosureGroup uses the closure syntax to separate the Text and label parts into two separate views, thus allowing for improved customization

DisclosureGroup views are versatile as they can be nested and used to display content hierarchically.

Creating SwiftUI widgets

Widgets show glanceable and relevant content from an app on an iOS device's home screen or the notification center in macOS. Examples of the most popular widgets are Apple's weather and stock apps.

There are two kinds of widgets configuration options. `StaticConfiguration` is used for widgets with no user-configurable properties, such as stock market apps, while `IntentConfiguration` is used for apps with user-configurable properties such as static widgets and intent widgets. `StaticConfiguration` widgets are not customizable, while `IntentConfiguration` widgets can be customized.

In this recipe, we'll create a static widget that displays a list of tasks sorted by priority. Each task will be displayed for 10 seconds to give the user enough time to complete the task (we are assuming the user has super speed and can do everything in 10 seconds).

Getting ready

Download this project from GitHub: <https://github.com/PacktPublishing/SwiftUI-Cookbook-2nd-Edition/tree/main/Chapter03-Advanced-Components/06-Creating-Widgets>. Then, run the TodoList app located in the StartingPoint folder.

How to do it...

Let's create a simple widget for a to-do list app. The app will display a list of tasks. Each task has four properties: `id`, `description`, its completed status, and `priority`. We will create a widget that sorts uncompleted tasks by priority and displays each one for 10 seconds. The steps are as follows:

1. Run the TodoList app in the StartingPoint folder.
2. Open the `Task.swift` file and notice how the task model is defined. Notice how an array of tasks is declared in the file.

3. Now, click on `ContentView.swift` and run the app preview to observe the app's original state. The result should look as follows:

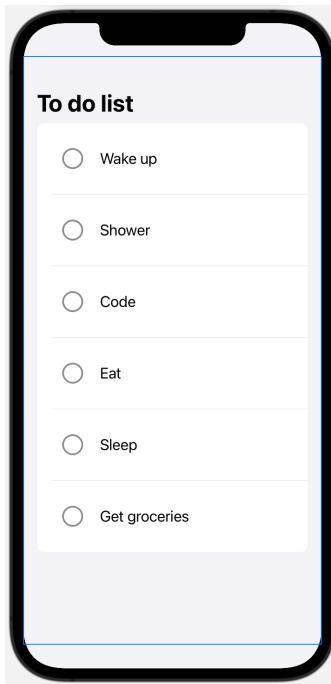


Figure 3.9 – The TodoList app's original state

4. Let's get started by adding new widget-related content. Select **File | New | Target**. Make sure that the template platform is **iOS** and enter the word **widget** in the filter, as shown here:

Choose a template for your new target:

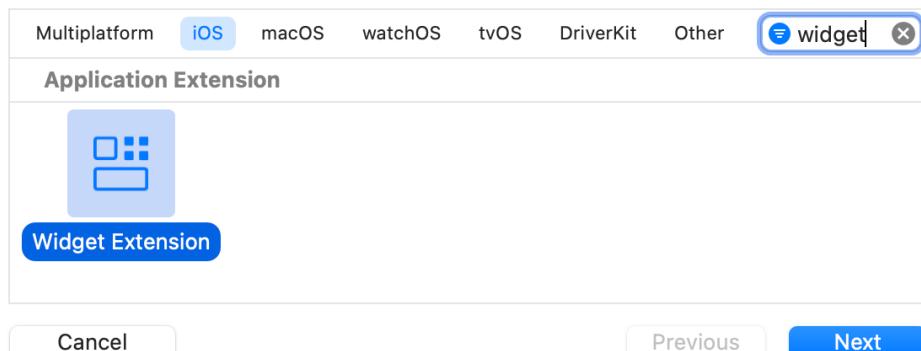


Figure 3.10 – Selecting the Widget Extension target

5. Click **Next**. Set **Product Name** to **TodoWidget** and make sure that you uncheck **Include Configuration Intent**:

Choose options for your new target:

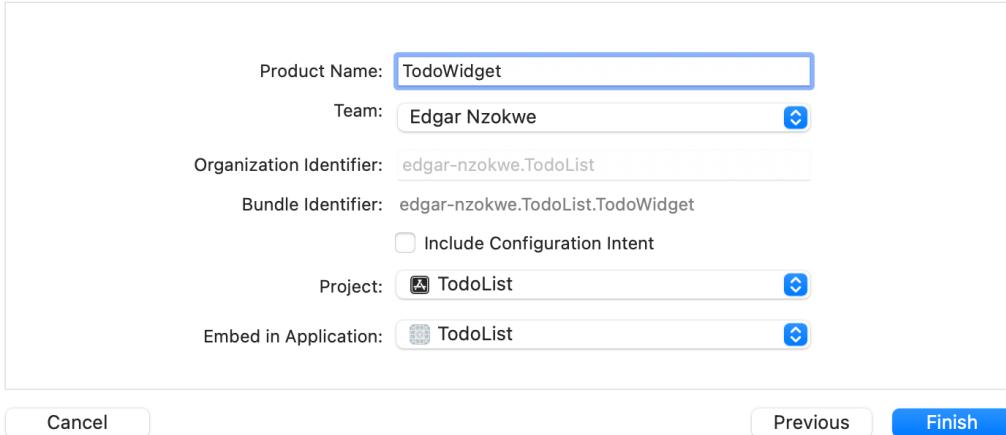


Figure 3.11 – Setting the widget's name

6. Click **Activate** on the next pop-up alert:

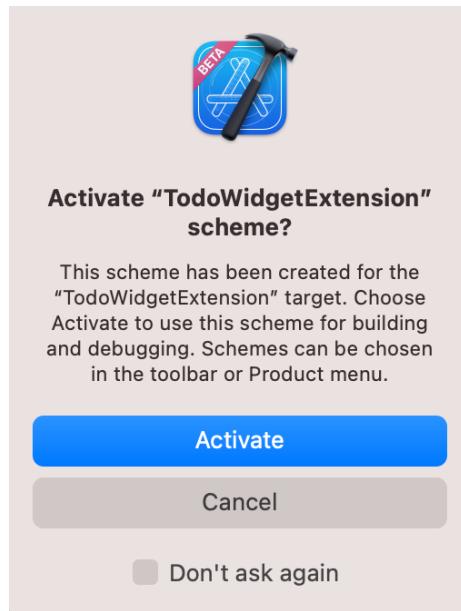


Figure 3.12 – Activating a new widget

7. Open the `TodoWidget.swift` file. It contains some template code for a widget that updates the current time every hour.
8. Optional: Run the widget and stare at it for an hour until the content changes (ha-ha, just kidding).
9. Let's start by updating our `TodoWidget_Previews` struct to display both the small and medium widget families.
10. Delete all the content of `TodoWidget.swift` except for the `import` statements.
11. Let's start by declaring how our entry should be defined. Create a `TaskEntry` struct that conforms to `TimelineEntry` and contains a `date` property and a `task` property:

```
struct TaskEntry: TimelineEntry{  
    let date: Date  
    let task: Task  
}
```

12. Now, create a `Provider` struct that conforms to the `TimelineProvider` protocol:

```
struct Provider: TimelineProvider{  
}  
}
```

13. Use *Command* (⌘) + *B* to build the code. Click on the red dot that appears in the editor. Click the **Fix** button to implement protocol stubs:

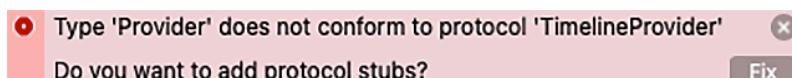


Figure 3.13 – Xcode build error notification

14. Enter `TaskEntry` to complete the `typealias` code prompt (`typealias` provides a new name to an existing type – using `Entry` to refer to our `TaskEntry`):

```
typealias Entry = TaskEntry
```

15. Use *Command* (⌘)+*B* to build the code again. Click on the Xcode alert and click **Fix** to add protocol stubs. The `placeholder`, `getSnapshot`, and `getTimeline` functions will be added to your project.
16. Now, update the `placeholder` and `getSnapshot` functions to display a sample task from our `tasks` array declared in the `Tasks.swift` file. The `Provider` struct should look as follows:

```
struct Provider: TimelineProvider{
    func placeholder(in context: Context) -> TaskEntry {
        TaskEntry(date: Date(), task:tasks[0])
    }

    func getSnapshot(in context: Context, completion:
        @escaping (TaskEntry) -> Void) {
        let entry = TaskEntry(date: Date(),
            task:tasks[0])
        completion(entry)
    }

    func getTimeline(in context: Context, completion:
        @escaping (Timeline<TaskEntry>) -> Void) {
        <#code#>
    }

    typealias Entry = TaskEntry
}
```

17. Delete `typealias Entry = TaskEntry`.
18. Use *Command* (⌘)+*B* to build the code again. This time, you'll get a **Cannot find 'tasks' in scope** error. There's no quick fix button on this error message ribbon to resolve the error.
19. Resolve the error by making the content of `Tasks.swift` available to our WidgetKit extension.

20. If the Inspector pane is not currently open, click the *Inspector* button at the top-right corner of Xcode. Next, select *Task.swift* from the navigator panel and then check the *TodoWidgetExtension* check box in the **Target Membership** section in the Inspector pane:

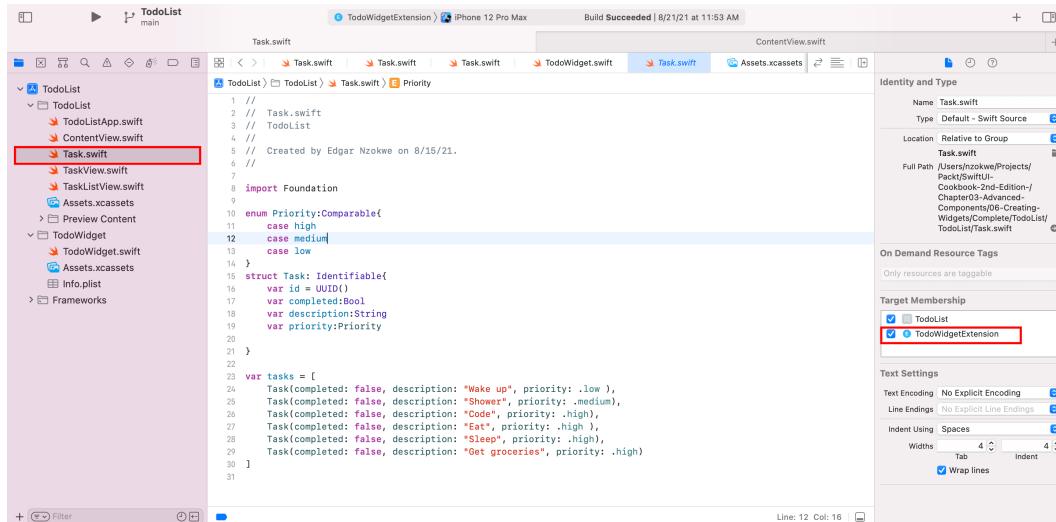


Figure 3.14 – Adding *Task.swift* to our *TodoWidgetExtension* target

21. Add the following code to *getTimeline*. This should allow us to sort our tasks by priority, create entries, and add the entries to our timeline:

```
func getTimeline(in context: Context, completion: @escaping (Timeline<Entry>) -> ()) {
    var entries: [TaskEntry] = []
    let currentDate = Date()
    let filteredTasks = tasks.sorted(by: {$0.priority > $1.priority})
    for index in 0..<filteredTasks.count {
        let task = filteredTasks[index]
        let entryDate =
            Calendar.current.date(byAdding: .second, value: index*10, to: currentDate)!
        let entry = TaskEntry(date: entryDate, task:task)
        entries.append(entry)
    }
    completion(Timeline(entries: entries, nextOffset: currentDate.timeIntervalSinceNow)))
}
```

```

        }

    let timeline = Timeline(entries: entries,
                           policy: .atEnd)
    completion(timeline)
}

```

22. Below the Provider struct, let's add a TodoWidgetEntryView that describes how to read and display content from one of our TodoEntry views:

```

struct TodoWidgetEntryView : View {
    var entry: Provider.Entry
    var body: some View {
        ZStack{
            Color("WidgetBackground")
                .ignoresSafeArea()
            Text(entry.task.description)
        }
    }
}

```

The WidgetBackground color has not been defined yet. Let's add it now.

23. Click on **Assets.xcassets** | **WidgetBackground** | select the **findHighlightColor** Content Color:

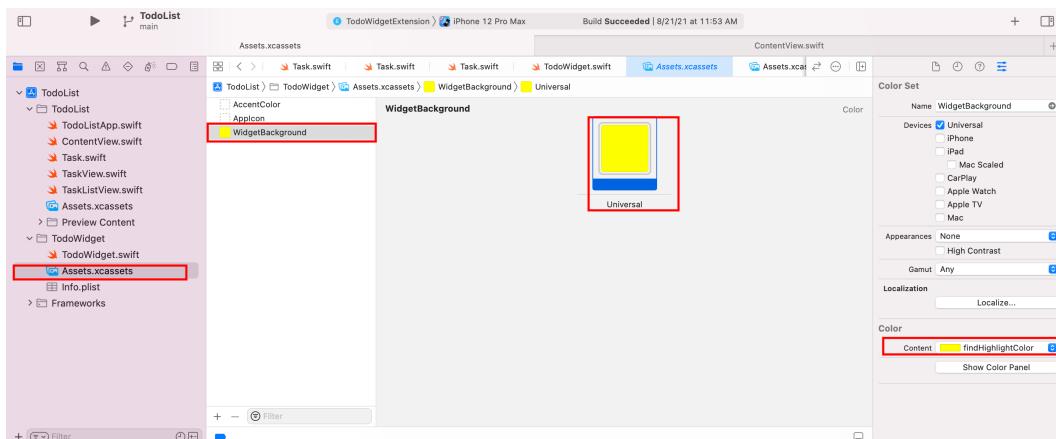


Figure 3.15 – Setting up the color of WidgetBackground

24. Let's put everything together with a struct that conforms to the Widget protocol. We will mark it with @main to let Xcode know that this should be the first function to execute when running our widget:

```
@main
struct TodoWidget: Widget {
    let kind: String = "TodoWidget"

    var body: some WidgetConfiguration {
        StaticConfiguration(kind: kind, provider:
            Provider()) { entry in
            TodoWidgetEntryView(entry: entry)
        }
        .configurationDisplayName("Task List Widget")
        .description("Shows next pressing item on a
                     todo list")
    }
}
```

25. Now, let's add a preview section so that we can preview any changes we make to the widget:

```
struct TodoWidget_Previews: PreviewProvider {
    static var previews: some View {
        Group{
            TodoWidgetEntryView(entry: TaskEntry(date:
Date(), task: tasks[0]))

            .previewContext(WidgetPreviewContext(family:
                .systemSmall))
            TodoWidgetEntryView(entry: TaskEntry
                (date: Date(), task: tasks[0]))

            .previewContext(WidgetPreviewContext(family:
                .systemMedium))

        }
    }
}
```

Run the app in an Xcode emulator. The result should look something like this:

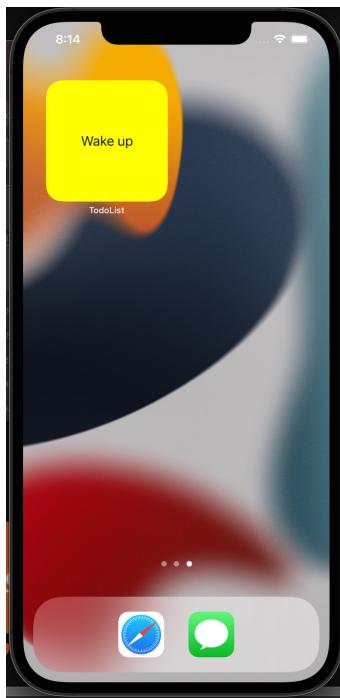


Figure 3.16 – The TodoList widget in the Xcode emulator

Observe the widget and notice that a new task is displayed every 10 seconds. You can also click on the widget to open the app and update the values in the to-do list. Nice work!

How it works...

An entry is a struct that holds some data and the date property that we would like WidgetKit to update the widget view with. For example, this project's entry is called TaskEntry.

A timeline is an array of entries that displays some content in the widget based on the date property that's assigned to each entry. This means that the content to be displayed within a certain timeframe is predetermined, thus requiring less computing resources. In addition, various refresh policies can be used to determine when new timeline entries should be computed.

Now that you've grasped the basics of WidgetKit, let's delve into the actions we took and why we took them.

Our widget should display a new task every 10 seconds. That means each of our entries should contain a date when it should be displayed and a task to be displayed. The `TaskEntry` struct fulfills that requirement.

The next task we did was creating a `Provider` struct that conforms to the `TimelineProvider` protocol. A `TimelineProvider` advises `WidgetKit` on when to update the widget's display. The `TimelineProvider` protocol contains three required functions:

- The `placeholder` function creates the `placeholder` view, which is used when `WidgetKit` displays the widget for the first time. A `placeholder` view gives the user a general idea of what the widget shows.
- The `getSnapshot` function is used to show the widget in the widget gallery. Use a sample date if it would take too long to fetch data for this function.
- The `getTimeLine` function provides an array of entries for the current time and optional future times to update the widget. For example, in our `getTimeLine` function, we sorted our list of tasks by priority with the highest one first, created an array of entries whose dates are 10 seconds apart, added those entries to our timeline, and called our `completion` handler function.

After completing our `Provider` struct, we created our `TodoWidgetEntryView` struct, which describes the design of our widget.

Our final step involved creating the main function, `TodoWidget`, which is the first function that gets called when we try to run the widget. We use `WidgetKit`'s `StaticConfiguration` object because we're not requesting any user configuration values. The `kind` string identifies the widget and should be descriptive. Next, the `provider` object created the timeline. Lastly, the callback function displayed a `TodoEntryView`. To specify what widget sizes we support, we added a `.supportedFamilies` modifier to the body view.

See also

Apple's documentation on creating widgets: <https://developer.apple.com/documentation/widgetkit/creating-a-widget-extension>.

4

Viewing while Building with SwiftUI Preview

Developing an application requires several interactions between clients and developers. For example, clients may sometimes request minor changes such as colors, fonts, and image positioning. Previously, developers would need to update their designs in Xcode and recompile all the code before viewing changes. SwiftUI solves this problem by introducing canvas previews. Previews allow for live viewing of UI changes without recompiling code.

In this chapter, we will learn how to make effective use of Xcode previews to speed up the UI development time. This chapter includes the following recipes:

- Previewing a layout in dark mode
- Previewing a layout at different dynamic type sizes
- Previewing a layout in a NavigationView
- Previewing a layout on different devices
- Using previews in UIKit
- Using mock data for previews

Technical requirements

The code for this chapter is based on Xcode 13, with iOS 15 set as the minimum target version.

The code for this section can be found in the GitHub repository at <https://github.com/PacktPublishing/SwiftUI-Cookbook-2nd-Edition/tree/main/Chapter04-Viewing-while-building-with-SwiftUI-Preview>.

Previewing a layout in dark mode

SwiftUI has built-in functionality to support dark mode. With Xcode previews, you can preview your views from light to dark mode with minimal effort by just changing the Xcode previews color scheme environment value.

In this recipe, we will add a dark mode to a minimalistic app with just a single `Text` view.

Getting ready

Create a new SwiftUI app named `DarkModePreview`.

How to do it

Since the inception of SwiftUI, there has been an issue with previews, where you cannot preview any view in dark mode except when the view is wrapped in a `NavigationView`. Is this a feature or a bug? Hopefully, we get the answer in the next releases of SwiftUI.

In this recipe, we'll enclose a `Text` view in a `NavigationView` and preview it in light and dark mode. The steps are as follows:

1. Open the `ContentView.swift` file.
2. Above the `body` variable, add an `@Environment` variable that checks for the current color scheme:

```
@Environment(\.colorScheme) var deviceColorScheme
```
3. Add a `NavigationView` component to the `body` variable of the `ContentView`:

```
NavigationView {  
    Text(deviceColorScheme == .dark ? "Quick journey to  
    the dark side" : "Back to the light")  
}
```

4. Within the `Content_Previews` struct, add a dark `.colorScheme()` modifier to the `ContentView` function:

```
ContentView().colorScheme(.dark)
```

5. To preview light mode at the same time as dark mode, enclose `ContentView` with a `Group` view and add another `ContentView` with a light color scheme:

```
static var previews: some View {  
    Group {  
        ContentView().colorScheme(.dark)  
        ContentView().colorScheme(.light)  
    }  
}
```

The resulting preview should look as follows:

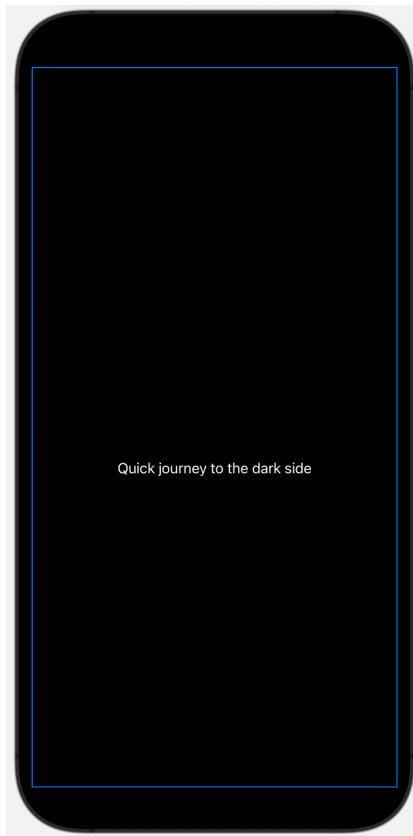


Figure 4.1 – Dark Mode preview

The light mode version should look as follows:

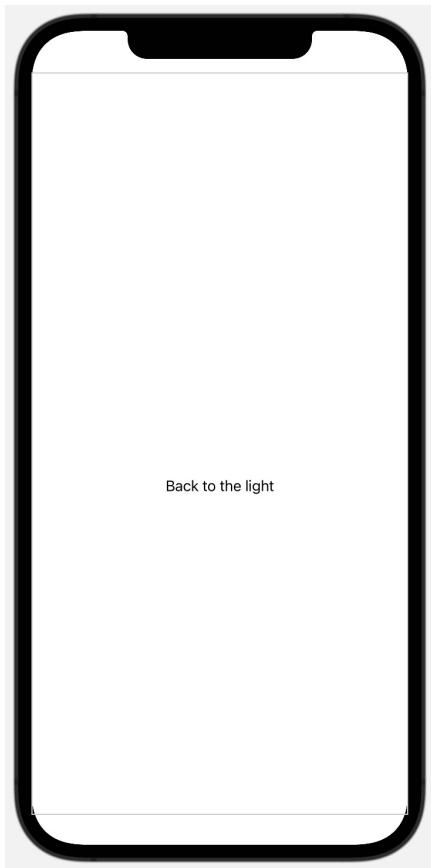


Figure 4.2 – Light mode preview

Notice little to no font changes were required to achieve dark mode because Xcode handles those adjustments by default.

How it works

The process of previewing content in dark mode is self-explanatory. The preview displays content from the `ContentView` view. If no color scheme is provided to the preview, it uses `.light` as the default color scheme.

To display different texts in light versus dark modes, we added a `deviceColorScheme` environment variable that detects the current scheme of the application. We then use a ternary operator in our `Text` view to change its value to **Quick journey to the dark side** when the dark color scheme is detected and **Back to the light** when the light color scheme is detected.

Important Note

Later versions of Xcode may not require a `NavigationView` to preview content in dark mode.

Previewing a layout at different dynamic type sizes

You may want to improve the accessibility features on your app to appeal to a wider variety of users. This means making sure the design looks good in different possible font sizes that users might prefer. SwiftUI previews are perfect for such use cases. They allow you to immediately preview how new content will look in different dynamic type sizes.

In this recipe, we will create an app that previews some news article titles using different dynamic type sizes.

Getting ready

Create a new SwiftUI app called `DynamicTypeSizesPreview`. Then, download the images for this chapter from GitHub: <https://github.com/PacktPublishing/SwiftUI-Cookbook-2nd-Edition/tree/main/Resources/Chapter04/recipe2>.

How to do it

A news article should contain an image, some text, and a description. First, let's create a model that describes a news article. Then let's create a view that shows how each article should be displayed, and finally, test out our view in different dynamic type sizes. The steps are as follows:

1. Expand the `Preview Content` folder in the Xcode navigation panel.
2. Click on `Preview Assets.xcassets` to view the page:

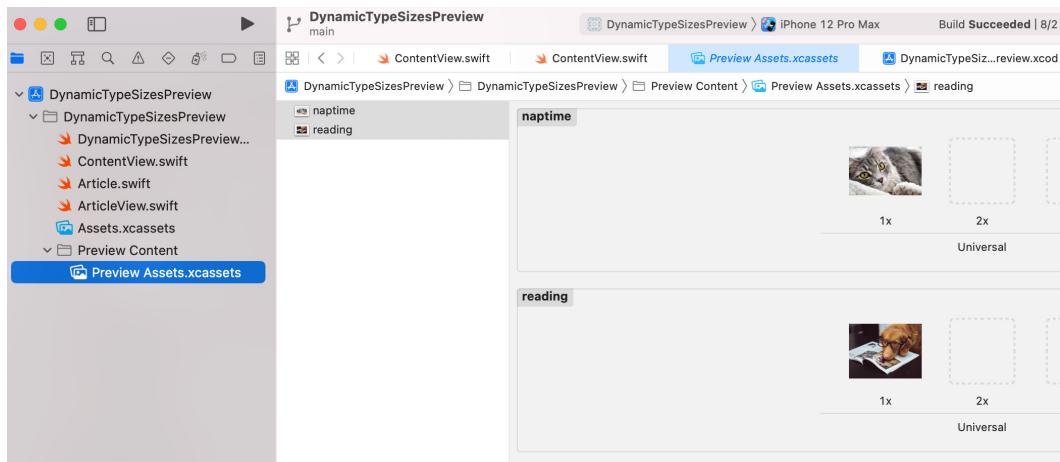


Figure 4.3 – Preview Content Assets

3. Drag and drop the image resources into the section shown in *Figure 4.3*.
4. Create a model for the news articles:
 - a. Press `Command (⌘) + N` to open the new file menu.
 - b. Select **Swift File** and click **Next**.
 - c. In the **Save As** field, enter `Article`.
 - d. Click **Create**.
5. Create an `Article` struct with `imageName`, `title`, and `description` properties:

```
struct Article {
    var imageName: String
    var title: String
    var description: String
}
```

6. Below our Article definition, create two sample articles. We will use them to preview our SwiftUI views:

```
let sampleArticle1 = Article(imageName: "reading",
    title: "Love reading", description: "Reading is
    essential to success")
let sampleArticle2 = Article(imageName: "naptime",
    title: "Nap time", description: "Take naps when
    tired to improve performance")
```

7. Create a SwiftUI view called ArticleView. This view explains how an article should be displayed:
 - a. Press **Command (⌘) + N** to open the new file menu.
 - b. Select **SwiftUI View** and click **Next**.
 - c. In the **Save As** field, enter ArticleView.
 - d. Click **Create**.
8. The ArticleView accepts an image as input and displays its content. The ArticleView should look as follows:

```
struct ArticleView: View {
    var article: Article
    var body: some View {
        HStack{
            Image(article.imageName)
                .resizable()
                .aspectRatio(contentMode: .fit)
                .frame(width:150, height:100)
                .clipShape(Ellipse())
        VStack {
            Text(article.title)
                .font(.title)
            Text(article.description)
                .padding()
        }
    }
}
```

```
    }  
}
```

9. To preview the design, let's pass a sample article to our ArticleView_Previews struct:

```
struct ArticleView_Previews: PreviewProvider {  
    static var previews: some View {  
        ArticleView(article: sampleArticle1)  
    }  
}
```

At this point, the ArticleView preview should look as follows:

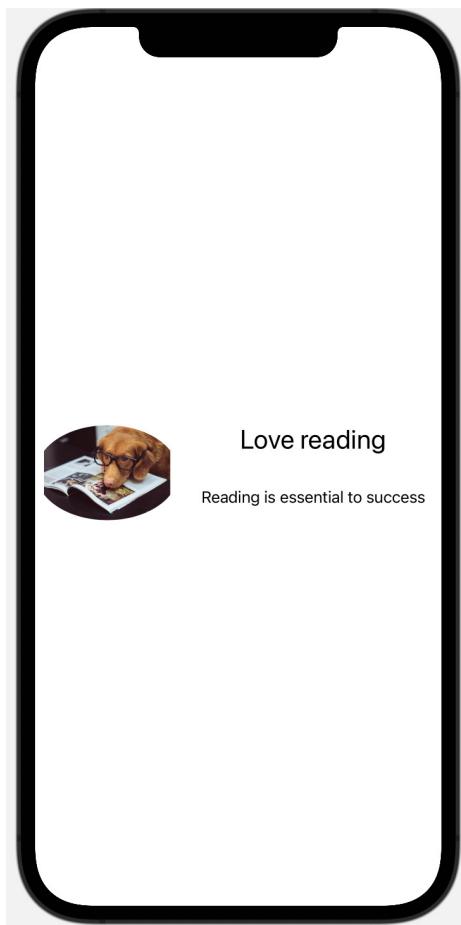


Figure 4.4 – ArticleView preview

10. Now update the `ContentView.swift` file to display a sample article:

```
var body: some View {  
    ArticleView(article: sampleArticle2)  
}
```

Run the canvas preview. The content result should be as follows:

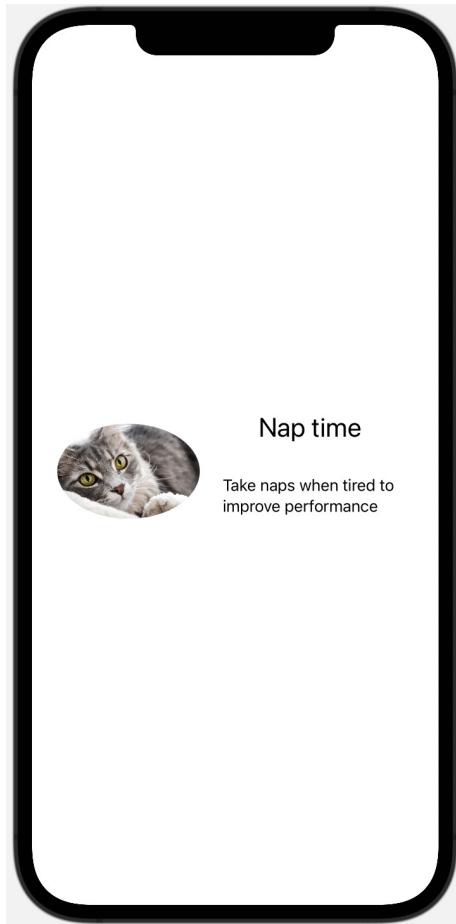


Figure 4.5 – Canvas preview

11. To view the results with different dynamic type sizes, we need to modify the `ContentView_Previews` struct. We would like to view multiple sizes at once, so let's start by enclosing the `ContentView` function in a `Group` view:

```
Group {  
    ContentView()  
}
```

12. Add a second `ContentView`, but this time add an `.environment()` modifier that shows different size categories:

```
Group {  
    ContentView()  
    ContentView()  
        .environment(\.dynamicTypeSize,  
                  .xSmall)  
}
```

13. Notice that each of our `ContentView` gets displayed on a new device, albeit our content doesn't use the whole screen. So, let's add a `.previewLayout()` modifier to our group to limit the preview content to the size that fits the device's screen:

```
Group {  
    //Previously mentioned content here  
} .previewLayout(.sizeThatFits)
```

14. Finally, add a third `ContentView` function call to our `Group` with a new `.environment` modifier:

```
ContentView()  
.environment(\.dynamicTypeSize, .accessibility5)
```

The preview should look as follows:

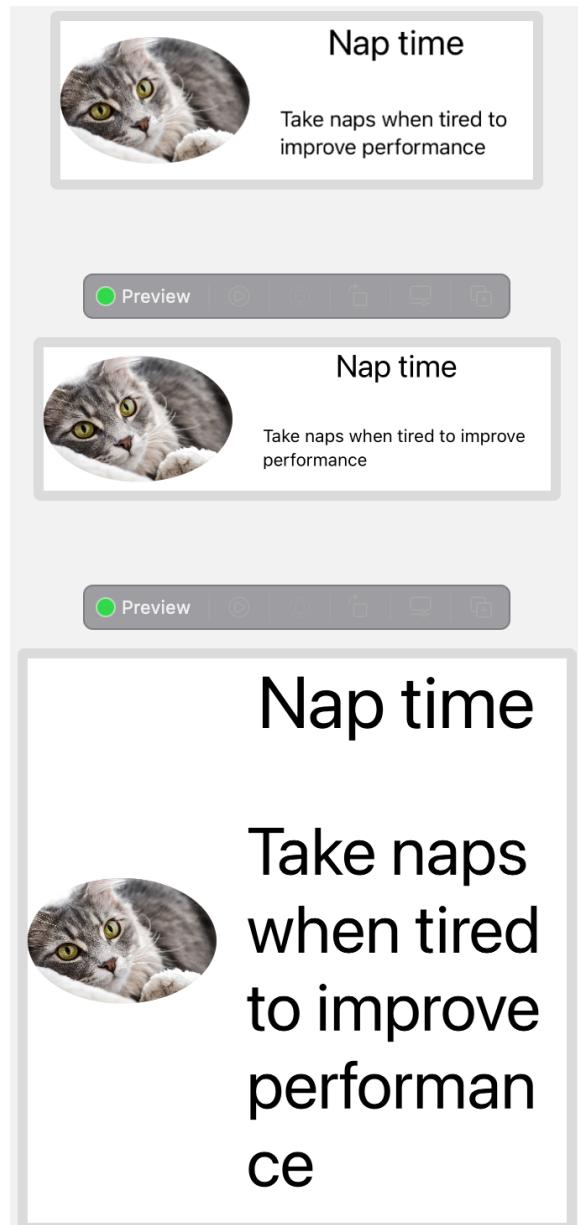


Figure 4.6 – Previewing content in dynamic type sizes

Click on the play button at the top of each size category to preview how the layout will look on a device.

How it works

We made this app modular by separating the model from the view. `Article.swift` describes the content that we should expect in an article, while `ArticleView.swift` defines how each article should be displayed. We've already touched on all the modifiers used in this section, so let's dive into our main concern, dynamic type sizes.

Our `ContentView.swift` file displays a sample article, and its preview implements the code required to display multiple dynamic type sizes.

Before previewing the layout with multiple dynamic type sizes, we embedded the `ContentView` function in a `Group` view. That way, we're able to display changes in multiple different devices simultaneously.

We preview the layout in different type sizes by using the `.environment()` modifier and different `dynamicTypeSize` enums. However, after adding more content views with different sizes, we notice that each gets displayed on a separate device. As a result, we end up spending too much time scrolling rather than viewing all our pertinent content at once. We solve this problem by adding the `.previewLayout(.sizeThatFits)` modifier to the `Group` view, thus restricting each preview to the amount of space used to display its content.

See also

More dynamic type sizes: <https://developer.apple.com/documentation/swiftui/dynamictypesize>

Previewing a layout in a NavigationView

Some views are designed to be presented in a navigation stack but are not part of a `NavigationView`. One solution to this problem would be to run the application and navigate to the view in question. However, previews provide a time-saving way to view UI change live without rebuilding the app.

In this recipe, we will create an app with a view that is part of the navigation stack and preview it in a `NavigationView`.

Getting ready

Let's create a SwiftUI app called `PreviewingInNavigationView`.

How to do it

We will add a `NavigationView` and `NavLink` component to the `ContentView` struct. The `NavLink` opens up a second view that doesn't contain a `NavigationView`. Since the second view will always be displayed in a navigation stack, we will update the preview to always display our design within a `NavigationView`. The steps are as follows:

1. Replace the `Text` view in `ContentView` with a `NavigationView` containing a `VStack` component and `NavLink` that directs us to `SecondView`:

```
struct ContentView: View {
    var body: some View {
        NavigationView {
            VStack {
                NavigationLink(destination:
                    SecondView(someText: "Sample text")){
                    Text("Go to second view")
                        .foregroundColor(Color.white)
                        .padding()
                        .background(Color.black)
                        .cornerRadius(25)
                }
            }.navigationBarTitle("Previews",
                displayMode: .inline)
        }
    }
}
```

2. Create a new SwiftUI view called `SecondView`:
 - a. Press *Command* (⌘) + *N*.
 - b. Select **SwiftUI View**.
 - c. Enter `SecondView` in the **Save As** field.
 - d. Click **Create**.
3. Add the following content to the `SecondView.swift` file:

```
struct SecondView: View {
    var someText: String
```

```
var body: some View {
    Text(someText)
        .navigationBarTitle("Second View",
            displayMode: .inline)
}
}

struct SecondView_Previews: PreviewProvider {
    static var previews: some View {
        SecondView(someText: "Testing")
    }
}
```

Observe the canvas preview; you'll see that no navigation bar title is present:

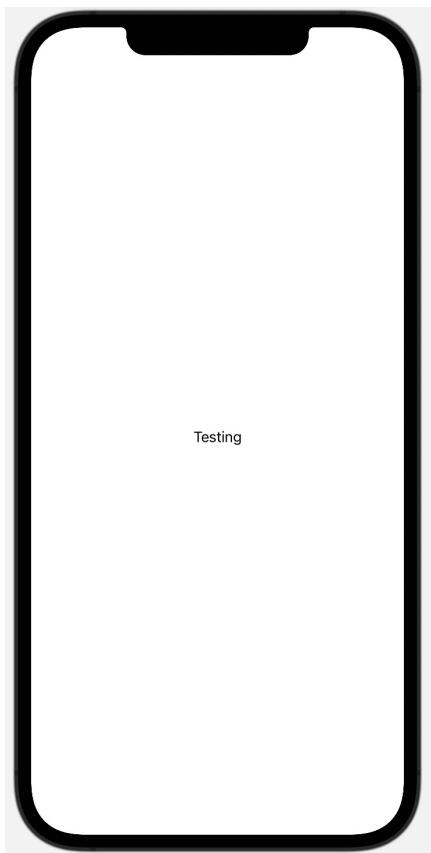


Figure 4.7 – SecondView with no navigation bar

4. Open the `ContentView.swift` file.
5. Run/resume the canvas preview and click the play button.
6. Click on the **Go to Second** view button to navigate to the second view.

The second view should look as follows:



Figure 4.8 – SecondView UI with a NavigationView

7. Now go back to our `SecondView.swift` file and add the following to preview it in a `NavigationView`:

```
struct SecondView_Previews: PreviewProvider {
    static var previews: some View {
        NavigationView{
            SecondView(someText: "Testing")
        }
    }
}
```

```
    }
}
```

Using `NavigationView` allows us to quickly and easily update views that are part of a navigation stack without first running the app.

How it works

Pages containing a `.navigationBarTitle()` modifier but no `NavigationView` will always be displayed as part of a navigation stack. Enclosing the preview of such pages in a `NavigationView` component provides a quick way of accurately previewing the UI.

Previewing a layout on different devices

SwiftUI allows us to preview designs on multiple screen sizes and device types simultaneously using the `.previewDevice()` modifier. In this recipe, we will create a simple app that displays an image and some text on multiple devices.

Getting ready

Let's create a SwiftUI app named `PreviewOnDifferentDevices`.

How to do it

We will add an image and some text to the `ContentView` struct and then modify the preview to show the content on multiple devices. The steps are as follows:

1. Click on the **Preview Content** folder in the Xcode navigation panel.
2. Import the `friendship.jpg` image from this chapter's resources file. Or download it at <https://github.com/PacktPublishing/SwiftUI-Cookbook-2nd-Edition/blob/main/Resources/Chapter04/recipe4/friendship.jpg>.
3. Open the `ContentView.swift` file and add the following code to display an image and some text:

```
struct ContentView: View {
    var body: some View {
        VStack{
            Image("friendship")
            .resizable()
```

```
        .aspectRatio(contentMode: .fit)
        Text("Importance of
Friendship").font(.title)
        Text("Friends helps us deal with
stress and make better life choices")
        .multilineTextAlignment(.center).padding()
    }
}
}
```

4. Now, let's modify the preview to view the content on multiple devices. Add the following to the `ContentView_Previews` struct:

```
struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        Group {
            ContentView()
                .previewDevice("iPhone 11 Pro Max")
                .previewDisplayName("Iphone 11 Pro Max")
            ContentView()
                .previewDevice("iPhone 8")
                .previewDisplayName("iPhone 8")
            ContentView()
                .previewLayout(.fixed(width: 568,
height: 320))
                .previewDisplayName("Custom Size to
infer landscape mode")
        }
    }
}
```

With the preceding code, we can view our changes on multiple devices immediately.

How it works

The `.previewDevice()` modifier lets you select a device to display your views. The device name you use in the modifier should be the same name as the devices available in the Xcode **Destination** menu:

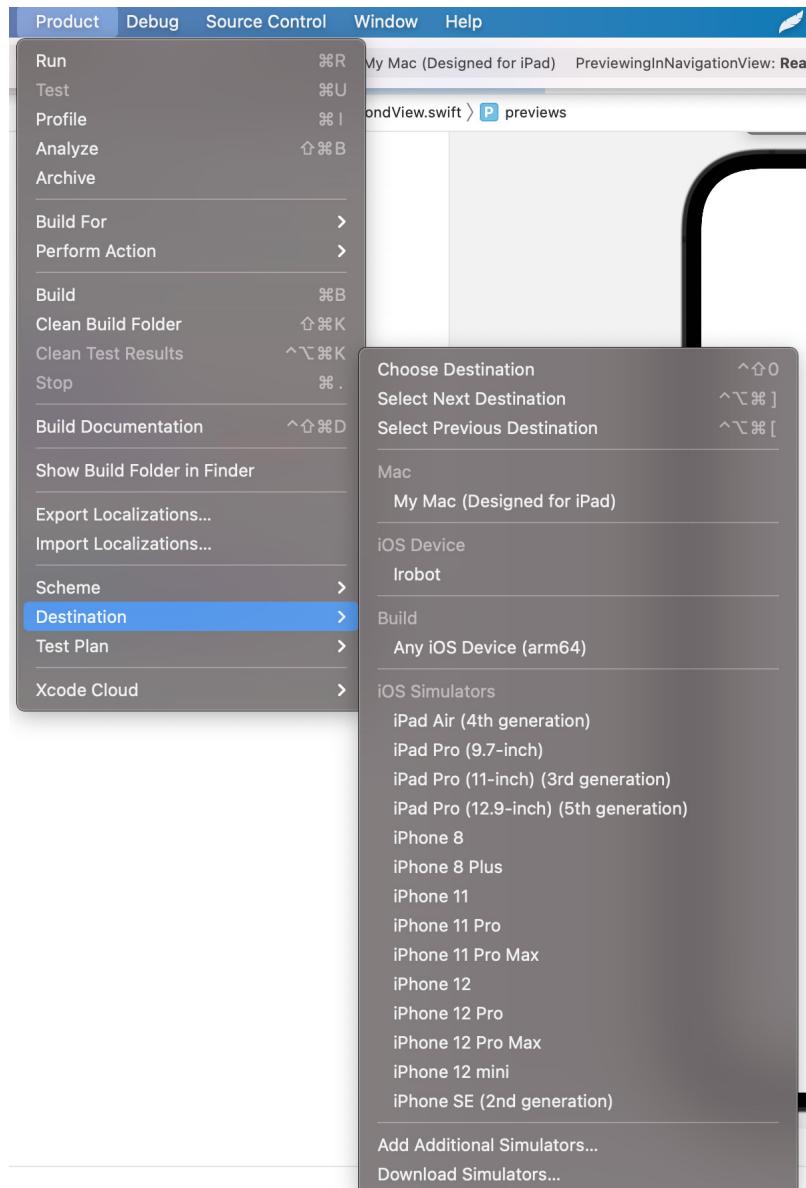


Figure 4.9 – List of available devices in the Xcode Destination menu

By default, all previews display devices in portrait mode. To preview views in landscape mode, find out the device landscape dimensions and specify the size in the `.previewLayout()` modifier as seen in the last preview item we added in *step 4*.

Using previews in UIKit

If you love the ease with which you can preview UI changes, but you are only working on UIKit projects, rest easy as you can also use this great feature while building UIKit apps.

In this recipe, we will learn how to wrap a `UIViewController` and `UIView` into SwiftUI views and then use live previews.

Getting ready

Clone or download the code for this book from GitHub: <https://github.com/PacktPublishing/SwiftUI-Cookbook-2nd-Edition>.

How to do it

We will check the app's build settings to ensure that iOS 13+ capabilities are enabled in the debug build. We'll then create a generic `struct` that can be used to preview any UIKit `ViewController` in the Xcode canvas. The steps are as follows:

1. Go to this chapter's code in GitHub. For this recipe, open the Xcode project located in the `StartingPoint` file.

2. Click on the **PreviewUIKitViews** project file in the Xcode navigation pane, select **Build Settings**, scroll down to the **Deployment** section, and make sure **iOS Deployment Target's Debug** field has a value of *iOS 13 or higher*:

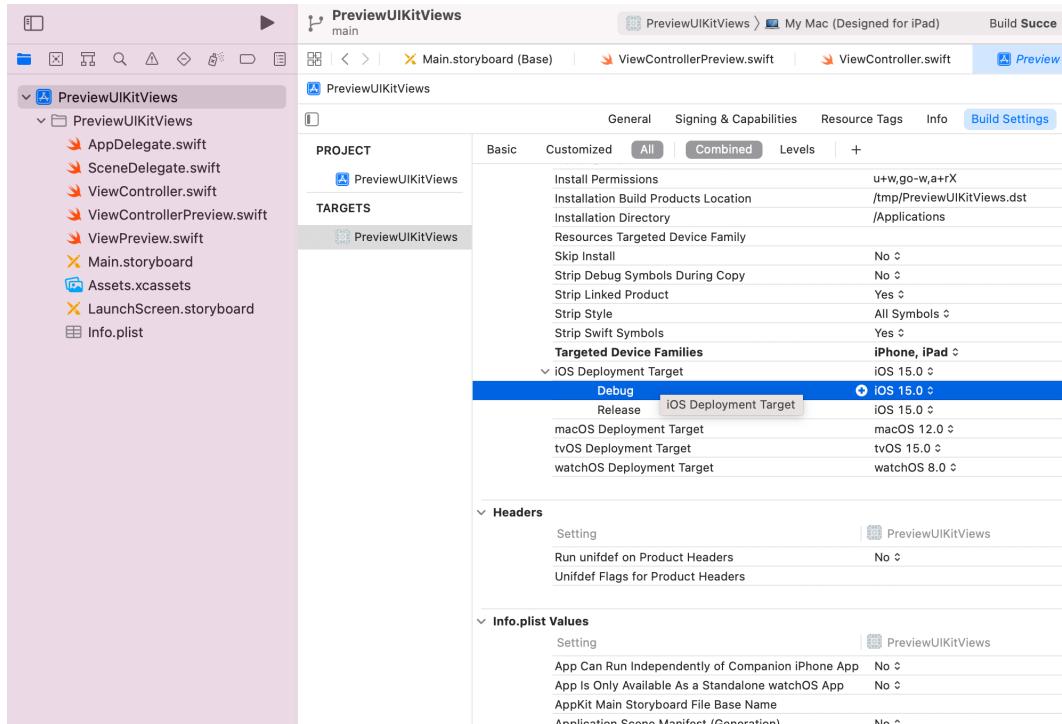


Figure 4.10 – Xcode build settings with the deployment targets highlighted

3. Open the **ViewController.swift** file. The `viewDidLoad` function contains some code to display a UIKit `Label`. Press the Play button in the top left corner of Xcode to run the project in a simulator and notice the label being displayed.

Now let's take steps that will allow us to preview design changes quickly using SwiftUI previews.

4. Create a new Swift file:
 - a. Press **Command (⌘) + N** to open the new file menu.
 - b. Select **Swift File** and click **Next**.
 - c. In the **Save As** field, enter `ViewControllerPreview`.
 - d. Click **Create**.

5. Delete any imports present, import SwiftUI and UIKit, then create a `ViewControllerPreview` that conforms to the `UIViewControllerRepresentable` protocol:

```
import UIKit
import SwiftUI

struct ViewControllerPreview:
    UIViewControllerRepresentable {
}
```

6. Let's add a `viewControllerBuilder` property that accepts a closure and returns a `UIViewController`:

```
struct ViewControllerPreview:
    UIViewControllerRepresentable {
    let viewControllerBuilder: () -> UIViewController
    init(_ viewControllerBuilder: @escaping () ->
        UIViewController) {
        self.viewControllerBuilder =
        viewControllerBuilder
    }
}
```

7. Now add the other two functions required to conform to the `UIViewControllerRepresentable` protocol – `makeUIViewController` and `updateUIViewController`:

```
func makeUIViewController(context: Context) ->
    some UIViewController {
    return viewControllerBuilder()
}

func updateUIViewController(_ uiViewController:
    UIViewControllerType, context: Context) {
    // Stays empty because we're not dealing with
    // state changes
}
```

8. Finally, add `ViewControllerPreview` to the `ViewController.swift` file, just below the `ViewController` class definition:

```
struct ViewController_Preview: PreviewProvider {  
    static var previews: some View {  
        ViewControllerPreview {  
            ViewController()  
        }  
    }  
}
```

9. Now open Xcode's canvas to display the Live Preview of your UIKit app. It should look as follows:



Figure 4.11 – Live Preview of UIKit app

Wrapping a `UIView` into a `SwiftUI` view is like wrapping a `UIViewController`. The code can be found in the `ViewPreview.swift` file located in the `Completed` folder of this recipe.

How it works

To preview a `UIViewController` object in `SwiftUI`, we need to implement the `UIViewControllerRepresentable` protocol. Our view controller accepts a closure that returns a `UIViewController`, thus creating a generic solution that will work with all view controllers instead of a specific one.

To conform to the `UIViewControllerRepresentable`, we further add two functions, `makeUIViewController` and `updateUIViewController`.

The `makeUIViewController` function creates the view controller object and configures its initial state while the `updateUIViewController` function updates the specified state of the view controller with new information from `SwiftUI`. The latter stays empty since we are not dealing with any state changes in this app.

After completing the preceding steps, all we need to do is implement our regular `SwiftUI` preview provider on the `UIKit` page, but this time we pass the `UIKit` view controller that has been wrapped in our custom `ViewControllerPreview` struct.

See also

How to use Live Previews by Finsi Ennes: <https://medium.com/swlh/how-to-use-live-previews-in-uikit-204f028df3a9>

Apple's documentation on `UIViewControllerRepresentable`:
<https://developer.apple.com/documentation/swiftui/uiviewcontrollerrepresentable>

Using mock data for previews

So far, we've built apps using our data. However, when dealing with large projects, data is usually obtained by making API calls. But that may be time-consuming and quickly become a bottleneck. The faster option would be to make some mock data available only at build time.

In this recipe, we will store some mock **JavaScript Object Notation (JSON)** data on insects in our `Preview Content` folder and fetch our data from the file instead of making API calls. JSON is lightweight format used for storing and transporting data.

Getting ready

Create a new SwiftUI project called `UsingMockDataForPreviews`.

To get access to the files used here, clone/download this project from GitHub:
<https://github.com/PacktPublishing/SwiftUI-Cookbook-2nd-Edition>.

How to do it

We will add a JSON file with sample data to our Xcode project and design an app that elegantly displays that data. The steps are as follows:

1. Open the `recipe6` folder for this project located at **Resources | Chapter04**.
2. Drag and drop the `insectData.json` file into the **Preview Content** folder of our Xcode project:

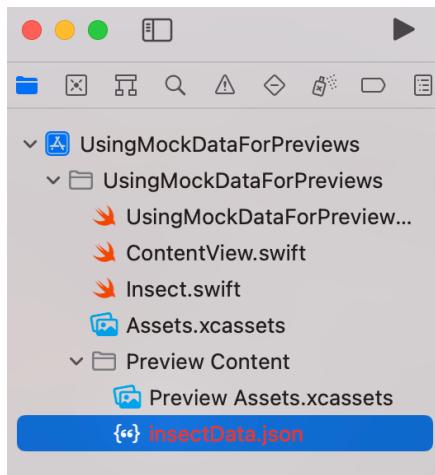


Figure 4.12 – Mock data in the Preview Content folder

3. Drag and drop the insect images into our Preview Assets.xcassets folder:

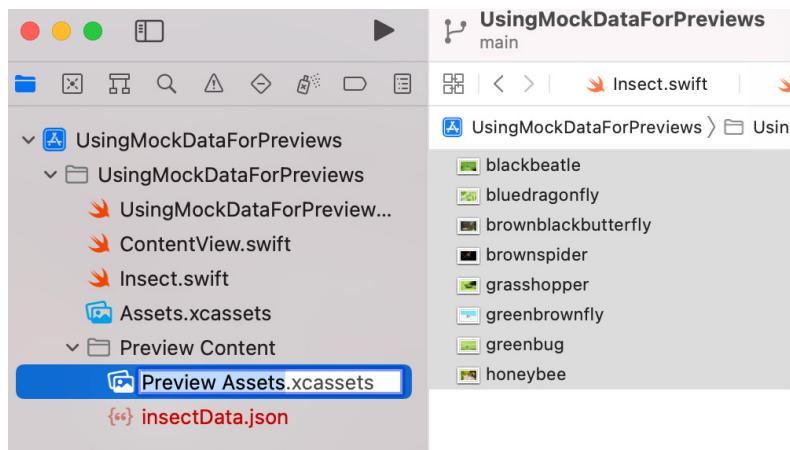


Figure 4.13 – Adding the insect images to the Preview Content folder

4. Click on the `insectData.json` file to view its content. Now create a model that describes the data:
 - a. Press *Command* (⌘) + *N* to open the new file menu.
 - b. Select **Swift File** and click **Next**.
 - c. In the **Save As** field, enter `Insect`.
 - d. Click **Create**.
5. Add the following content to the `Insect.swift` file:

```
struct Insect : Decodable, Identifiable{
    var id: Int
    var imageName: String
    var name: String
    var habitat: String
    var description: String
}

let testInsect = Insect(id: 1, imageName: "grasshopper",
name: "grass", habitat: "rocks", description: "none")
```

6. Open the ContentView.swift file. Add the following code to display a list of insects:

```
struct ContentView: View {
    var insects: [Insect] = []
    var body: some View {
        NavigationView {
            List {
                ForEach(insects) {insect in
                    HStack{
                        Image(insect.imageName)
                            .resizable()
                            .aspectRatio(contentMode: .fit)
                            .clipShape(Rectangle())
                            .frame(width:100, height: 80)

                        VStack(alignment: .leading){
                            Text(insect.name).font(.title)
                            Text(insect.habitat)
                        }.padding(.vertical)
                    }
                }
            }.navigationBarTitle("Insects")
        }
    }
}
```

7. Now let's update ContentView_Previews to read and display content from our JSON file with the mock data:

```
struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView(insects: Self.testInsects)
    }
    static var testInsects : [Insect]{
        guard let url = Bundle.main.url(forResource:
            "insectData", withExtension: "json"),

            let data = try? Data(contentsOf: url)
```

```
        else{
            return []
        }
        let decoder = JSONDecoder()
        let array = try?decoder.decode([Insect].self,
            from: data)
        return array ?? [testInsect]
    }
}
```

If everything was done right, your canvas preview should look as follows:

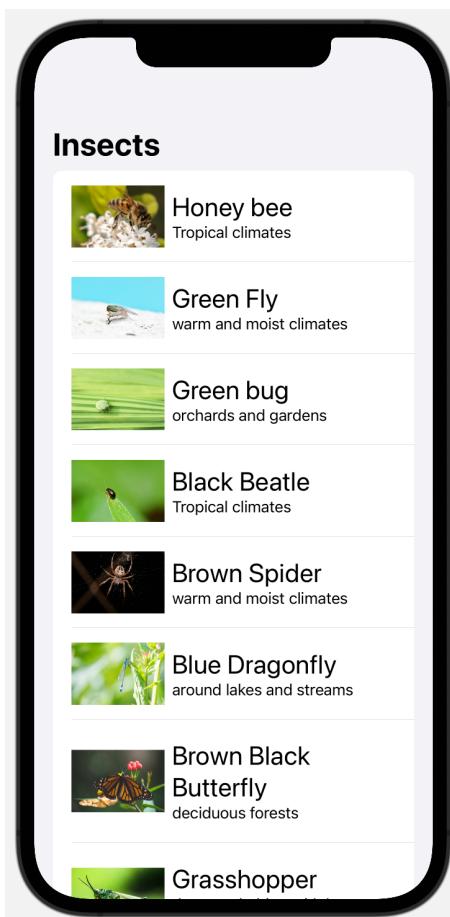


Figure 4.14 – Using mock data in the project preview

Play with the live preview to see how the app works, and then make any design changes you'd like to see.

How it works

In this recipe, we modeled our `Insect` struct based on our sample JSON data. The `Insect` struct implements the `Decodable` protocol to decode JSON objects into the struct. Implementing the `Identifiable` protocol requires each item to have a unique `id`, and thus can be used in a `ForEach` structure without explicitly passing in an `id` parameter.

In `ContentView_Previews`, we perform a number of steps to convert our data into an array of `Insect`. We first get the data from the file:

```
guard let url = Bundle.main.url(forResource: "insectData",
    withExtension: "json"),

        let data = try? Data(contentsOf: url)
    else{
        return []
    }
```

Then we decode the data into an array of `Insect` structs and return the results:

```
let decoder = JSONDecoder()
let array = try?decoder.decode([Insect].self, from: data)
return array ?? [testInsect]
```

The `return` statement sends back a decoded array if not `nil`. Otherwise, it sends back an array of one object, the `testInsect` that we created in our `Insect.swift` file.

There's more

The app could be modularized by breaking down some components in the `ContentView.swift` file.

Try breaking down the code into `InsectView` and `InsectListView` SwiftUI views. The `InsectView` should focus on displaying data regarding one insect. The `InsectListView` takes an array of insects and displays them in a list using our `InsectView` SwiftUI view.

Finally, our `ContentView.swift` file should do nothing else but call our `InsectListView` while passing an array of `Insects`.

5

Creating New Components and Grouping Views with Container Views

Form views are one of the best ways to get input from users. Proper implementation of forms improves the **User Experience (UX)** and increases the chances of retention, whereas complex or frustrating forms lead to a negative UX and low retention rates. We used forms and containers in the earlier chapters, but here we will learn additional ways of using forms.

We will also learn more about SwiftUI's `TabView` struct; this is similar to UIKit's `UITabBarController`, which provides an easy and intuitive way to present multiple views to the user while allowing them to easily navigate between each of them.

In this chapter, we will focus on grouping views using `Form` and `TabView` views. The following concepts will be discussed:

- Showing and hiding sections in forms
- Disabling and enabling items in forms
- Filling out forms easily using Focus and Submit
- Navigating between multiple views with `TabView`
- Using gestures with `TabView`

Technical requirements

The code in this chapter is based on Xcode 13 and Swift iOS 15.

You can find the code for this chapter in this book's GitHub repository at <https://github.com/PacktPublishing/SwiftUI-Cookbook-2nd-Edition/tree/main/Chapter05-Creating-new-Components-Grouping-views-in-Container-Views>.

Showing and hiding sections in forms

Forms provide a means of getting information from the user. Users get discouraged when completing very long forms, yet fewer people submitting a form may mean less data for your surveys, fewer signups for your app, or fewer people providing whatever data you're collecting.

In this recipe, we will learn how to show/hide the additional address section of a form based on the user's input.

Getting ready

Create a new SwiftUI project named `SignUp`.

How to do it...

We will create a signup form with sections for various user inputs. One of the sections, additional address, will be shown or hidden based on how long the user has lived at their current address.

The steps are given here:

1. Create a new SwiftUI view called `signUpView`:
 - a. Press *Command* (⌘) + *N*.
 - b. Select **SwiftUI View**.
 - c. Click **Next**.
 - d. Name the view `signUpView`.
 - e. Click **Finish**.
2. In `signUpView`, declare and initialize the `@State` variables that will be used in the form:

```
struct signUpView: View {
    @State private var fname = ""
    @State private var lname = ""
    @State private var street = ""
    @State private var city = ""
    @State private var zip = ""
    @State private var lessThanTwo = false
    @State private var username = ""
    @State private var password = ""
    // More content here
}
```

3. Add a `NavigationView` a `Form` view, and a `.navigationBarTitle` modifier to the `body` variable:

```
var body: some View {
    NavigationView{
        Form{
            }.navigationBarTitle("Sign Up")
    }
}
```

4. Add a `Section` to the form with two `TextField` structs to store the user's first and last names:

```
Section(header: Text("Names")){
    TextField("First Name", text: $fname)
```

```
    TextField("Last Name" , text: $lname)
}
```

5. Just below the `Section` defined in the previous step, add another `Section` with fields for the user's current address:

```
Section(header: Text("Current Address")) {
    TextField("Street Address" , text: $street)
    TextField("City" , text: $city)
    TextField("Zip" , text: $zip)
}
```

6. Add a `Toggle` field to the preceding form to inquire whether the user has been at their current address for at least 2 years. Place the `Toggle` field below the `TextField` where we request the user's zip code:

```
Toggle(isOn: $lessThanTwo) {
    Text("Have you lived here for 2+ years")
}
```

7. Add an `if` conditional statement to display a `Previous Address` field if the user has not been at their current location for more than 2 years:

```
if !lessThanTwo{
    Section("Previous Address") {
        TextField("Street Address" , text: $street)
        TextField("City" , text: $city)
        TextField("Zip" , text: $zip)
    }
}
```

8. Next, add a `Section` to the form with a `TextField` struct for the username and a `SecureField` struct for the password:

```
Section(header:Text("Create Account Info")) {
    TextField("Create Username" , text: $username)
    SecureField("Password", text: $password)
}
```

- Finally, let's add a **Submit** button. The Button displays the text "Submit", but does not perform any action when clicked:

```
Button("Submit") {  
    print("Form submit action here")  
}
```

The resulting preview should be similar to this:

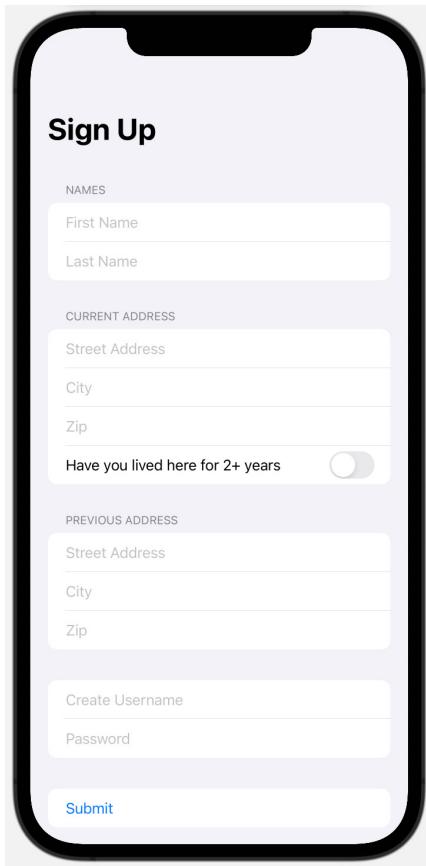


Figure 5.1 – Signup form

How it works...

A basic Form view with a TextView view is created by wrapping the TextView view with a Form view, then wrapping the Form view with a NavigationView view, as follows:

```
struct signUp: View {  
    @State var firstName: String = ""  
  
    var body: some View {  
        NavigationView {  
            Form {  
                TextField("firstName", text: $username)  
            }  
            .navigationBarTitle("Settings")  
        }  
    }  
}
```

We also create a `@State` variable to hold the user's input and a field to request for said input. You can add more variables and more fields, but the overall structure of the code is similar to that of the preceding code block.

We use an `if` conditional statement to show or hide sections or fields from our form. For example, the **Previous Address** section of the form gets displayed only when the value of the `lessThanTwo` variable is set to `true`.

We also use `Section` views to provide a visible separation between related fields. Additionally, you can add headers to `Section` views, such as the **NAMES** section, or provide no header, such as the fields for `username` and `password`.

There's more...

If you run the code in an emulator and click the **Submit** button, you'll notice that no `print` statement gets displayed. To see `print` statements, you need to make sure the **Debug Area** is visible (`View | Debug Area | Show Debug Area`):

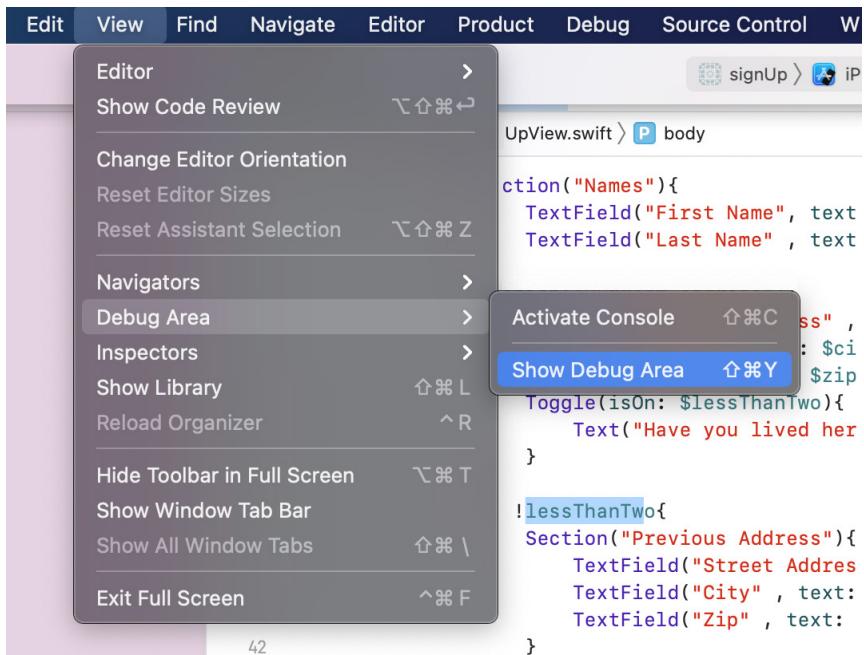


Figure 5.2 – Debug Area toggle

Now, run the app on an emulator or physical device.

If the **Debug Area** is visible and the app is running on an emulator or physical device, you will see print statements being displayed in the **Debug Area** each time the **Submit** button is clicked.

See also

Apple documentation on forms: <https://developer.apple.com/documentation/swiftui/form>

Disabling and enabling items in forms

Form fields may have additional requirements such as minimum text length or a combination of uppercase and lowercase characters. We may want to perform actions based on the user's input, such as disabling a **Submit** button till all requirements are met.

In this recipe, we will create a sign-in view where the **Submit** button only gets enabled if the user enters some content in both the username and password fields.

Getting ready

Create a SwiftUI project called `FormFieldDisable`.

How to do it...

We will create a login screen containing a username, password, and a **Submit** button. We will disable the **Submit** button by default and only enable it when the user enters some text in the username and password fields. The steps are given here:

1. Create a new SwiftUI view file named `LoginView`:
 - a. Press *Command* (⌘) + *N*.
 - b. Select **SwiftUI View**.
 - c. Click **Next**.
 - d. Enter `LoginView` in the **Save as** field.
 - e. Click **Finish**.
2. Open the `LoginView.swift` file and add a `@State` variable to hold the username and password input:

```
struct LoginView: View {  
    @State private var username = ""  
    @State private var password = ""  
    ...  
}
```

3. In the `body` variable, add a `VStack` component and a `Text` struct that displays the game's title, **Dungeons and Wagons**:

```
var body: some View {  
    VStack{  
        Text("Dungeons and Wagons")  
            .fontWeight(.heavy)  
            .foregroundColor(.blue)  
            .font(.largeTitle)  
            .padding(.bottom, 30)  
    }  
    ...  
}
```

4. Add a placeholder below the Text view for the user's profile picture:

```
Image(systemName: "person.circle")
    .font(.system(size: 150))
    .foregroundColor(.gray)
    .padding(.bottom, 40)
```

5. Below the preceding code block, add a Group view for the username and password fields:

```
Group{
    TextField("Username", text: $username)
    SecureField("Password", text: $password)
}
.padding()
.overlay(
    RoundedRectangle(cornerRadius: 10)
        .stroke(Color.black, lineWidth: 2)
)
```

6. Next, add a **Login** button that prints a message when someone clicks on it:

```
Button(action: {
    print("Login clicked")
}) {
    Text("Login")
    .padding()
    .background((username.isEmpty || password.isEmpty) ?
        .gray : Color.blue)
    .foregroundColor(Color.white)
    .clipShape(Capsule())
    .disabled(username.isEmpty || password.isEmpty)
}
```

7. Finally, click on `ContentView.swift` and replace the Text view with a `LoginView`:

```
struct ContentView: View {
    var body: some View {
        LoginView()
    }
}
```

The resulting preview should look like this:

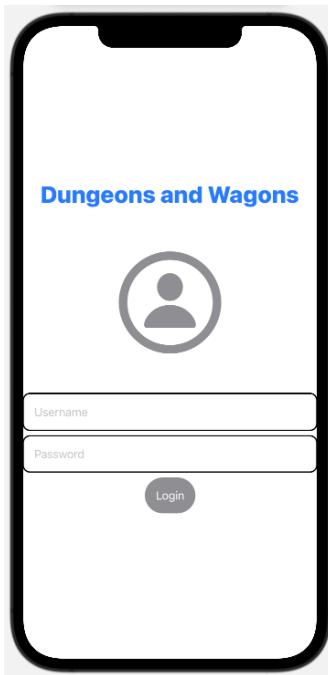


Figure 5.3 – FormFieldDisable preview

Run the app preview in the canvas and notice the **Login** button only turns blue when the username and password fields both have some content. If you'd like to see the print statement when the **Login** button is clicked, make sure to activate the debug console (**View | Debug Area | Activate Console**).

How it works...

The `.disable()` modifier can be used to deactivate a button and prevent the user from submitting a form until certain conditions have been met. The `.disable()` modifier takes a boolean parameter and disables the button when the boolean parameter's value is set to true.

In this recipe, we check for whether our `@State` variables for the username and password fields are not empty. The **Login** button stays disabled as long as one or both of these fields are empty.

We also introduce the concept of applying a modifier to multiple items at once by using a `Group` view. Our `Group` view applies the same modifiers to the username and password fields without affecting how they are displayed on the screen.

Filling out forms easily using Focus and Submit

Filling out forms can be tedious if the user has to manually click on each field, fill it out, and then click on the next field. An easier and faster way would be to use a button on the keyboard to navigate from one form field to the next.

In this recipe, we will create an address form with easy navigation between the various fields.

Getting ready

Create a new SwiftUI project named `FocusAndSubmit`.

Check the project's build settings and make sure the iOS target version is set to 15.0 or higher.

How to do it...

We add some text fields for various address fields within a `VStack` component and use `@FocusState` to navigate between them and submit the filled-out form at the end. The steps are given here:

1. Open the `ContentView.swift` file and add an enum for the fields of an address:

```
struct ContentView: View {  
    enum AddressField{  
        case streetName  
        case city  
        case state  
        case zipCode  
    }  
    ...  
}
```

2. Below the `AddressField`, add `@State` variables to hold the user's address input values:

```
@State private var streetName = ""  
@State private var city = ""  
@State private var state = ""  
@State private var zipCode = ""
```

3. Now, add a `@FocusState` variable that will keep track of which field should be in focus:

```
 @FocusState private var currentFocus: AddressField?
```

4. Add a `VStack` and a `TextField` with `.focused()`, `.textContent()`, and `.submitLabel()` modifiers:

```
 var body: some View {
    VStack{
        TextField("Address", text: $streetName)
            .focused($currentFocus, equals:
                .streetName)
            .submitLabel(.next)
        TextField("City", text: $city)
            .focused($currentFocus, equals: .city)
            .submitLabel(.next)
        TextField("State", text: $state)
            .focused($currentFocus, equals:
                .state)
            .submitLabel(.next)
        TextField("Zip code", text: $zipCode)
            .focused($currentFocus, equals:
                .zipCode)
            .submitLabel(.done)
    }
}
```

5. Add the following `.onSubmit` modifier to the `VStack`:

```
 .onSubmit {
    switch currentFocus {
    case .streetName:
        currentFocus = .city
    case .city:
        currentFocus = .state
    case .state:
        currentFocus = .zipCode
    default:
```

```
print("Thanks for providing your  
address")  
}  
}
```

Run the app in an emulator. The result should be similar to this:

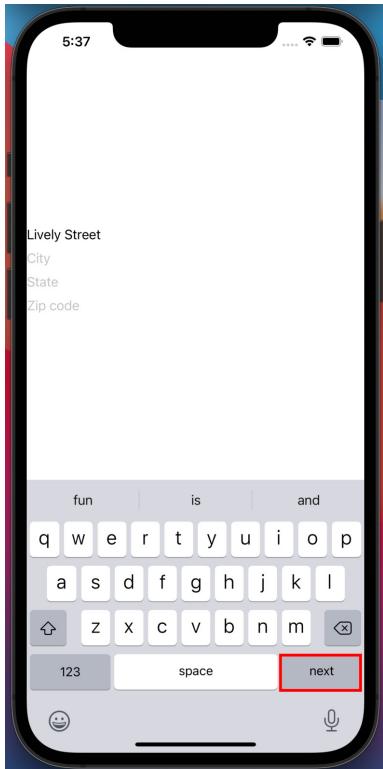


Figure 5.4 – FocusAndSubmit canvas preview

Notice that the **next** button appears on the keyboard, and you can use it to navigate to the next field easily. When you fill out the form and get to the last button, the text in the bottom-right corner of the keyboard changes from **next** to **done**. You can click **done** to run the function you'd like to run on submit.

How it works...

The Apple documentation defines `@FocusState` as a property wrapper that can read and write a value that SwiftUI updates, as the placement of focus within a scene change. We use the `@FocusState` property and the `.focused(_ : equals:)` modifier to describe which item on the scene we are currently updating. `@FocusState` is initialized as an optional argument in order to handle the situation where none of the items on the scene has been selected. Dismissing the keyboard also releases all items from focus, thereby setting its value to `nil`.

To go from one field to another without touching the screen, we add a `.submitLabel()` modifier to each of our `TextField` views such that when the user clicks **next**, we switch the focus to the next item in the list. We are thus able to navigate from the address field to **City**, **State**, and **Zip code** fields without ever leaving the keyboard. We are even able to submit the form at the end directly from the keyboard.

There's more...

You can also use the `@FocusState` boolean to show and dismiss the keyboard. For example, the following code displays some text and a button:

```
struct ContentView: View {  
    @State private var description = ""  
    @FocusState private var isFocused: Bool  
    var body: some View {  
        TextField("Enter the description", text:  
            $description)  
            .focused($isFocused)  
        Button("Hide keyboard") {  
            isFocused = false  
        }  
    }  
}
```

Here, `isFocused` is set to `true` when the user clicks on the `TextField`, causing the keyboard to be displayed. When the user clicks on **Hide keyboard**, `isFocused` is set to `false`, thereby hiding the keyboard.

See also

Apple documentation on FocusState: <https://developer.apple.com/documentation/swiftui/focusstate>

Navigating between multiple views with TabView

Navigation views are ideal for displaying hierarchical data because they allow users to drill down into data. However, navigation views don't work well with unrelated data. We use SwiftUI's TabView struct for that purpose.

Getting ready

Create a new SwiftUI iOS app named UsingTabViews.

How to do it...

We will create an app with two TabView structs. One will display the made-up best games of 2021, while the other displays various currencies used around the world. The steps are given here:

1. Create a new SwiftUI view file named HomeView:
 - a. Press *Command (⌘)+ N*.
 - b. Select **SwiftUI View**.
 - c. Click **Next**.
 - d. Enter **HomeView** in the **Save as** field.
 - e. Click **Finish**.
2. Update **HomeView.swift** to display a list of games from a string array named **games**:

```
let games = ["Doom", "Final F", "Cyberpunk",
    "avengers", "animal trivia", "sudoku", "snakes and
    ladders", "Power rangers", "ultimate frisbee",
    "football", "soccer", "much more"]
struct HomeView: View {
    var body: some View {
        NavigationView{
```

```
List {  
    ForEach(games, id: \.self){ game in  
        Text(game).padding()  
    }  
}.navigationBarTitle("Best Games for  
2021", displayMode: .inline)  
}  
}
```

The `HomeView.swift` preview should look like this:

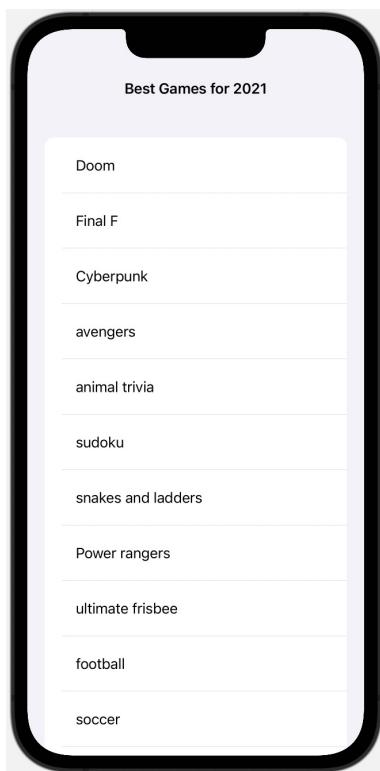


Figure 5.5 – `HomeView` preview

3. We would also like to display a list of currencies. Each currency should have an id, a name and should conform to the `Identifiable` protocol. Let's create a `Currency.swift` file that describes the fields we should expect from a currency:
 - a. Press *Command* (⌘) + *N*.
 - b. Select **Swift File**.
 - c. Click **Next**.
 - d. Enter **Currency** in the **Save as** field.
 - e. Click **Finish**.
4. In `Currency.swift`, define a `Currency` model and create an array of currencies:

```
Import Foundation
struct Currency: Identifiable {
    let id = UUID()
    var name: String
    var image: String
}

var currencies = [Currency(name: "Dollar", image:
    "dollarsign.circle.fill"),
    Currency(name: "Sterling", image:
        "sterlingsign.circle.fill"),
    Currency(name: "Euro", image:
        "eurosign.circle.fill"),
    Currency(name: "Yen", image:
        "yensign.circle.fill"),
    Currency(name: "Naira", image:
        "nairasign.circle.fill")]
```

5. Create a new SwiftUI view file named CurrenciesView:
 - a. Press **Command** (⌘) + **N**.
 - b. Select **SwiftUI View**.
 - c. Click **Next**.
 - d. Enter CurrenciesView in the **Save as** field.
 - e. Click **Finish**.
6. Update the CurrenciesView view file as follows:

```
struct CurrenciesView: View {
    var body: some View {
        NavigationView {
            VStack{
                ForEach(currencies){ currency in
                    HStack{
                        Group{
                            Text(currency.name)
                            Spacer()
                            Image(systemName:
                                currency.image)
                        }.font(Font.system(size: 40,
                            design: .default))
                        .padding()
                    }
                }
            }.navigationBarTitle("Currencies")
        }
    }
}
```

At this point, the CurrenciesView preview should look like this:



Figure 5.6 – CurrenciesView preview

7. Now that we have set up the subviews, let's open `ContentView.swift` and replace the `Text` view with a `TabView` containing our `HomeView` and `CurrenciesView`:

```
struct ContentView: View {
    var body: some View {
        TabView {
            HomeView()
                .tabItem{
                    Label("Home", systemImage:
                        "house.fill")
                }
            CurrenciesView()
                .tabItem{
                    Label("Currencies", systemImage:
                        "coloncurrencysign.circle.fill" )
                }
        }
    }
}
```

```
    }
}
```

The resulting preview should look like this:

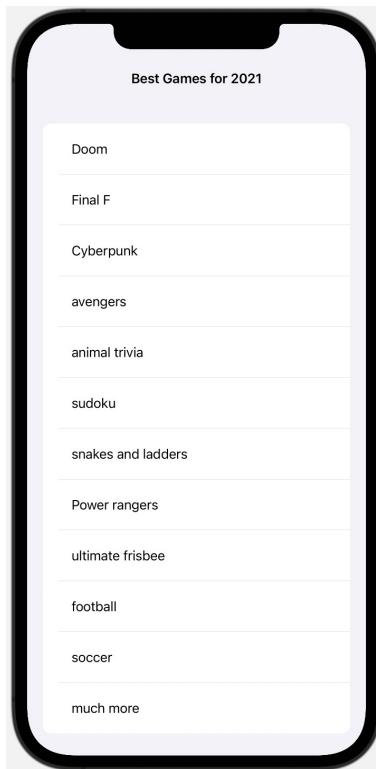


Figure 5.7 – ContentView preview

Now, run the app and click on the **Home** or **Currencies** tab to switch between them.

How it works...

In its most basic form, a `TabView` struct can hold multiple views. The following code will display two tabs without tab images:

```
TabView {
    Text("First")
    Text ("Second")
}
```

However, you probably want to display more information than simple text in each tab. The `.tabItem()` modifier can be added to each view so that it gets displayed in a new tab.

Let's begin with a discussion about the `ContentView.swift` file because this contains the main logic for displaying tabs. To display the `HomeView` and `CurrenciesView` in separate tabs, we enclose them both in a `TabView` struct and add a `.tabItem()` modifier to each of them. Finally, within each `.tabItem()` modifier, we enclose a `Label` containing a title and image for the tabs.

As for `HomeView.swift` and `ContentView.swift`, both implement concepts already studied in *Chapter 1, Using the Basic SwiftUI Views and Controls*, and *Chapter 2, Going Beyond the Single Component with Lists and Scroll Views*.

There's more...

In iOS 14 and later, SwiftUI's `TabView` can also be used as a `UIPageViewController`. You can allow swiping through multiple screens using paging dots. To implement this style, add a `.tabViewStyle(.page)` modifier to the `TabView` as follows:

```
TabView {  
} .tabViewStyle(.page)
```

Important Note

If you are using `TabView` and `NavigationView` at the same time, make sure that `TabView` is the parent and `NavigationView` is nested within `TabView`.

See also

Apple documentation on `TabView`: <https://developer.apple.com/documentation/swiftui/tabview>

Using gestures with TabView

In the preceding recipe, we learned how to switch between tabs by clicking on tab items at the bottom of the screen. However, we may also want to programmatically trigger tab switching. In this recipe, we will use a tap gesture to trigger the transition from one tab to another.

Getting ready

Create a new SwiftUI app named TabViewWithGestures.

How to do it...

We will create a TabView with two items, each containing some text that triggers a tab switch on click. The steps are given here:

1. Open the ContentView.swift file and add a @State variable to hold the value of the currently selected tab:

```
struct ContentView: View {  
    @State private var tabSelected = 0  
    ...  
}
```

2. Replace the Text view in the body variable with a TabView struct and some tabs:

```
TabView(selection: $tabSelected) {  
    Text("Left Tab. Click to switch to right")  
        .onTapGesture {  
            self.tabSelected = 1  
        }.tag(0)  
    .tabItem{  
        Label("Left", systemImage:  
            "l.circle.fill")  
    }  
    Text("Right Tab. Click to switch to left")  
        .onTapGesture {  
            self.tabSelected = 0  
        }  
    .tabItem{  
        Label("Right", systemImage:  
            "r.circle.fill")  
    }.tag(1)  
}
```

The resulting preview should look like this:

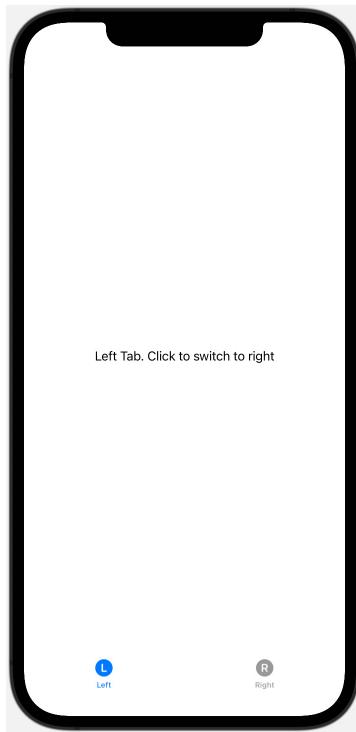


Figure 5.8 – TabViewWithGestures preview

Run the app in live preview and click on the text to switch between tabs. Notice the tab items at the screen bottom indicate which tab you're currently viewing.

How it works...

To programmatically switch between tabs, we first create a state variable, `tabSelected`, that stores the value of the currently selected tab. We then add a `selection` argument to our `TabView` struct that binds to the `tabSelected` variable. We next add a `.tag()` modifier to each of the tabs with a value that uniquely identifies each tab. Finally, we use the `.onTapGesture()` modifier to change the value of our `tabSelected` variable.

When the `tabSelected` is 0, our left tab becomes the active tab displayed on the screen, and when `tabSelected` changes to 1, our right tab becomes active.

6

Presenting Extra Information to the User

When interacting with mobile applications, users require a certain level of handholding. For example, suppose they are about to perform an irreversible action such as permanently deleting a file. In that case, they should probably be shown an alert asking for confirmation if they want to proceed with the action. Then, depending on the user's response, the file would be deleted or left alone.

Since SwiftUI is a declarative programming language, presenting extra information to the user mainly involves adding modifiers to already existing views. It is possible to add one or several such modifiers to a `View` and set the conditions for each trigger. In this chapter, we will learn how to present extra information to the user using alerts, modals, context menus, and popovers.

The recipes we'll cover in this chapter are as follows:

- Presenting alerts
- Adding actions to alert buttons
- Presenting confirmation dialogs
- Presenting new views using sheets
- Creating context menus
- Displaying popovers

Technical requirements

The code in this chapter is based on Xcode 13 and Swift iOS 15.

The code for this section can be found in the Chapter 6 folder of this book's GitHub repository: <https://github.com/PacktPublishing/SwiftUI-Cookbook-2nd-Edition/tree/main/Chapter06-Presenting-Extra-information-to-the-user>.

Presenting alerts

A common way of showing important information to users is by using alerts with a message and some buttons. In this recipe, we will create a simple alert that gets displayed when a button is pressed.

The way alerts are presented in iOS 13 and 14 has been deprecated. iOS 15 introduced a new way of presenting alerts. For the sake of being able to handle devices on earlier iOS versions, we'll go over both ways of presenting alerts, starting with iOS 15. The code for iOS 13 and 14 alerts is found in the `oldAlerts.swift` file while the newer iOS 15 version is located in `ContentView.swift`.

Getting ready

Create a new SwiftUI project called `PresentingAlerts`.

Set the project's **iOS Deployment Target** to **iOS 15.0**.

How to do it

We will add a Button or Text to the scene that can be used to display an alert containing a single button.

The iOS 15 steps are as follows:

1. Create a @State variable that will be used to trigger the presentation of our Alert view:

```
@State private var showSubmitAlert = false;
```

2. Replace the Text struct in ContentView with a Button with the text Show Alert, and a closure that sets our showSubmitAlert variable to true:

```
Button("Show Alert") {  
    showSubmitAlert = true  
}
```

3. Finally, add a .alert() modifier that displays our alert when showSubmitAlert is true:

```
.alert("Confirm Actions", isPresented:  
    $showSubmitAlert ){  
    Button("OK", role: .cancel){}  
    } message: {  
    Text("Are you sure you want to submit the form")  
}
```

The iOS 13 and 14 steps are as follows:

1. Create a new SwiftUI view called oldAlerts.swift.
2. Create a @State variable that will be used to trigger the presentation of our Alert view:

```
@State private var showSubmitAlert = false;
```

3. Replace the Text struct in the oldAlerts struct with a Button with the text `Show alert`, and whose action changes our `showSubmitAlert` variable to true:

```
Button(action: {  
    self.showSubmitAlert=true  
}) {  
    Text("Show alert")  
    .padding()  
    .background(Color.blue)  
    .foregroundColor(.white)  
    .clipShape(Capsule())  
}
```

4. Finally, add a `.alert()` modifier that displays our Alert view when `showSubmitAlert` is true:

```
.alert(isPresented: $showSubmitAlert ) {  
    Alert(title: Text("Confirm Action"),  
          message: Text("Are you sure you want to  
                      submit the form?"),  
          dismissButton: .default(Text("OK"))  
    )  
}
```

Now run the preview and click on the **Show Alert** button to display the alert:

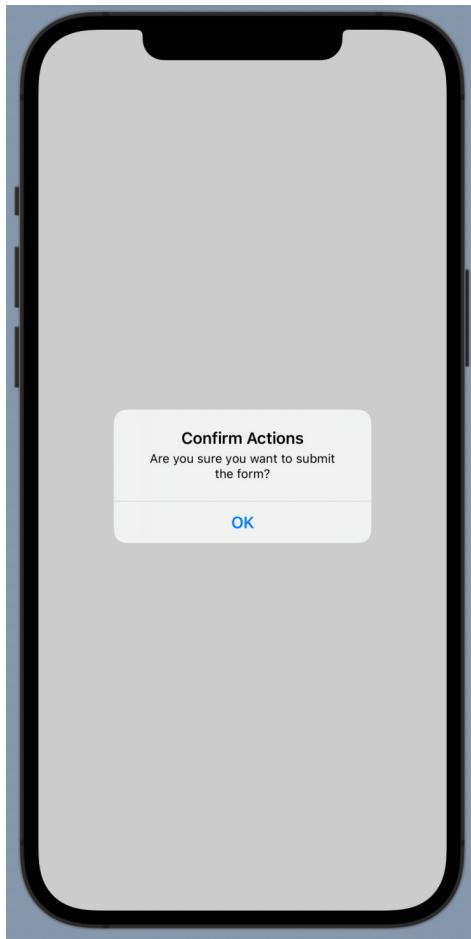


Figure 6.1 – PresentingAlerts preview

The alert displays the message we passed to our Alert view. Click on **OK** to dismiss it.

How it works

Let's look at how the two methods work individually.

iOS 15

We used a `.alert` modifier with four parameters: a `title` parameter for the title of the alert; an `isPresented` parameter that is a binding to a Boolean value that determines whether the alert is shown or hidden; and finally, two closures, `action` and `message`. The `action` parameter is a `ViewBuilder` that returns the alert's actions, while the `message` parameter is a `ViewBuilder` for displaying the message in the alert.

Notice the `.cancel` role added to the alert's **OK** button. If the `.cancel` role is assigned to a button, SwiftUI will automatically create and display one in the alert.

iOS 13 and 14

Displaying alerts is a three-step process. First, we create an `@State` variable whose value is used to trigger the showing or hiding of the alert. Then, we add a `.alert()` modifier to the view we are modifying. And finally, we embed an `Alert` view inside the `.alert()` modifier.

In this recipe, the `showSubmitAlert` state variable is passed to the modifier's `isPresented` argument. The alert gets displayed when `showSubmitAlert` is set to `true`. When the user clicks on the **OK** button in the `Alert` view, its value changes back to `false`, and the alert gets hidden once more.

We used an `Alert` view with three parameters, the alert `title`, a message, and a `Button` that dismissed the alert when clicked.

See also

Types of alert buttons: <https://developer.apple.com/documentation/swiftui/alert/button>

Apple Human Interface guidelines regarding alerts: <https://developer.apple.com/design/human-interface-guidelines/macos/windows-and-views/alerts>

Adding actions to alert buttons

We may want to display alerts with more than just an **OK** button to confirm the alert has been read. In some cases, we may wish to present a **Yes** or **No** choice to the user. For example, if the user wants to delete an item in a list, we may wish to present an alert that provides the option of whether to proceed with the deletion or cancel the action.

In this recipe, we will look at how to add multiple buttons to our `Alert`. We will provide the descriptions for iOS 15 alerts. You can find the code for iOS 13 and 14 alerts in the `olderAlertsWithActions.swift` file.

Getting ready

Create a new SwiftUI app called `AlertsWithActions`.

How to do it

We will implement an alert with two buttons and an action. The alert will get triggered by a tap gesture. When triggered, the alert displays a message asking the user if they want to change the text currently being displayed. A click on **OK** changes the text, while a click on **Cancel** leaves the text unchanged.

The steps for iOS 15 are as follows:

1. Create a `@State` variable that holds the on/off state of our alert. Also, add a variable for the text. Place the variable just below the `ContentView` struct declaration:

```
@State private var changeText = false  
@State private var displayedText = "Tap to Change Text"
```

2. Replace the original `Text` view with a `Button` that displays text from our `displayedText` variable. When clicked, the button should set our `changeText` variable to `true`:

```
Button(displayedText) {  
    changeText = true  
}
```

3. Add a `.alert()` modifier containing two buttons, one for changing the displayed text and the other that does nothing but displays some text in a button:

```
.alert(alertTitle, isPresented: $changeText) {  
    Button("Yea") {  
        displayedText = (displayedText == "Stay  
        Foolish") ? "Stay Hungry" : "Stay Foolish"  
    }  
    Button("Do nothing") {}  
}message: {  
    Text("Do you want to toggle the text?")  
}
```

Run the app in preview mode and click on the **Tap to Change Text** button. The result should look as follows:



Figure 6.2 – AlertsWithActions Preview

A click on the **Yes** alert button toggles the displayed text between **Stay Foolish** and **Stay Hungry**. The **Cancel** and **Do nothing** buttons do nothing but dismiss the alert.

How it works

We explained how to use `.alert()` in the previous recipe. In this recipe, we introduced alerts with multiple buttons. A click on the **Tap to Change Text** button changes the value of the `changeText` variable to `true`, thereby triggering the display of the alert.

We defined an alert with two buttons, **Yes** and **Do nothing**, but three buttons were displayed in the SwiftUI preview. This is because SwiftUI automatically adds a button with the `.cancel` role if none is provided.

Presenting confirmation dialogs

The SwiftUI confirmation dialogs and alerts are both used to request additional information from the user by interrupting the normal flow of the app to display a message. Confirmation dialogs give the user some additional choices related to an action that they are currently taking, whereas the `Alerts` view informs the user if something unexpected happens or they are about to perform an irreversible action.

Confirmation dialogs were introduced in iOS 15. The similar but deprecated functionality in iOS 13 and 14 is called `Actionsheet`. The implementation for `Actionsheet` is located in the `oldActionSheets.swift` file.

In this recipe, we will create a confirmation dialog that gets displayed when the user taps some text in the view.

Getting ready

Create a new SwiftUI project called `PresentingConfirmationDialogs`.

How to do it

We will implement an action sheet by adding a `.confirmationDialog()` modifier to a view as well as a trigger that changes the value of a variable, which is used to determine whether the dialog is shown or not. The steps are as follows:

1. Open the `ContentView.swift` file and add a `@State` variable, `showDialog`, that triggers the display/hiding of the alert. Also, add a variable for the alert title:

```
@State private var showDialog = false  
var title = "Confirmation Dialog"
```

2. Add a Button that changes our showDialog variable to true when clicked:

```
Button("Present Confirmation Dialog") {
    showDialog = true
}
```

3. Add a .confirmationDialog() modifier to the Button:

```
.confirmationDialog(title, isPresented: $showDialog) {
    Button("Dismiss Dialog", role: .destructive) {
        // Add dismiss dialog action here
    }
    Button("Save") {
        // Add Save Action Here
    }
    Button("Cancel", role: .cancel){}
    Button("Print something to console") {
        print("Print button clicked")
    }
}message: {
    Text """
        Use Dialogs to give users alternatives for
        completing a task
        """
}
```

Now run the app preview and click on the **Present Confirmation Dialog** button.
The result should look as follows:

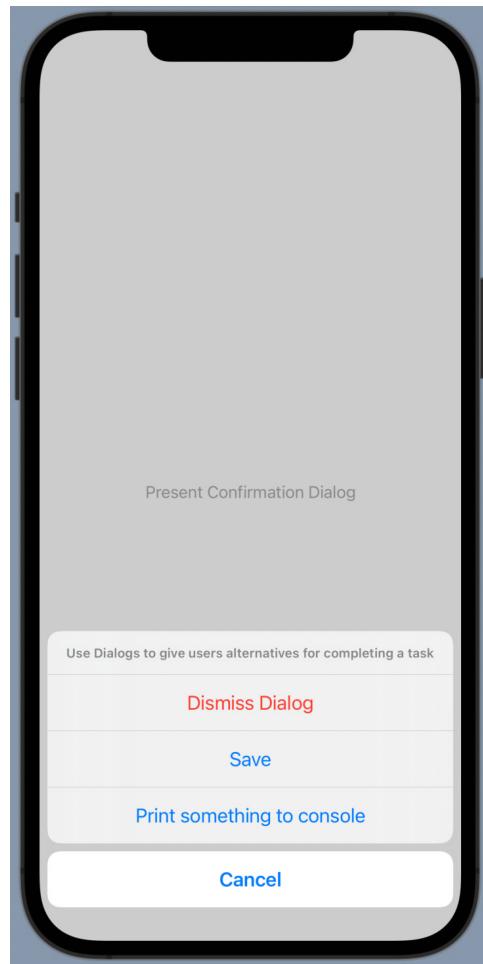


Figure 6.3 – PresentingConfirmationDialog Preview

You can click on **Cancel** to dismiss the dialog. If running on an emulator or a physical device, click the **Print something to console** button to execute the print action related to that button and view the print statement in Xcode's debug area.

How it works

In this recipe, we used a confirmation dialog with four parameters. The `title` parameter is a string that describes the title of the dialog. The `isPresented` parameter is a binding to a Boolean that determines whether to display or hide the dialog. The `action` parameter is a `ViewBuilder` that returns the dialog's actions. Finally, the `message` parameter is a view builder that returns the message to be displayed in the dialog.

We implement all the buttons for our dialog within the `action` parameter. Each button should have a `title` and an `action`. Since these are regular SwiftUI buttons, you can also add roles such as `.destructive` to have the system style them appropriately.

See also

Apple documentation on confirmation dialogs: [https://developer.apple.com/documentation/swiftui/view/confirmationdialog\(_:ispresented:-titlevisibility:presenting:actions:message:\)](https://developer.apple.com/documentation/swiftui/view/confirmationdialog(_:ispresented:-titlevisibility:presenting:actions:message:)) -8y541

Presenting new views using sheets

SwiftUI's sheets are used to present new views over existing ones while allowing the user to dismiss the new view by dragging it down.

In this recipe, we will learn how to present a sheet to the user and add navigation items to the sheet.

Getting ready

Create a new SwiftUI project named `PresentingSheets`.

How to do it

We shall create two SwiftUI views named `SheetView` and `SheetWithNavController` that will be displayed modally when a button is clicked. The steps are as follows:

1. In `ContentView.swift`, between the `struct` declaration and the `body` variable, add two state variables, `showSheet` and `sheetWithNav`. The state variables will be used to trigger the sheet presentation:

```
@State private var showSheet = false  
@State private var sheetWithNav = false;
```

2. Add a `VStack` and a `Button` that changes the value of our `showSheet` variable to `true` when clicked. Add a `.sheet()` modifier to the button that gets displayed when `showSheet` is set to `true`:

```
 VStack {  
     Button("Display Sheet") {  
         self.showSheet = true  
     }.sheet(isPresented: $showSheet) {  
         SheetView()  
     }  
 }
```

3. Now let's create the `SheetView()` SwiftUI view mentioned in the `Button` struct:
 - a. Create a new SwiftUI view by pressing the *Command* (⌘) + *N* keys.
 - b. Select **SwiftUI View** and click **Next**.
 - c. In the **Save As** field, enter `SheetView`.
 - d. Click **Create**.
4. In `SheetView.swift`, change the `Text` struct to display "This is the sheet we want to display View":

```
 var body: some View {  
     Text("This is the sheet we want to display View")  
 }
```

5. Now navigate back to the `ContentView.swift` file and run the canvas Live Preview. Click on the **Display Sheet** button to present the sheet. The preview should be as follows:

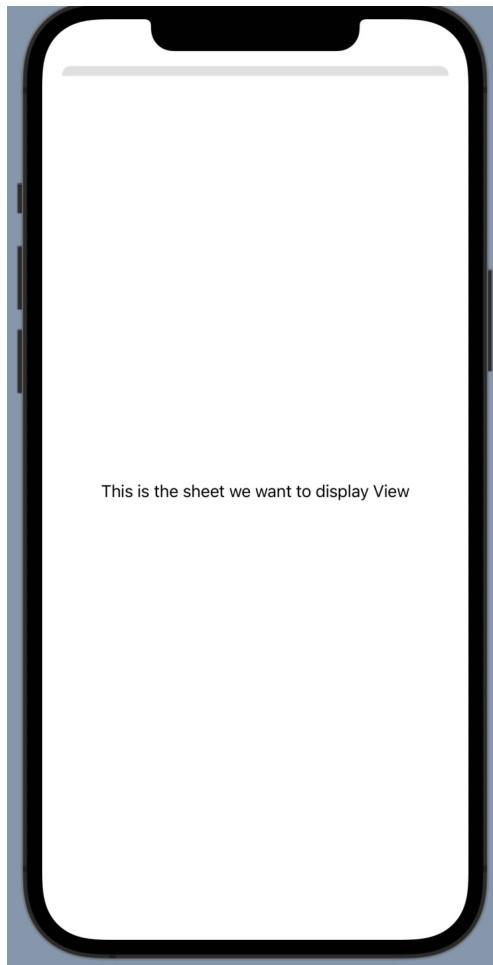


Figure 6.4 – Sheet without Navigation Preview

Swipe down to dismiss the sheet.

6. Now let's add a `Button` struct and `.sheet()` modifier to display a modal with navigation items:

```
Button("SheetWithNavigationBar") {  
    self.sheetWithNav = true  
} .sheet(isPresented: $sheetWithNav) {
```

```
    ModalWithNavController(sheetWithNav: self.$sheetWithNav)
}
```

7. Let's create the `SheetWithNavController()` SwiftUI view mentioned in the `Button` struct:
 - a. Create a new SwiftUI view by pressing the *Command* (⌘) + *N* keys.
 - b. Select **SwiftUI View** and click **Next**.
 - c. In the **Save As** field, enter `SheetWithNavController`.
 - d. Click **Create**.
8. In the `SheetWithNavController` struct, add a `@Binding` variable named `sheetWithNav`. Also add a `NavigationView` to the body of the struct:

```
struct SheetWithNavController: View {
    @Binding var sheetWithNav: Bool
    var body: some View {
        NavigationView{
            Text("Sheet with navigation")
                .navigationBarTitle(Text("Sheet title"), displayMode: .inline)
                .navigationBarItems(trailing:
                    Button(action: {
                        self.sheetWithNav = false;
                    }) {
                        Text("Done")
                    }
                )
        }
    }
}
```

9. We can't build or run the code at this point because we need to pass a value to the `SheetWithNavController()` located in the preview. To fix the problem, update the view to pass in a binding with an immutable value of `true`:

```
static var previews: some View {
    SheetWithNavController(sheetWithNav: .constant(true))
}
```

10. Finally, navigate back to `ContentView.swift` and run the canvas Live Preview. A click on the `SheetWithNavigationBar` button should display the following:



Figure 6.5 – Sheet with Navigation Preview

You can swipe down or click **Done** to dismiss the sheet.

How it works

To present the sheet, we create an `@State` variable, `showSheet`, that tracks whether the sheet should be displayed or not. Next, we create a button that changes the `showSheet` variable to `true` when clicked. Finally, we add the `.sheet()`, whose `isPresented` parameter takes our state variable and displays a sheet when its value is set to `true`.

The second sheet follows a similar pattern to the first sheet but takes `@Binding` as a parameter. To find out why we use the binding, let's take a look at the view displayed by the second sheet, `SheetWithNavController()`.

The `SheetWithNavController` struct has a `@Binding` property, where we usually use a `@State` property:

```
@Binding var sheetWithNav: Bool
```

The `@Binding` property lets us declare that a value comes from elsewhere and should be shared between both places. In this case, we want to share the value of the `sheetWithNav` variable declared in `ContentView.swift`. Changing the value in `SheetWithNavController` struct should change the value in the `ContentView` struct and vice versa.

The `SheetWithNavController` struct's body variable contains a `NavigationView` struct and a `Text` struct with two modifiers. The `.navigationBarTitle()` modifier adds the title to the view, while the `.navigationBarItems()` modifier adds a trailing button that sets our binding variable, `sheetWithNav`, back to `false` when clicked.

When clicked, the trailing button sets the `SheetWithNavController` variable to `false`, thereby dismissing the sheet.

Finally, the `SheetWithNavController` preview expects a `@Binding` for the preview to run. The `SheetWithNavController(sheetWithNav: true)` statement will cause an error because the preview expects a binding to be passed to the view. This can be solved by passing in a binding with an immutable value, `.constant(true)`. Your resulting preview statement should be as follows:

```
SheetWithNavController(sheetWithNav: .constant(true))
```

See also

Apple Human Interface guidelines regarding sheets: <https://developer.apple.com/design/human-interface-guidelines/macos/windows-and-views/sheets/>

Creating context menus

A context menu is a pop-up menu used to display actions that the developer anticipates the user might want to take. SwiftUI context menus are triggered using 3D Touch on iOS and a right-click on macOS.

Context menus consist of a collection of buttons displayed horizontally in an implicit `HStack`.

In this recipe, we will create a context menu to change the color of an SF symbol.

Getting ready

Create a new SwiftUI project named `DisplayingContextMenu`s.

How to do it

We will display a light bulb in our view and change its color using a context menu. To achieve this, we'll need to create an `@State` variable to hold the current color of the bulb and change its value within the context menu. The steps are as follows:

1. Just above the `body` variable in `ContentView.swift`, add an `@State` variable to hold the color of the bulb. Initialize it to `red`:

```
@State private var bulbColor = Color.red
```

2. Within the `body` variable, change the `Text` struct to an `Image` struct that displays a light bulb from SF Symbols:

```
Image(systemName: "lightbulb.fill")
```

3. Add `.font()` and `.foregroundColor()` modifiers to the image. Change the font size to 60 and the foreground color to our `bulbColor` variable:

```
.font(.system(size: 60))
.foregroundColor(bulbColor)
```

4. Add a `.contextMenu()` modifier with three buttons. Each `Button` changes the value of our `bulbColor` variable to a new color:

```
.contextMenu{
    Button("Yellow Bulb") {
        self.bulbColor = Color.yellow
    }
}
```

```
Button("Blue Bulb") {  
    self.bulbColor = Color.blue  
}  
Button("Green Bulb") {  
    self.bulbColor = Color.green  
}  
}
```

Run the live preview and long press on the light bulb. A context menu gets displayed as seen in the following figure:



Figure 6.6 – DisplayingContextMenus live preview

You can click on any of the options in the context menu to change the light bulb color.

How it works

The `.contextMenu()` modifier can be attached to any view. In this recipe, we attached it to an `Image` view such that a long press on the view displays the context menu. The `.contextMenu()` modifier contains buttons whose action closure performs a particular function. In our case, each `Button` changes our `bulbColor @State` variable to a new color, thereby updating our view.

See also

Apple documentation on context menu: [https://developer.apple.com/documentation/swiftui/view/contextmenu\(menuitems:\)](https://developer.apple.com/documentation/swiftui/view/contextmenu(menuitems:))

Displaying popovers

A popover is a view that can be displayed on screen to provide more information about a particular item. They include an arrow pointing to the location where they originate from. You can tap on any other screen area to dismiss the popover. Popovers are typically used on larger screens such as iPads.

In this recipe, we will create and display a popover on an iPad.

Getting ready

Create a new SwiftUI project named `DisplayingPopovers`.

How to do it

Following the pattern we've used so far in this chapter, we shall first create a `@State` variable whose value triggers the displaying or hiding of a popover. Then we add a `.popover()` modifier that displays the popover when the `@State` variable is `true`. The steps are as follows:

1. Just above the body variable in the `ContentView.swift` file, add a `state` variable that will be used to trigger the display of the popover:

```
@State private var showPopover = false  
}
```

2. Within the body variable, replace the Text struct with a Button that changes the value of showPopover to true when clicked:

```
Button(action: {  
    self.showPopover = true  
}) {  
    Text("Show popover").font(.system(size: 40))  
}
```

3. Add the .popover() modifier to the end of the Button:

```
.popover(  
    isPresented: self.$showPopover,  
    arrowEdge: .bottom  
) {  
    Text("Popover content displayed here")  
    .font(.system(size: 40))  
}
```

Since popovers should be used on iPads, let's change the canvas preview to display an iPad. Click on the active scheme button located in the Xcode toolbar area, highlighted in the following screenshot:



Figure 6.7 – Set the preview with the active scheme button on Xcode

4. Select any of the iPads in the list (or add a simulator if no iPads are in the list):



Figure 6.8 – List of Xcode simulators

Run the code in the canvas Live Preview and click on the **Show popover** button to display the popover. The resulting view should be as follows:

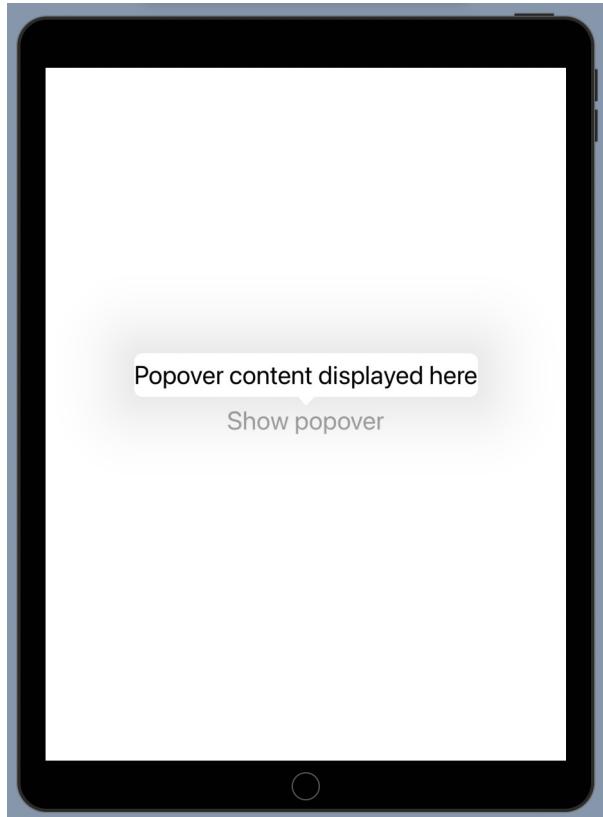


Figure 6.9 – DisplayingPopovers Preview

Click on any other area of the screen to hide the popover.

How it works

A `.popover()` modifier takes four possible arguments, although three were used here. Possible arguments are `isPresented`, `attachmentAnchor`, `arrowEdge`, and `content`.

The `isPresented` argument takes a binding that's used to trigger the display of the popover, `showPopover`.

We did not use an attachment anchor in this recipe. `attachmentAnchor` is a positioning anchor that defines where the popover is attached.

The `arrowEdge` argument represents where the popover arrow should be displayed. It takes four possible values, `.bottom`, `.top`, `.leading`, and `.trailing`. We used the `.bottom` value to display our arrow edge at the bottom of the popover, just above our text.

The `content` argument is a closure that returns the content of the popover. In this recipe, a `Text` struct was used as content, but we could also use a separate SwiftUI view in the `content` closure, such as an image or a view from another SwiftUI View file.

See also

Apple Human Interface guidelines regarding popovers: <https://developer.apple.com/design/human-interface-guidelines/ios/views/popovers/>

7

Drawing with SwiftUI

One of the strongest points of SwiftUI is that all the components are uniform and that they can be used in an interchangeable and mixed way, whereas in UIKit, intermixing labels, buttons, and custom shapes was a bit cumbersome. In this chapter, we'll learn how to use the basic shapes offered out of the box by SwiftUI and how to create new shapes using the `Path` class. We'll learn how simple and natural it is to deal with, extend, and use custom shapes with standard components such as text and sliders.

By the end of the chapter, you'll be able to create a view from a custom path, add a gradient to fill a custom view, and will know how to write a Tic-Tac-Toe game using basic shapes.

In this chapter, we will cover the following recipes:

- Using SwiftUI's built-in shapes
- Drawing a custom shape
- Drawing a curved custom shape
- Drawing using the Canvas API
- Implementing a progress ring

- Implementing a Tic-Tac-Toe game in SwiftUI
- Rendering a gradient view in SwiftUI
- Building a bar chart
- Building a pie chart

Technical requirements

The code in this chapter is based on Xcode 13 and iOS 15.

All the code examples for this chapter can be found on GitHub at <https://github.com/PacktPublishing/SwiftUI-Cookbook-2nd-Edition-/tree/main/Chapter07-Drawing-with-SwiftUI>.

Using SwiftUI's built-in shapes

SwiftUI provides five different basic shapes:

- Rectangle
- RoundedRectangle
- Capsule
- Circle
- Ellipse

They can be used to create more complex shapes if we combine them.

In this recipe, we'll explore how to create them, add a border and a fill, and how to lay out the shapes.

There will be more than what we can show here, but with this recipe as a starting point, you can modify the shapes to discover the potential of the built-in shapes of SwiftUI.

Getting ready

As usual, start by creating a new SwiftUI project with Xcode and call it `BuiltInShapes`.

How to do it...

We are going to implement a simple app that shows the different basic shapes laid out vertically:

1. Create a VStack component with a spacing of 10 and a horizontal padding of 20:

```
struct ContentView: View {  
    var body: some View {  
        VStack(spacing: 10) {  
            }  
            .padding(.horizontal, 20)  
        }  
    }
```

2. Add the shapes inside the stack:

```
VStack(spacing: 10) {  
    Rectangle()  
        .stroke(.orange,  
                lineWidth: 15)  
    RoundedRectangle(cornerRadius: 20)  
        .fill(.red)  
    Capsule(style: .continuous)  
        .fill(.green)  
        .frame(height: 100)  
    Capsule(style: .circular)  
        .fill(.yellow)  
        .frame(height: 100)  
    Circle()  
        .strokeBorder(.blue, lineWidth: 15)  
    Ellipse()  
        .fill(.purple)  
}
```

This renders the following preview:

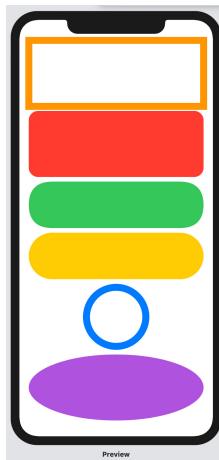


Figure 7.1 – SwiftUI's built-in shapes

How it works...

Thanks to SwiftUI and its declarative syntax, the code is self-explanatory, but there are a couple of notes to make. Firstly, a capsule can have two types of curvatures for the rounded corners:

- Continuous
- Circular

In the following diagram, you can see the difference between the corners of the two capsules when taken in isolation:



Figure 7.2 – Capsule corners style

The shape at the top is a capsule with a continuous style, while the capsule at the bottom has a circular style. In the case of the circular style, the sides of the capsule are two perfect semi-circles, while with the continuous style, the corners smoothly transition from a line to a curve.

The other thing to note is that the rectangles are closer to each other than the other shapes: the gap is narrower than the one between the circle and the ellipse. The difference lies in the way the borders are applied to the shapes.

The rectangle uses the `.stroke()` modifier, which creates a border centered in the frame, as shown in the following diagram:



Figure 7.3 – stroke applied to a shape

On the other hand, the circle uses the `.strokeBorder()` modifier, which creates a border contained inside the frame:



Figure 7.4 – strokeBorder applied to a shape

Depending on where you want to lay the shapes down, you can use either one of the `View` modifiers.

Drawing a custom shape

SwiftUI's drawing functionality permits more than just using the built-in shapes: creating a custom shape is just a matter of creating a `Path` component with the various components and then wrapping it in a `Shape` object.

In this recipe, we will work through the basics of custom shape creation by implementing a simple rhombus, which is a geometric shape with four equal, straight sides that resembles a diamond.

Getting ready

Create a new single-view app with SwiftUI called `RhombusApp`.

How to do it...

As we mentioned in the introduction, we are going to implement a `Shape` object that defines the way our custom view must be visualized:

1. Let's add a `Rhombus` struct:

```
struct Rhombus: Shape {  
    func path(in rect: CGRect) -> Path {  
        Path() { path in  
            path.move(to: CGPoint(x: rect.midX,  
                                  y: rect.minY))  
            path.addLine(to: CGPoint(x: rect.maxX,  
                                    y: rect.midY))  
            path.addLine(to: CGPoint(x: rect.midX,  
                                    y: rect maxY))  
            path.addLine(to: CGPoint(x: rect.minX,  
                                    y: rect.midY))  
            path.closeSubpath()  
        }  
    }  
}
```

2. Use the `Rhombus` struct inside the body of the `ContentView` struct:

```
struct ContentView: View {  
    var body: some View {  
        Rhombus()  
            .fill(.orange)  
            .aspectRatio(0.7,  
                        contentMode: .fit)  
            .padding(.horizontal, 10)  
    }  
}
```

A nice orange rhombus will be displayed in the preview:

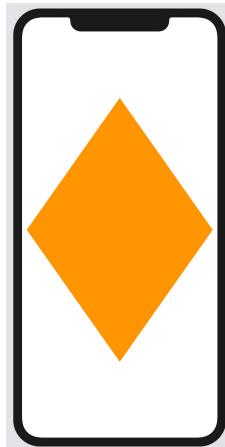


Figure 7.5 – A rhombus as a custom SwiftUI shape

How it works...

To create a custom shape in SwiftUI, our class must conform to the `Shape` protocol, whose only function is `path(in: CGRect)`. This function provides a rectangle that represents the frame where the shape will be rendered as a parameter and returns a `Path` struct, which is the description of the outline of the shape.

The `Path` struct has several drawing primitives that permit us to move the point, add lines and arcs, and so on. This allows us to define the outline of the shape.

Using a `Path` object resembles a bit of the old educative language known as **Logo**, where you are in charge of guiding a turtle that draws a line on the screen, and you can only give it simple commands: go to location A, draw a line until location B, draw an arc up to location C, and so on.

Note that the coordinates system of SwiftUI has its origin in the top-left corner of the screen, with the *x*-axis from left to right, and the *y*-axis from top to bottom.

The following diagram shows the coordinates system and the shortcuts defined in iOS for the most important points:

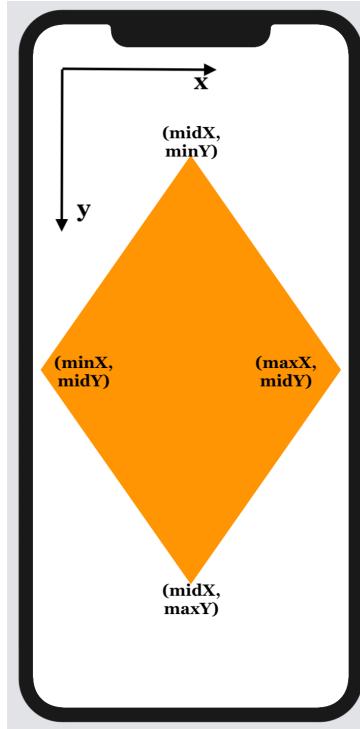


Figure 7.6 – SwiftUI coordinates system

As mentioned in the introduction, the `rect` instance, which is passed as a parameter, is the container frame. By exploiting a few of the convenient properties it exposes, we initially set our starting point in the top vertex of the rhombus, from where we added lines in an anti-clockwise direction connecting the other three corners. Finally, we closed the path, which creates a complete line from the last point back to the first point.

SwiftUI recognizes the closed shape and intuitively knows the space that needs to be filled inside the boundaries of the shape.

Drawing a curved custom shape

Following on from the *Drawing a custom shape* recipe, what if we want to define a shape that is made up not only of straight lines but also has a curved line in it? In this recipe, we'll build a heart-shaped component using the arc and curve primitives of `Path`.

Getting ready

As usual, create a SwiftUI app called `Heart`.

How to do it...

We are going to follow the same steps we implemented in the *Drawing a custom shape* recipe, adding curves and an arc from one point to another.

Since the control points of a heart shape are in the midpoint of each side, we must add some convenient properties to the `CGRect` struct.

Let's do this by performing the following steps:

1. Let's add the properties that will return the coordinates from each of the quarters:

```
extension CGRect {
    var quarterX: CGFloat {
        minX + size.height/4
    }
    var quarterY: CGFloat {
        minY + size.height/4
    }
    var threeQuartersX: CGFloat {
        minX + 3*size.height/4
    }
    var threeQuartersY: CGFloat {
        minY + 3*size.height/4
    }
}
```

2. Implement the `Heart` shape, starting with the definition of the `Path` object:

```
struct Heart: Shape {
    func path(in rect: CGRect) -> Path {
        Path { path in
            //...
        }
    }
}
```


6. To finish the shape, we must close Path:

```
path.closeSubpath()
```

7. Finally, the Heart shape is ready to be added to the body, where we will use a red fill and add an orange border:

```
struct ContentView: View {  
    var body: some View {  
        Heart()  
            .fill(.red)  
            .overlay(Heart()  
                .stroke(.orange, lineWidth: 10))  
            .aspectRatio(contentMode: .fit)  
            .padding(.horizontal, 20)  
    }  
}
```

The following screenshot shows a nice heart shape as a result of our code:

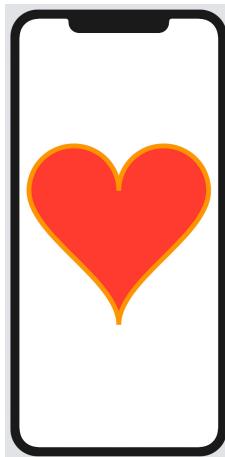


Figure 7.7 – Custom heart-shaped component

How it works...

Like in the *Drawing a custom shape* recipe, we initially set the starting point to be the top tip of the shape, and then added the curves and arcs clockwise. You can see that the `arc()` function has a `clockwise` parameter, which is set to `false` in our example. However, the arcs are drawn in a clockwise direction: how is this possible?

The thing is that SwiftUI, like UIView, uses a flipped coordinate system, so clockwise for SwiftUI means counterclockwise for the user, and vice versa. You can imagine a flipped coordinate system as a system with the origin in the top-left corner and the y -axis pointing to the bottom, so if it is reverted to the original position, you can see how a clockwise movement is then clockwise for the observer too.

The following diagram should help you visualize this:

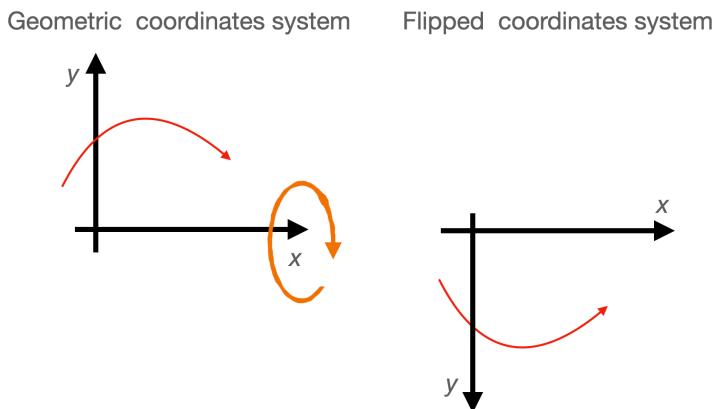


Figure 7.8 – The clockwise direction in the geometric and flipped coordinate systems

Drawing using the Canvas API

One of the most powerful features of UIKit is the possibility of creating a subclass of `UIView` where we can draw directly into the graphic context using Core Graphic Framework's functions.

SwiftUI implemented this functionality similarly via the `Canvas View`, where we can draw directly in the Core Graphic's context.

In this recipe, will explore the `Canvas View` by implementing a simple app to draw using our finger.

Getting ready

Let's implement a SwiftUI app called `Drawing`.

How to do it...

The app will use a struct as a model to save the touches. Then, each model will be used to draw a line in a Canvas View. To do this, follow these steps:

1. First, create the Line struct model and add a @State property to the main View:

```
struct Line {  
    var points: [CGPoint]  
}  
  
struct ContentView: View {  
    @State var lines: [Line] = []  
  
    var body: some View {  
  
    }  
}
```

2. In the body of the ContentView struct, we will add a Canvas View. In its content block, we will iterate for each entry in the model:

```
var body: some View {  
    Canvas { ctx, size in  
        for line in lines {  
            var path = Path()  
            path.addLines(line.points)  
  
            ctx.stroke(path, with: .color(.red),  
                       style: StrokeStyle  
                           (lineWidth: 5,  
                            lineCap: .round,  
                            lineJoin: .round))  
        }  
    }  
}
```

3. To register the finger movements, we will add a `Gesture` modifier that updates the model:

```
Canvas { ctx, size in
    //...
}
.gesture(DragGesture(minimumDistance: 0,
                      coordinateSpace: .local)
    .onChanged { value in
        let position = value.location
        if value.translation == .zero {
            lines.append(Line(points: [position]))
        } else {
            guard let lastIdx = lines.indices.last else {
                return
            }
            lines[lastIdx].points.append(position)
        }
    })
})
```

With the gesture code, the app is ready and we can draw our masterpieces:

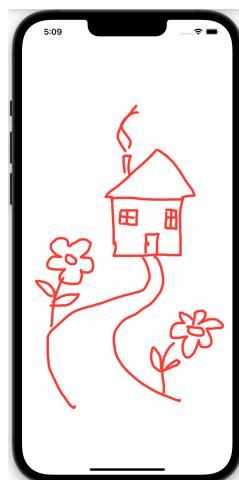


Figure 7.9 – Drawing with a finger in action

Of course, this app isn't close to a professional drawing app, such as Procreate or Photoshop. However, we should appreciate the fact that with just a few lines of code, we can reach more than decent results.

How it works...

There are two main parts to this app:

- **Drawing in a View starting from an array of points:** As we mentioned in the introduction, `Canvas` provides a context where we can draw using the `Core Graphics` function. In this case, it is just a matter of creating a `Path` from the points and then drawing it using the `.stroke()` modifier.
- **Create an array of points for each finger movement:** The `DrawGesture` object listens for changes in the touches and movement. If `translation` of the movement is zero, this means that the touch has started, so a new line must be added to the model. If `translation` is greater than zero, the finger is moved from the previous position. Then, a new point must be added to the last element of the model.

There's more...

We have just seen how simple it is to draw in a `GraphicsContext` object, so our simple app could be made more sophisticated. For example, we could add a menu with a list of buttons to change the color of the background, or another menu to change the color of the drawing line.

Implementing a progress ring

Admit it – since you bought your Apple Watch, you are getting fitter, aren't you? I bet it is because of the activity rings you want to close every day. Am I right?

In this recipe, we'll implement a progress ring similar to those on the Apple Watch, and we'll change its value via some sliders.

Getting ready

You don't need to prepare anything for this recipe; just create a SwiftUI project called `ProgressRings`.

How to do it...

We are going to implement two components:

- A single `ProgressRing` View
- A composite `RingsView`

We'll add the latter to `ContentView`, and we'll simulate progress using three sliders.

To do this, follow these steps:

1. Implement a ring view using a `Shape` object and an arc:

```
struct ProgressRing: Shape {
    private let startAngle = Angle.radians(1.5 * .pi)

    @Binding
    var progress: Double

    func path(in rect: CGRect) -> Path {
        Path() { path in
            path.addArc(
                center: CGPoint(x: rect.midX,
                                y: rect.midY),
                radius: rect.width / 2,
                startAngle: startAngle,
                endAngle: startAngle +
                    Angle(
                        radians: 2 * .pi * progress),
                clockwise: false
            )
        }
    }
}
```

2. Create a `RingsView` view with the definition of some common properties and some `@State` properties to hold the state of each ring:

```
struct ProgressRingsView: View {
    private let ringPadding = 5.0
    private let ringWidth = 40.0
```

```
private var ringStrokeStyle: StrokeStyle {
    StrokeStyle(lineWidth: ringWidth,
                lineCap: .round,
                lineJoin: .round)
}

@Binding
var progressExternal: Double
@Binding
var progressCentral: Double
@Binding
var progressInternal: Double

var body: some View {
}
```

3. The body area of the View presents the three rings, one on top of each other in a ZStack View. Add a ZStack with the three components:

```
var body: some View {
    ZStack {
        ProgressRing(progress: $progressInternal)
            .stroke(.blue,
                    style: ringStrokeStyle)
            .padding(2*(ringWidth + ringPadding))
        ProgressRing(progress: $progressCentral)
            .stroke(.red,
                    style: ringStrokeStyle)
            .padding(ringWidth + ringPadding)
        ProgressRing(progress: $progressExternal)
            .stroke(.green,
                    style: ringStrokeStyle)

    }
    .padding(ringWidth)
}
```

4. Finally, add `RingsView` and three sliders to the `ContentView` struct:

```
struct ContentView: View {  
    @State  
    private var progressExternal = 0.3  
    @State  
    private var progressCentral = 0.7  
    @State  
    private var progressInternal = 0.5  
  
    var body: some View {  
  
        ZStack {  
            ProgressRingsView(progressExternal:  
                $progressExternal,  
                progressCentral:  
                    $progressCentral,  
                progressInternal:  
                    $progressInternal)  
            .aspectRatio(contentMode: .fit)  
            VStack(spacing: 10) {  
                Spacer()  
                Slider(value: $progressInternal,  
                    in: 0...1, step: 0.01)  
                Slider(value: $progressCentral,  
                    in: 0...1, step: 0.01)  
                Slider(value: $progressExternal,  
                    in: 0...1, step: 0.01)  
            }  
            .padding(30)  
        }  
    }  
}
```

Previewing the app in Xcode, we can play with the rings and even close all of them without doing any physical activities!

You can see these rings in the following screenshot:



Figure 7.10 – Our three progress rings

How it works...

ProgressRing is a simple Path object with an arc.

The origin of an arc in Path is from the horizontal right axis. Since we want to make the ring start at the vertical axis, we set our initial arc to $3/2 * \pi$ because the angle grows clockwise. Even though the direction of the angle is clockwise, you will notice that we set `false` in the parameter of the arc; this is because SwiftUI has a flipped coordinate system, as discussed in the *Drawing a curved custom shape* recipe, where the *y*-axis points downward instead of upward, so the clock's direction is inverted.

The various progress variables are decorated with `@Binding` in the ProgressRing and RingsView components because they are injected and controlled by an external class. In ContentView, they are decorated with `@State` because they are mutated in the same component by the sliders.

When we change the value of a slider, the mutation is reflected in the rendering of ContentView and then mirrored down to the child components that update the arc's length.

Implementing a Tic-Tac-Toe game in SwiftUI

SwiftUI's drawing primitives are powerful, and it is even possible to implement a game using just these. In this recipe, we'll learn how to build a simple touchable and playable Tic-Tac-Toe game, in which the game alternates between inserting a cross and a nought every time you put your finger on a cell of the board.

For those who are unfamiliar with the game, Tic-Tac-Toe is a paper-and-pencil game where two players take turns to mark either a cross or a circle (also called a *nought*) in a 3x3 grid. The player who can place three of their marks in a line horizontally, vertically, or diagonally wins.

Getting ready

For this recipe, we don't need any external resources, so it is enough just to create a SwiftUI project in Xcode called TicTacToe to hit the ground running.

How to do it...

As you may imagine, Tic-Tac-Toe is composed of three components:

- The game grid
- A nought (circle)
- A cross

Using SwiftUI, we can model these components and split each one in two: a shape that renders the design and a view that manages the business logic.

So, let's take a look at how to do this:

1. Let's start by adding the code for the nought (which, as we know, is just a circle):

```
struct Nought: View {
    var body: some View {
        Circle()
            .stroke(.red, lineWidth: 10)
    }
}
```

2. Next, we will implement the shape of a cross:

```
struct CrossShape: Shape {  
    func path(in rect: CGRect) -> Path {  
        Path() { path in  
  
            path.move(to: CGPoint(x: rect.minX,  
                                  y: rect.minY))  
            path.addLine(to: CGPoint(x: rect.maxX,  
                                    y: rect.maxY))  
  
            path.move(to: CGPoint(x: rect.maxX,  
                                  y: rect.minY))  
            path.addLine(to: CGPoint(x: rect.minX,  
                                    y: rect.maxY))  
        }  
    }  
}
```

3. Now, we will implement a Cross view that renders the shape of the CrossShape struct with a green stroke:

```
struct Cross: View {  
    var body: some View {  
        CrossShape()  
            .stroke(.green, style:  
                    StrokeStyle(lineWidth: 10,  
                                lineCap: .round,  
                                lineJoin: .round))  
    }  
}
```

4. Next, we will add the Cell view, which can be either a nought or a cross, and visible or hidden:

```
struct Cell: View {  
    enum CellType {  
        case hidden
```

```
        case nought
        case cross
    }
    @State private var type: CellType = .hidden
    @Binding var isNextNought: Bool

    @ViewBuilder
    private var content: some View {
        switch type {
        case .hidden:
            Color.clear
        case .nought:
            Nought()
        case .cross:
            Cross()
        }
    }

    var body: some View {
    }
}
```

5. In the body of the struct, we will add the code to present the correct View and handle the tap gesture to show the cell:

```
var body: some View {
    content
    .padding(20)
    .contentShape(Rectangle())
    .onTapGesture {
        guard type == .hidden else {
            return
        }
        type = isNextNought ? .nought : .cross
        isNextNought.toggle()
    }
}
```

- Now, let's implement the Tic-Tac-Toe grid, starting from its shape, which is defined by a `Path` object:

```
struct GridShape: Shape {  
    func path(in rect: CGRect) -> Path {  
        Path() { path in  
            }  
    }  
}
```

- Inside the `Path` object, we will add the four lines that will make up the Tic-Tac-Toe grid:

```
path.move(to: CGPoint(x: rect.width/3,  
                      y: rect.minY))  
path.addLine(to: CGPoint(x: rect.width/3,  
                        y: rect.maxY))  
path.move(to: CGPoint(x: 2*rect.width/3,  
                      y: rect.minY))  
path.addLine(to: CGPoint(x: 2*rect.width/3,  
                        y: rect.maxY))  
path.move(to: CGPoint(x: rect.minX,  
                      y: rect.height/3))  
path.addLine(to: CGPoint(x: rect.maxX,  
                        y: rect.height/3))  
path.move(to: CGPoint(x: rect.minX,  
                      y: 2*rect.height/3))  
path.addLine(to: CGPoint(x: rect.maxX,  
                        y: 2*rect.height/3))
```

- At this point, we could create a `Grid` view with nine nested cells, but it is better to introduce the concept of `Row`, which contains three cells horizontally:

```
struct Row: View {  
    @Binding  
    var isNextNought: Bool  
  
    var body: some View {  
        HStack {
```

```
        Cell(isNextNought: $isNextNought)
        Cell(isNextNought: $isNextNought)
        Cell(isNextNought: $isNextNought)
    }
}
}
```

9. Add the following code, which includes the Grid view with three vertically stacked Row structs:

```
struct Grid: View {
    @State
    var isNextNought: Bool = false

    var body: some View {
        ZStack {
            GridShape()
                .stroke(.indigo, lineWidth: 15)
            VStack {
                Row(isNextNought: $isNextNought)
                Row(isNextNought: $isNextNought)
                Row(isNextNought: $isNextNought)
            }
        }
        .aspectRatio(contentMode: .fit)
    }
}
```

10. The last thing we must do is add the Grid view to the ContentView struct:

```
struct ContentView: View {
    var body: some View {
        Grid()
            .padding(.horizontal, 20)
    }
}
```

This has been quite a long recipe, but when you preview the screen, you can see how we can play Tic-Tac-Toe almost as if we have a pencil and paper in front of us:

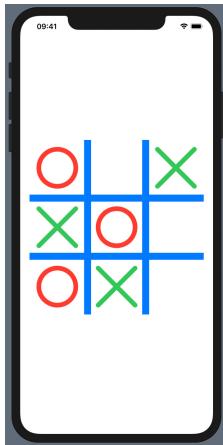


Figure 7.11 – Playing Tic-Tac-Toe with SwiftUI

How it works...

This simple game is a perfect example of how it is possible to create quite sophisticated interactions composed of simple components.

The `isNextNought` variable defines which type of mark will be placed next. It is set to `false` in the `Grid` component, which means that the first mark will always be `Cross`. When tapping on a cell, the `isNextNought` variable will be toggled, alternating the type of mark, a cross or a nought, that will be placed each time.

It is interesting to note that before applying the view modifier for the `onTapGesture` gesture, we must set a `contentShape()` modifier. The reason for this is because the default tappable area is given by the visible part of the component, but at the start, all the cells are hidden, so the area is empty!

The `contentShape()` modifier then defines the hit test area in which touch can be detected. In this case, we want that to occupy the whole area, so `Rectangle` is used, but we could use `Circle`, `Capsule`, or even a custom shape.

There's more...

The game is almost complete, but it is missing at least a couple of things:

- Selecting the first mark
- Detecting when a player has won

Create these features to explore SwiftUI and the way it manages shapes even further. Adding these two features will help you understand how the components work together. With these features, you can create a complete Tic-Tac-Toe game app that you could even release in the App Store!

Rendering a gradient view in SwiftUI

SwiftUI has several ways of rendering gradients. A gradient can be used to fill a shape, or even fill a border.

In this recipe, we will understand what types of gradients we can use with SwiftUI and how to define them.

Getting ready

Create a SwiftUI app called `GradientViews`.

How to do it...

SwiftUI has three different types of gradients:

- Linear gradients
- Radial gradients
- Angular gradients

In each one, we can define the list of colors that will smoothly transform into each other. Depending on the type of gradient, we can define some additional properties such as the direction, radius, and angles of the transformation.

To explore all of them, we are going to add a `Picker` component to select the type of gradient.

The `ContentView` struct will have a `Text` component that shows the selected gradient. We can do this by performing the following steps:

1. Let's start by adding a style, including a custom font and color, to our `Text` component:

```
extension Text {  
    func bigLight() -> some View {  
        font(.system(size: 80))  
        .fontWeight(.thin)
```

```
    .multilineTextAlignment(.center)
    .foregroundColor(.white)
}
}
```

2. The first gradient we will add is the linear one, where the color transitions in a linear direction:

```
struct LinearGradientView: View {
    var body: some View {
        ZStack {
            LinearGradient(
                gradient:
                    Gradient(colors:
                        [.orange, .green,
                         .blue, .black]),
                startPoint: .topLeading,
                endPoint: .bottomTrailing)
            Text("Linear Gradient")
                .bigLight()
        }
    }
}
```

3. The second gradient we will create is the radial gradient, where the colors transition through concentric circles:

```
struct RadialGradientView: View {
    var body: some View {
        ZStack {
            RadialGradient(
                gradient:
                    Gradient(colors:
                        [.orange, .green,
                         .blue, .black]),
                center: .center,
                startRadius: 20,
                endRadius: 500)
        }
    }
}
```

```
        Text("Radial Gradient")
            .bigLight()
        }
    }
}
```

4. The last gradient we will create is the angular gradient, where the transition rotates in a complete rotation:

```
struct AngularGradientView: View {
    var body: some View {
        ZStack {
            AngularGradient(
                gradient:
                    Gradient(
                        colors: [.orange, .green,
                                  .blue, .black,
                                  .black, .blue,
                                  .green, .orange]),
                center: .center)
            Text("Angular Gradient")
                .bigLight()
        }
    }
}
```

5. Finally, we will prepare `ContentView` with a builder to create the selected gradient view:

```
struct ContentView: View {
    @State
    private var selectedGradient = 0
    @ViewBuilder var content: some View {
```

```
        switch selectedGradient {
            case 0:
                LinearGradientView()
            case 1:
                RadialGradientView()
            default:
                AngularGradientView()
        }
    }

    var body: some View {
}
```

6. In the body, we will render the selected view and Picker to select the view:

```
var body: some View {
    ZStack(alignment: .top) {
        content
            .edgesIgnoringSafeArea(.all)

        Picker(selection: $selectedGradient,
               label: Text("Select Gradient")) {
            Text("Linear").tag(0)
            Text("Radial").tag(1)
            Text("Angular").tag(2)
        }
        .pickerStyle(SegmentedPickerStyle())
        .padding(.horizontal, 32)
    }
}
```

By running the app, we can see the different types of gradients. You may be amazed by how such a beautiful effect can be achieved with so few lines of code:



Figure 7.12 – Linear, radial, and angular gradients

How it works...

Each type of gradient has a list of color parameters, and this provides great flexibility. In this recipe, we saw that there are three different ways of representing a gradient: linear, radial, and angular. For each of them, we defined a list of the colors it draws from the origin to the destination.

It is worth noting that the colors don't need to be just the original and destination colors; rather, we can have as many as we want. The `Gradient` SwiftUI view will take care of creating a smooth transition between each pair of colors.

Each different gradient type offers further possibilities for customization:

- In the case of a **linear** gradient, you can define the direction in which the gradient changes (for example, top to bottom, or top-leading to bottom-trailing).
- For the **radial** gradient, you can set the radius of the concentric circles for each of the transitions.
- In the **angular** gradient, you can define the center where the angle of the color transition starts, as well as the start and end angles. If you omit these angles, a full rotational gradient will be created.

Building a bar chart

Using simple shapes, it is possible to build some nice features. For example, by just using a bunch of rectangles, we can create a bar chart.

In this recipe, we are going to create a bar chart that presents the average monthly precipitation for three European cities: Dublin, Milan, and London. The data that we'll use can be found at <https://www.climatestotravel.com/>.

Getting ready

This recipe doesn't need any external resources, so it's enough to create a SwiftUI project called BarChart.

How to do it...

Based on information from <https://www.climatestotravel.com/>, the data we have represents the average quantity of rain in centimeters. Looking at the data, we can see that the maximum rainfall is 1 centimeter, so we can adapt the bars to have a full shape for that value.

To implement our bar chart, follow these steps:

1. Add enum to represent the months and a value type datapoint to hold the values that will be plotted on the bar chart:

```
enum Month: String, CaseIterable {
    case jan, feb, mar, apr, may, jun,
    jul, aug, sep, oct, nov, dec
}

struct MonthDataPoint: Identifiable {
    var id: String { month.rawValue }
    let month: Month
    let value: Double
    var name: String {
        month.rawValue.capitalized
    }
}
```

2. To simplify how we will save the raw data, let's introduce an extension to an array of Double to transform it into an array of MonthDataPoint:

```
extension Array where Element == Double {  
    func monthDataPoints() -> [MonthDataPoint] {  
        zip(Month.allCases, self)  
            .map(MonthDataPoint.init)  
    }  
}
```

3. The data for the average precipitation is collected from <https://www.climatestotravel.com> and we will save it as a series of static constants:

```
struct DataSet {  
    static let dublin = [  
        0.65, 0.50, 0.55, 0.55, 0.60, 0.65,  
        0.55, 0.75, 0.60, 0.80, 0.75, 0.75  
    ].monthDataPoints()  
  
    static let milan = [  
        0.65, 0.65, 0.80, 0.80, 0.95, 0.65,  
        0.70, 0.95, 0.70, 1.00, 1.00, 0.60  
    ].monthDataPoints()  
  
    static let london = [  
        0.55, 0.40, 0.40, 0.45, 0.50, 0.45,  
        0.45, 0.50, 0.50, 0.70, 0.60, 0.55,  
    ].monthDataPoints()  
}
```

4. Now, let's implement the bar chart, starting from the single BarView struct:

```
struct BarView: View {  
    var dataPoint: MonthDataPoint  
    var body: some View {  
        VStack {  
            ZStack (alignment: .bottom) {  
                Rectangle()  
            }  
        }  
    }  
}
```

```

        .fill(.blue)
        .frame(width: 18,
               height: 180)
    Rectangle()
        .fill(.white)
        .frame(width: 18,
               height: dataPoint.value * 180.0)
    }
Text(dataPoint.name)
    .font(.system(size: 11))
    .rotationEffect(.degrees(-45))
}
}
}

```

5. Add `BarChartView`, which is a horizontal stack that we will populate using a `ForEach` loop. This explains why our the data points conform to `Identifiable`:

```

struct BarChartView: View {
    var dataPoints: [MonthDataPoint]

    var body: some View {
        HStack(spacing: 12) {
            ForEach(dataPoints) {
                BarView(dataPoint: $0)
            }
        }
    }
}

```

6. In `ContentView`, we will add the model containing the list of the dataset and a `@State` property to select the city:

```

struct ContentView: View {
    let dataSet = [
        DataSet.dublin,
        DataSet.milan,
        DataSet.london
    ]
}

```

```
]

@State
var selectedCity = 0

var body: some View {
}
}
```

7. Finally, we will add BarChartView in body of ContentView, along with a title and a segmented picker view to allow us to select the city:

```
var body: some View {
    VStack (spacing: 24) {
        Spacer()
        Text ("Average Precipitation")
            .font (.system (size: 32))

        Picker (selection: self.$selectedCity,
                label: Text ("Average Precipitation")) {
            Text ("Dublin").tag(0)
            Text ("Milan").tag(1)
            Text ("London").tag(2)
        }
        .pickerStyle (SegmentedPickerStyle())

        BarChartView (dataPoints:
                      dataSet [selectedCity])
        Spacer()
    }
    .padding (.horizontal, 10)
    .background (
        .mint,
        ignoresSafeAreaEdges: .vertical)
}
```

By running the app, we can see how, when selecting the different cities, the bars nicely change to represent their values:

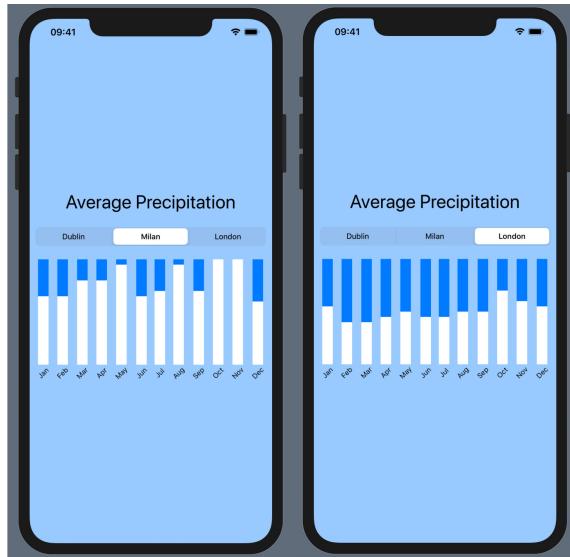


Figure 7.13 – Average Precipitation bar charts

How it works...

As you've seen, the code of the bars is really simple. A bar is simply two rectangles: an empty rectangle for the background and a rectangle that represents the percentage of the value as a proportion of the full length of the bar.

Below each bar, we added a `Text` component that shows the name of the month that the given bar represents, and we rotated it 45 degrees for readability.

Unrelated to SwiftUI, the extended `monthDataPoints()` function shows a nice trick for joining two sequences. The list of the months is one sequence and the array of the values is another sequence; the `zip()` function joins these two sequences as if it were a sequence of tuples so that we can iterate and create a new `MonthDataPoint` struct from each tuple. Pretty neat!

There's more...

The bars are scaled so that there's a full bar for 1 centimeter, but what if the maximum rainfall is less than or greater than 1 centimeter?

A good exercise would be to adapt the code we just wrote so that it has a flexible and configurable maximum height, depending on the maximum value in the dataset.

Another nice feature could be adding a label for each bar representing the value: could you do this while taking into account when the value is the maximum or the minimum? Give it a try!

Building a pie chart

We know that a pie chart is a way to represent proportional numeric values by using slices that form a circle – a pie. With a pie chart being made up of a slice and a circle – simple geometric shapes – it is simple enough to implement them in SwiftUI.

This recipe will use a dataset based on the number of pets in three different European cities. Unlike the previous recipe, the data here has been made up.

Getting ready

This recipe doesn't have any external resources, so it is enough just to create a SwiftUI project called `PieChart`.

How to do it...

This recipe is slightly more complicated than usual because it has two main parts:

- Manipulating datapoints
- Visualizing datapoints

The datapoints value must be adjusted by finding the maximum value and adapting the others to fit the chart. Also, we must calculate the angles for each slice – the starting angle (which is the ending angle of the previous slice) and the ending angle, both of which are proportional to the value of that slice.

Let's start by writing the code. You'll see that it is less complicated than it sounds:

1. First of all, we will implement `enum` to define the pet we want to track, along with a function that returns its associated color to present it in the pie chart:

```
enum Animal: String {  
    case cat  
    case dog  
    case fish
```

```

        case horse
        case hamster
        case rabbit
        case bird

    var color: Color {
        switch self {
        case .cat: return .red
        case .dog: return .blue
        case .fish: return .green
        case .horse: return .orange
        case .hamster: return .purple
        case .rabbit: return .gray
        case .bird: return .yellow
        }
    }
}

```

2. Then, we will add a simple struct to contain the raw data:

```

struct PetData {
    let value: Double
    let animal: Animal
    var color: Color {
        animal.color
    }

    var name: String {
        animal.rawValue.capitalized
    }
}

```

3. For simplicity, we will set the raw data as static variables in a DataSet struct:

```

struct DataSet {
    static let dublin: [PetData] = [
        .init(value: 2344553, animal: .cat),
        .init(value: 1934345, animal: .dog),
    ]
}

```

```
        .init(value: 323454, animal: .fish),
        .init(value: 403400, animal: .rabbit),
        .init(value: 1003445, animal: .horse),
        .init(value: 1600494, animal: .hamster),
    ]
static let milan: [PetData] = [
    .init(value: 3344553, animal: .cat),
    .init(value: 2004345, animal: .dog),
    .init(value: 923454, animal: .fish),
    .init(value: 803400, animal: .rabbit),
    .init(value: 1642345, animal: .bird),
    .init(value: 804244, animal: .hamster),
]
static let london: [PetData] = [
    .init(value: 3355553, animal: .cat),
    .init(value: 4235345, animal: .dog),
    .init(value: 1913454, animal: .fish),
    .init(value: 1103400, animal: .rabbit),
    .init(value: 683445, animal: .horse),
    .init(value: 3300494, animal: .hamster),
]
}
```

4. Add a Datapoint struct to hold the value and the information to be presented in the chart, such as the start angle, the color, and the percentage value:

```
struct DataPoint: Identifiable {
    let id = UUID()
    let label: String
    let value: Double
    let color: Color
    var percentage = 0.0
    var startAngle = 0.0
}
```

5. To build the legend, add a helper function to format the value in a readable way:

```
struct DataPoint: Identifiable {  
    //...  
    var formattedPercentage: String {  
        String(format: "%.2f %%", percentage * 100)  
    }  
}
```

6. To calculate and adapt the points, we will introduce a `DataPoints` struct that holds the points. Every time a new point is added, it will calculate the maximum value, scale the percentages of the data points, and move the angles accordingly:

```
struct DataPoints {  
    var points = [DataPoint]()  
  
    mutating func add(value: Double,  
                      label: String, color: Color) {  
        points.append(DataPoint(label: label,  
                               value: value, color: color))  
        let total = points.map(\.value).reduce(0.0,+)  
        points = points.map {  
            var point = $0  
            point.percentage = $0.value / total  
            return point  
        }  
  
        for i in 1..            let previous = points[i - 1]  
            let angle = previous.startAngle +  
                previous.value*360/total  
            var current = points[i]  
            current.startAngle = angle  
            points[i] = current  
        }  
    }  
}
```

- With all the data in place, it is time to prepare the drawing part of our project, starting with a single slice. As you can see, the slice conforms to `InsettableShape`:

```
struct PieSliceShape: InsettableShape {  
    var percent: Double  
    var startAngle: Angle  
    var insetAmount: CGFloat = 0  
  
    func inset(by amount: CGFloat) ->  
        some InsettableShape {  
        var slice = self  
        slice.insetAmount += amount  
        return slice  
    }  
  
    func path(in rect: CGRect) -> Path {  
        Path() { path in  
            //...  
        }  
    }  
}
```

- In the `Path` that we just defined in the `PieSliceShape` struct, we will add an arc that conforms to the passed parameters:

```
func path(in rect: CGRect) -> Path {  
    Path() { path in  
        path.addArc(center: CGPoint  
(x: rect.size.width/2,  
y: rect.size.height/2),  
radius: rect.size.width/2 - insetAmount,  
startAngle: startAngle,  
endAngle: startAngle +  
Angle(degrees: percent * 360),  
clockwise: false)  
    }  
}
```

9. Given the shape defined in the preceding code, we will wrap it in a view using a `GeometryReader` component, where we will render it with a stroke of a size that is half the width of the frame:

```
struct PieSlice: View {  
    var percent: Double  
    var degrees: Double  
    var color: Color  
  
    var body: some View {  
        GeometryReader { geometry in  
            PieSliceShape(percent: percent,  
                          startAngle: Angle(degrees: degrees))  
                .strokeBorder(color,  
                              lineWidth:geometry.size.width/2)  
                .rotationEffect(.degrees(-90))  
                .aspectRatio(contentMode: .fit)  
        }  
    }  
}
```

10. The `PieChart` view consists of two other views: a legend that describes the percentage of pet owners for each type of pet and a list of pie slices. Since we want to show what each slice represents, a legend with the color and the related percentage is the simplest way to achieve that result. Each slice is then rendered on top of the others to create the effect of a pie. Add a `PieChart` with the following code:

```
struct PieChart: View {  
    var dataPoints: DataPoints  
  
    var body: some View {  
        VStack(alignment: .leading, spacing: 30) {  
            VStack(alignment: .leading) {  
                ForEach(dataPoints.points) { p in  
                    HStack(spacing: 16) {  
                        Rectangle()  
                            .foregroundColor(p.color)  
                            .frame(width: 16, height: 16)  
                }  
            }  
        }  
    }  
}
```

```
        Text("\(p.label):  
        \(p.formattedPercentage) ")  
    }  
}  
}  
ZStack {  
    ForEach(dataPoints.points) { point in  
        PieSlice(percent: point.percentage,  
                  degrees: point.startAngle,  
                  color: point.color)  
    }  
}  
.aspectRatio(contentMode: .fill)  
}  
}  
}
```

11. In the ContentView struct, add a dataset property and a variable to select the city:

```
struct ContentView: View {  
    var dataSet: [DataPoints] = [  
        DataSet.dublin.reduce(into: DataPoints()) {  
            $0.add(value: $1.value,  
                    label: $1.name, color: $1.color)  
        },  
        DataSet.milan.reduce(into: DataPoints()) {  
            $0.add(value: $1.value,  
                    label: $1.name, color: $1.color)  
        },  
        DataSet.london.reduce(into: DataPoints()) {  
            $0.add(value: $1.value,  

```

```
        label: $1.name, color: $1.color)
    },
]

@State var selectedCity = 0

var body: some View {
    //...
}
```

12. Finally, the body function lays out the three cities in Picker and the PieChart View:

```
var body: some View {
    VStack (spacing: 50) {
        Text("Most Popular Pets")
            .font(.system(size: 32))

        Picker(selection: self.$selectedCity,
               label: Text("Most Popular Pets")) {
            Text("Dublin").tag(0)
            Text("Milan").tag(1)
            Text("London").tag(2)
        }.pickerStyle(SegmentedPickerStyle())
        PieChart(dataPoints: dataSet[selectedCity])
            .aspectRatio(1, contentMode: .fit)
        Spacer()
    }
    .padding(.horizontal, 20)
}
```

By running the app, we can see how the pie chart is rendered and how it changes its values when we select a different city:

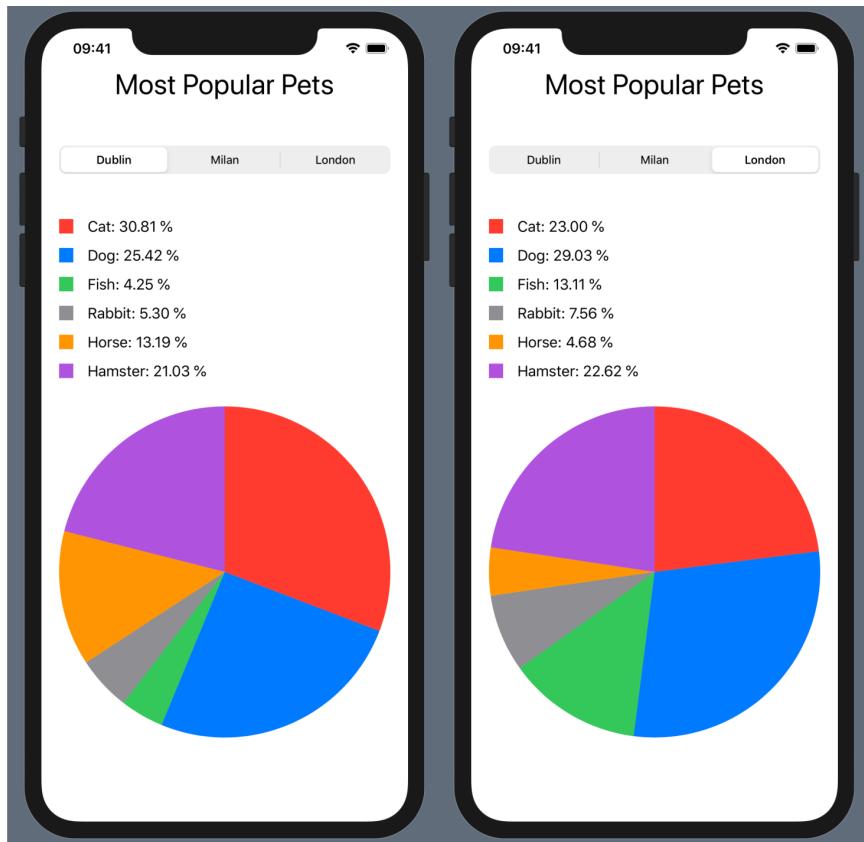


Figure 7.14 – Most Popular Pets pie chart

How it works

In general, this recipe is not complicated, but there are a few caveats to take into consideration.

First, let's analyze the `DataPoints` struct, whose objective is to hold a list of ready-to-render data points. To do this, every time a new point is added, the total is calculated with the following line of code:

```
let total = points.map(\.value).reduce(0.0,+)
```

Then, the percentage of each point is updated:

```
points = points.map {
    var point = $0
    point.percentage = $0.value / total
    return point
}
```

Finally, the angles are updated, with the start angle considered as the end angle of the previous slice:

```
for i in 1..<points.count {
    let previous = points[i - 1]
    let angle = previous.startAngle +
    previous.value*360/total
    var current = points[i]
    current.startAngle = angle
    points[i] = current
}
```

Another nice tip is to use the `InsettableShape` protocol, which allows us to use `strokeBorder()` instead of `stroke()`:

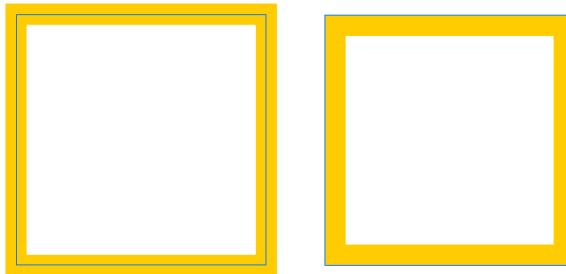


Figure 7.15 – `stroke()` and `strokeBorder()` applied to two shapes

The difference is that in the case of `stroke()`, the yellow border grows beyond the outline of the shape, which remains in the middle of the border; whereas in the case of `strokeBorder()`, the yellow border is completely contained in the outline.

To conform to the `InsettableShape` protocol, a shape must implement an `inset(by:)` function that is called when `strokeBorder()` is invoked, passing the size of the inset to be applied. In our pie slice, we save this in a property called `insetAmount`, which will be used to reduce the radius of the arc to be drawn:

```
path.addArc(center: CGPoint(x: rect.size.width/2,
                             y: rect.size.width/2),
            radius: rect.size.width/2 - insetAmount,
            startAngle: startAngle,
            endAngle: startAngle + Angle(degrees: percent *
            360),
            clockwise: false)
```

Finally, the use of the `GeometryReader` object in the `PieSlice` view is interesting as it reads the frame in which it is encapsulated. It also explicitly creates a `geometry` object that contains the information of the frame that we use to define the size of the stroke for that slice:

```
GeometryReader { geometry in
    //...
    .strokeBorder(color, lineWidth: geometry.size.width/2)
    //...
}
```

This has been a quite long recipe, but we created quite a sophisticated component and learned about a few tricks we can reuse in other components.

8

Animating with SwiftUI

SwiftUI has introduced not only a new way of describing UI elements and components but also a new way of implementing animations. In the case of animations, an even more complex change of thinking is needed. Though the layout concept is inherently declarative, the animation concept is inherently imperative.

When creating an animation in UIKit, for example, it is normal to describe it as a series of steps: when *this* happens, do *that* animation for one second, then *another* animation for two seconds.

Animation in the SwiftUI way requires us to define three parts:

- A **trigger**: An event that *happens*, such as a button click, a slider, a gesture, and so on
- A **change of data**: A change of an `@State` variable, such as a Boolean flag
- A **change of UI**: A change of something that is represented visually following the change of data – for example, a vertical or horizontal offset, or the size of a component that has one value when the flag is false and another value when the flag is true

In the following recipes, we'll learn how to implement basic and implicit animations, how to create custom animations, and also how to recreate some effects that we experience every day in the most used apps that we have on our devices.

At the end of the chapter, we'll know a set of techniques that will allow us to create the most compelling animations in the SwiftUI way.

In this chapter, we will cover animations in SwiftUI through the following recipes:

- Creating basic animations
- Transforming shapes
- Creating a banner with a spring animation
- Applying a delay to a view modifier animation to create a sequence of animations
- Applying a delay to a `withAnimation` function to create a sequence of animations
- Applying multiple animations to a view
- Creating custom view transitions
- Creating a hero view transition with `.matchedGeometryEffect`
- Lottie animations in SwiftUI
- Implementing a stretchable header in SwiftUI
- Creating floating hearts in SwiftUI
- Implementing a swipeable stack of cards in SwiftUI

Technical requirements

The code in this chapter is based on Xcode 13 and iOS 15.

You can find the code in the book's GitHub repo at the following link: <https://github.com/PacktPublishing/SwiftUI-Cookbook-2nd-Edition/tree/main/Chapter08-Animating-with-SwiftUI>.

Creating basic animations

Let's introduce the way to animate in SwiftUI with a simple app that moves a component on the screen. SwiftUI brings a few predefined temporal curves: `.easeInOut`, `.linear`, `.spring`, and so on. In this recipe, we'll compare them with the default animation.

We are going to implement two circles that move to the top or the bottom of the screen. One circle moves using the default animation and the other with the selected animation; we can then select the other animation using an action sheet, which is a modal view that appears at the bottom.

Getting ready

Let's implement a SwiftUI app called `BasicAnimations`.

How to do it...

This is a super-simple app where we are going to render two circles, a red and a blue one, and an action sheet to choose the animation for the red circle, while the animation for the blue circle is always the default one. We will select the animation for the red circle with a button that presents an action sheet with a list of the possible animations. Finally, the animation is triggered by a default button with the text **Animate**.

Let's get started:

1. First, add a type for the animation to be able to list all of the possible animations and then select one:

```
struct AnimationType {
    let name: String
    let animation: Animation

    static var all: [AnimationType] = [
        .init(name: "default", animation: .default),
        .init(name: "easeIn", animation: .easeIn),
        .init(name: "easeOut", animation: .easeOut),
        .init(name: "easeInOut", animation: .easeInOut),
        .init(name: "linear", animation: .linear),
        .init(name: "spring", animation: .spring()),
    ]
}
```

2. Then, add three `@State` variables to drive the animation and the components to be shown in the view:

```
struct ContentView: View {
    @State
```

```
var onTop = false
@State
var type = AnimationType(name: "default",
                           animation: .default)
@State
var showSelection = false
//...
}
```

3. The next step is to add two circles of the same size but of different colors:

```
var body: some View {
    VStack(spacing: 12) {
        GeometryReader { geometry in
            HStack {
                Circle()
                    .fill(.blue)
                    .frame(width: 80, height: 80)
                    .offset(y: onTop ?
                            -geometry.size.height/2 :
                            geometry.size.height/2)
                    .animation(.default, value: onTop)
                Spacer()
                Circle()
                    .fill(.red)
                    .frame(width: 80, height: 80)
                    .offset(y: onTop ?
                            -geometry.size.height/2 :
                            geometry.size.height/2 )
                    .animation(type.animation, value: onTop)
            }.padding(.horizontal, 30)
            //...
        }
    }
}
```

4. The code for the UI and the animation is there; let's just add the ActionSheet component with the list of the available animations:

```
struct ContentView: View {  
    //...  
    var actionSheet: ActionSheet {  
        ActionSheet(title: Text("Animations"),  
                    buttons: AnimationType  
                        .all  
                        .map { type in  
                            .default(Text(type.name)) } {  
                                self.type = type  
                            }  
                        } + [ .destructive(Text("Cancel")) ]  
        )  
    }  
    //...  
}
```

5. Finally, after adding the buttons to trigger the animation and the selection of the animation, we are ready to test it:

```
struct ContentView: View {  
    //...  
    var body: some View {  
        VStack(spacing: 12) {  
            GeometryReader { geometry in  
                //...  
            }  
            Button {  
                onTop.toggle()  
            } label: {  
                Text("Animate")  
            }  
            Button {  
                showSelection.toggle()  
            } label: {  
                Text("Choose Animation")  
            }  
        }  
    }  
}
```

```
        }
        .actionSheet(isPresented: $showSelection) {
            actionSheet
        }
        Text("Current: \(type.name)")
    }
}
```

6. Running the app, we can now see how the different animations run compared to each other:

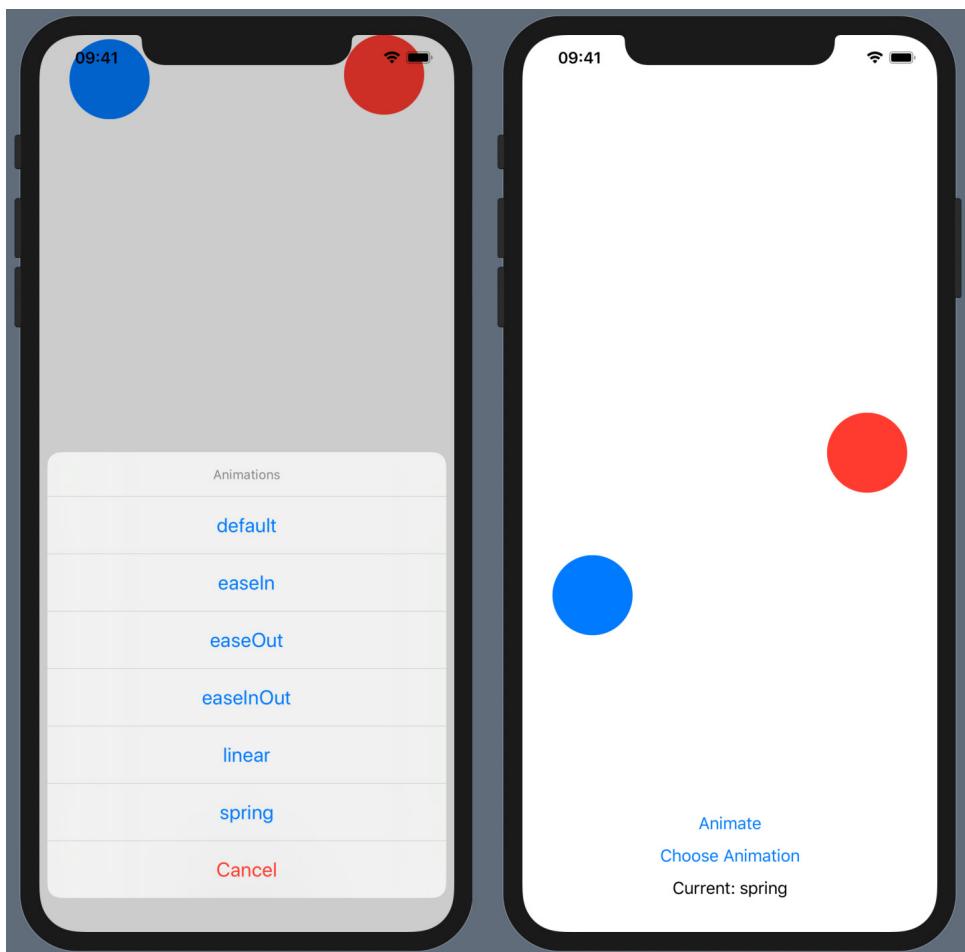


Figure 8.1 – Basic animations in SwiftUI

How it works...

As you can see, the result is pretty neat considering that we have used just a couple of lines to add an animation.

In our code, we can see the three steps we mentioned in the introduction:

- The trigger is the tap on the **Animate** button.
- The change of data is the change of the `@State` variable `onTop`.
- The change of UI is the vertical position of the circle, which is guided by the `onTop` variable.

When given the three previous steps, and after adding the `.animation()` view modifier, SwiftUI will then apply that animation.

Note that we must pass the change of data parameter to the `.animation()` modifier, along with the type of animation. In our case, the `onTop` property must be passed as a second parameter.

In practical terms, this means that for the duration of the animation, SwiftUI calculates the position of the circle using the selected curve.

For example, with `.easeInOut`, the animation starts and finishes slowly but it is fast in between, whereas with `.linear`, the speed is always constant.

If you want to slow down the animation to see the difference, you can apply a `.speed()` modifier to the animation, such as the following:

```
//...
.animation(Animation.default.speed(0.1), value: onTop)
//...
.animation(self.type.animation.speed(0.1), value: onTop)
//...
```

By playing around with that, you should better understand the difference between the different animations.

There's more...

Changing the offset is just one of the possible changes of the UI. What about experimenting with changing other things – for example, the fill colors or the size of the circles? Does the animation work for every modification? Feel free to experiment and get familiar with the way SwiftUI manages the animations.

See also

If you want to have a visualization of the different curves, you will find a graph for the most common easing functions here: <https://easings.net/en>.

Bear in mind that that site is not SwiftUI-related, so the names are slightly different.

Transforming shapes

In the *Creating basic animations* recipe, you can see that SwiftUI is able to animate the change of common characteristics, such as position, color, size, and so on. But what if the feature we want to animate is not part of the framework?

In this recipe, we'll create a triangular shape whose height is equal to the width times a fraction of the width. When we tap on the triangle, we set that multiplier to a random number, making the height change.

How can we instruct SwiftUI to animate the change of the multiplier? We'll see that the code needed is simple but that the underlying engine is quite sophisticated.

Getting ready

This recipe doesn't need any external resources, so let's just create a SwiftUI project called `AnimateTriangleShape` in Xcode.

How to do it...

We are going to implement a triangle shape where the height is equal to a fraction of the width. Then, when tapping on the shape, the multiplier will randomly change. To do this, follow these steps:

1. Let's start by adding a `Triangle` view:

```
struct Triangle: Shape {
    var multiplier: CGFloat
    func path(in rect: CGRect) -> Path {
        Path { path in
            path.move(to: CGPoint(x: rect.minX,
                                  y: rect.maxY))
            path.addLine(to: CGPoint(x: rect.maxX,
                                    y: rect.maxY))
            path.addLine(to: CGPoint(x: rect.midX,
```

```

                y: rect.maxY
                - multiplier *
                rect.width))
            path.closeSubpath()
        }
    }
}

```

2. Then, add the shape to `ContentView` and a gesture that changes the multiplier:

```

struct ContentView: View {
    @State
    var multiplier = 1.0

    var body: some View {
        Triangle(multiplier: multiplier)
            .fill(.red)
            .frame(width: 300, height: 300)
            .onTapGesture {
                withAnimation(.easeOut(duration: 1)) {
                    multiplier = .random(in: 0.3...1.5)
                }
            }
    }
}

```

This looks like all the code we need, but if we run the app now, we can see that although the triangle changes if we tap on it, the change is not animated.

This is because SwiftUI doesn't know which data it has to animate.

3. To instruct it, what we have to do is to add a property called `animatableData` to the `Triangle` struct:

```

struct Triangle: Shape {
    var animatableData: CGFloat {
        get { multiplier }
        set { multiplier = newValue }
    }
}

```

```
// ...  
}
```

Running the app now, the height changes smoothly when we tap on the triangle:

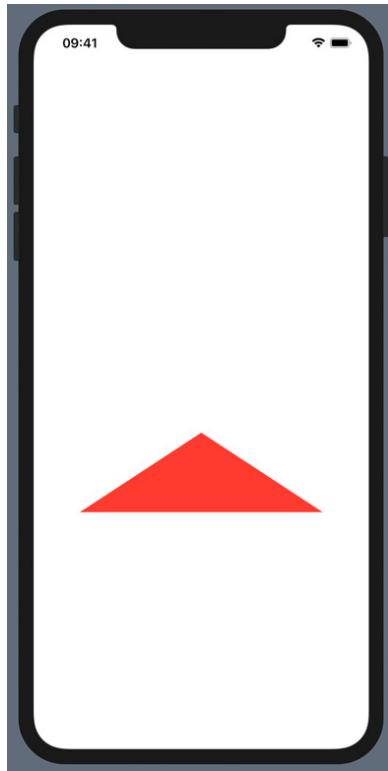


Figure 8.2 – Shape animated transformation

How it works...

SwiftUI can only animate components that conform to `Animatable`. It means that they should have a property called `animatableData` so that SwiftUI can save and retrieve the intermediate steps during an animation.

To inspect the behavior, let's add `print` to the setter:

```
set {  
    multiplier = newValue  
    print("value: \(multiplier)")  
}
```

Running the app, the Xcode console will print something such as the following:

```
value: 1.0
value: 1.0
value: 1.0000194373421276
value: 1.0025879432661413
value: 1.0051161861243192
value: 1.0076164878031886
value: 1.0100930134474908
value: 1.0125485398203875
value: 1.0149842817557613
value: 1.0174018011828905
value: 1.0198019658402626
value: 1.0221849492755757
value: 1.0245516192273172
value: 1.02690249633858
value: 1.0292379277047592
value: 1.03155826042125
value: 1.0338634944880523
value: 1.0361541505482588
```

Figure 8.3 – Intermediate value of animatableData

For every step, SwiftUI calculates the value for `animatableData`, sets it in the shape, and then renders the shape.

The shape already conforms to `Animatable`, so the only thing we have to do is define the `animatableData` property, specifying the characteristic we want to animate.

Another thing to note is the way we are triggering the animation. In the gesture action, we are wrapping the change of the `@State` variable with a `withAnimation` function:

```
withAnimation(.easeOut(duration: 1)) {
    multiplier = .random(in: 0.3...1.5)
}
```

This is like saying to SwiftUI, *“Everything that changes inside this function must be animated using the animation configuration passed as a parameter.”*

I hope this gives you an understanding of how animations work inside SwiftUI. If you still have any doubts, the other recipes in this chapter should help dispel all of them.

Creating a banner with a spring animation

A nice and configurable easing function is the spring, where the component bounces around the final value. We are going to implement a banner that is usually hidden, and when it appears, it moves from the top with a spring animation.

Getting ready

No external resources are needed, so let's just create a SwiftUI project in Xcode called `BannerWithASpringAnimation`.

How to do it...

This is a really simple recipe, where we create a banner view that can be animated when we tap on a button:

1. Implement `BannerView`:

```
struct BannerView: View {  
    let message: String  
    var show: Bool  
  
    var body: some View {  
        Text(message)  
            .font(.title)  
            .frame(width:UIScreen.main.bounds.width - 20,  
                   height: 100)  
            .foregroundColor(.white)  
            .background(Color.green)  
            .cornerRadius(10)  
            .offset(y: show ?  
                      -UIScreen.main.bounds.height / 3 :  
                      -UIScreen.main.bounds.height)  
            .animation(  
                .interpolatingSpring(mass: 2.0,  
                                      stiffness: 100.0,  
                                      damping: 10,  
                                      initialVelocity: 0),  
                value: show)  
    }  
}
```

```
    }
}
```

2. Then, we add the banner and a `Button` component to trigger the visibility in `ContentView`:

```
struct ContentView: View {
    @State
    var show = false

    var body: some View {
        VStack {
            BannerView(message: "Hello, World!", show: show)
            Button {
                show.toggle()
            } label: {
                Text(show ? "Hide" : "Show")
                    .padding()
                    .frame(width: 100)
                    .foregroundColor(.white)
                    .background(show ? .red : .blue)
                    .cornerRadius(10)
            }
        }
    }
}
```

When we run the app, we can see that the banner nicely bounces when it appears:

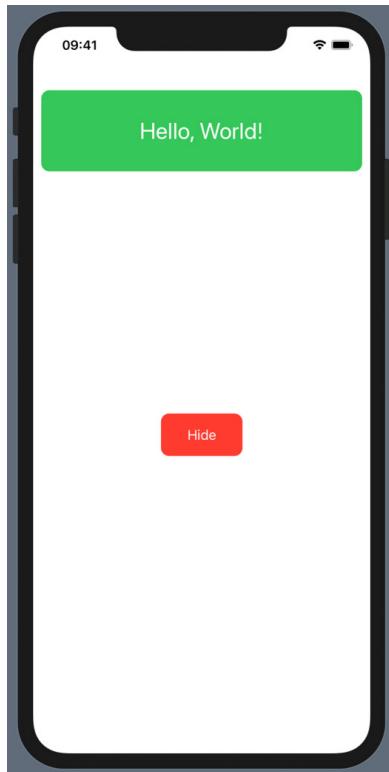


Figure 8.4 – A bouncing banner view

How it works...

To reiterate what we mentioned in the introduction – the trigger is the tap on the button, the change in data is the `show` variable, and the change in the UI is the position of the banner.

The curve is a spring curve, which has a few parameters:

- **mass:** This is the mass of the object attached to the spring – the bigger it is, the more inertia it gains when it reaches the speed, so it bounces more.
- **stiffness:** This is how much the spring resists when a force is applied – the more resistance, the more rigid the spring is.

- `damping`: This is how much the spring resists changes – the more resistance, the less bouncing of the spring.
- `initialVelocity`: This is the velocity when the animation starts.

Feel free to change the parameters and see how the animation changes.

Applying a delay to a view modifier animation to create a sequence of animations

We have reached version 3 of SwiftUI, but still there is no way of joining different animations together to create a sequence of animations. This will surely be fixed in a later SwiftUI version, but at the moment, we can implement a sequence of animations using a delay.

As you should know, there are two ways of defining an animation:

- Using the `.animation()` view modifier
- Using the `withAnimation` function

In this recipe, we'll see how to use the `.animation()` view modifier, and we'll cover the `withAnimation` function in the next recipe, *Applying a delay to a withAnimation function to create a sequence of animations*.

Getting ready

Let's create a SwiftUI project in Xcode called `DelayedAnimations`.

How to do it...

In our app, we will create a sequence of three animations on a rectangle:

- A change of the vertical offset
- A change of scale
- A 3D rotation around the *X* axis

Since we cannot create an animation with these sub-animations, we are going to use a delay to reach the same effect:

1. Let's start by adding an `@State` variable to activate the animation, as well as the rectangle with the three animations:

```
struct ContentView: View {
    let duration = 1.0
    @State
    var change = false

    var body: some View {
        VStack(spacing: 30) {
            Rectangle()
                .fill(.blue)
                .offset(y: change ? -300 : 0)
                .animation(
                    .easeInOut(duration: duration).delay(0),
                    value: change)
                .scaleEffect(change ? 0.5 : 1)
                .animation(
                    .easeInOut(duration: duration)
                    .delay(duration),
                    value: change)
                .rotation3DEffect(
                    change ? .degrees(45) : .degrees(0),
                    axis: (x: 1, y: 0, z: 0))
                .animation(
                    .easeInOut(duration: duration)
                    .delay(2*duration), value: change)
                .frame(width: 200, height: 200)
        }
    }
}
```

2. Then, let's add Button to trigger the animation:

```
var body: some View {
    VStack(spacing: 30) {
        //...
        Button {
            change.toggle()
        } label: {
            Text("Animate")
                .fontWeight(.heavy)
                .foregroundColor(.white)
                .padding()
                .background(.green)
                .cornerRadius(5)
        }
    }
}
```

Running the app, we can see the animations joining together as if they were a single animation:

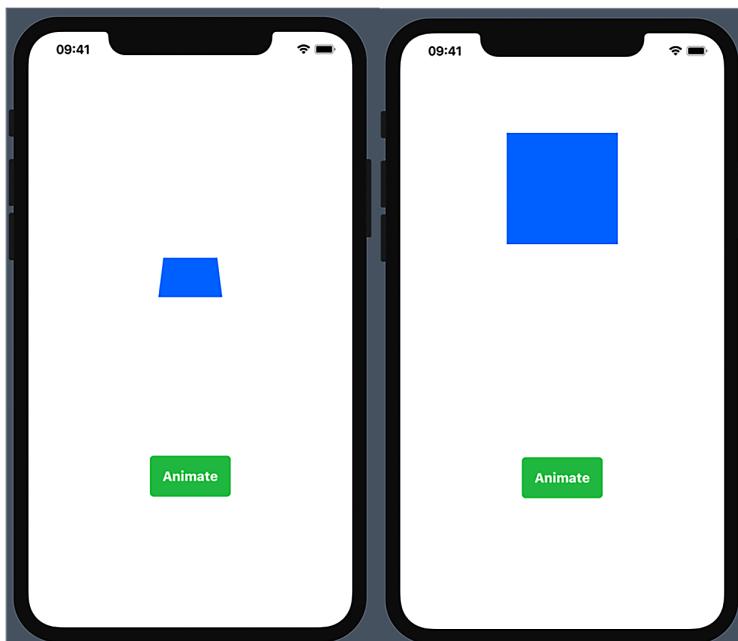


Figure 8.5 – Chaining animations with a delay

How it works...

When defining an animation, you can add a delay to make it start after a while.

You will notice that every animation is related to the previous change, so you can have multiple changes that happen at the same time.

For simplicity, we defined three animations with the same duration. The second animation must start after the first finishes as we used a `duration` delay. It's the same thing for the third animation too, which must start after both the previous animations have finished.

Also, even though the technique is pretty simple, it doesn't work for all the view modifiers. So, play around and test them to find the correct sequence of animations.

Applying a delay to a `withAnimation` function to create a sequence of animations

As we mentioned in the previous recipe, SwiftUI doesn't have a way to define a chain of animations yet, but this can be simulated using delay animations.

As mentioned in the previous recipe, there are two ways of defining an animation:

- Using the `.animation()` view modifier
- Using the `withAnimation` function

In this recipe, we'll see how to use the `withAnimation` function. We covered the `.animation()` view modifier in the previous recipe, *Applying a delay to a view modifier animation to create a sequence of animations*.

Getting ready

This recipe doesn't need any external resources, so let's just create a SwiftUI project called `DelayedAnimations`.

How to do it...

To illustrate the delay applied to the withAnimation function, we are going to implement an app that presents three text elements that appear and disappear in sequence when tapping on a button:

1. To add a nice look, define a custom modifier for Text:

```
struct CustomText: ViewModifier {  
    let foreground: Color  
    let background: Color  
    let cornerRadius: Double  
  
    func body(content: Content) -> some View {  
        content  
            .foregroundColor(foreground)  
            .frame(width: 200)  
            .padding()  
            .background(background)  
            .cornerRadius(cornerRadius)  
    }  
}
```

2. In order for our code to be less verbose, we use the modifier in an extension to the Text struct definition, which defines a helper function for adding a uniform style to our Text instances:

```
extension Text {  
    func styled(color: Color) -> some View {  
        modifier(CustomText(foreground: .white,  
                            background: color,  
                            cornerRadius: 10))  
  
    }  
}
```

3. We can now add three `@State` variables to drive the visibility of each `Text` component:

```
struct ContentView: View {  
    @State var hideFirst = true  
    @State var hideSecond = true  
    @State var hideThird = true  
    var body: some View {  
        VStack {  
            VStack(spacing: 30) {  
                Text("First")  
                    .styled(color: .red)  
                    .opacity(hideFirst ? 0 : 1)  
                Text("Second")  
                    .styled(color: .blue)  
                    .opacity(hideSecond ? 0 : 1)  
  
                Text("Third")  
                    .styled(color: .yellow)  
                    .opacity(hideThird ? 0 : 1)  
            }  
        }  
    }  
}
```

4. Finally, let's add the trigger Button:

```
var body: some View {  
    VStack {  
        //...  
        Spacer()  
        Button {  
            withAnimation(Animation.easeInOut) {  
                hideFirst.toggle()  
            }  
            withAnimation(Animation.easeInOut.delay(0.3))  
            {  
                hideSecond.toggle()  
            }  
        }  
    }  
}
```

```
        }
        withAnimation(Animation.easeInOut.delay(0.6))
    {
        hideThird.toggle()
    }
} label: {
    Text("Animate")
        .fontWeight(.heavy)
        .styled(color: .green)
}
}
}
```

Running the app, we can see how the opacity of the components changes in sequence, simulating a single animation:

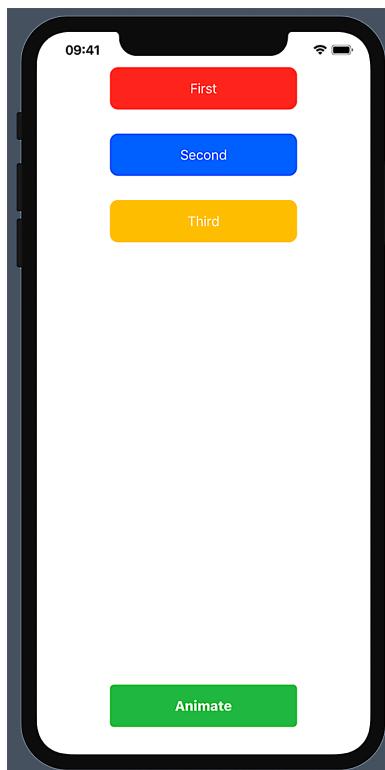


Figure 8.6 – Delayed animations to create a sequential appearance of components

How it works...

Remember that with the `withAnimation` function, we are telling SwiftUI to animate what is inside the function that we pass as the last parameter; it is pretty obvious that applying a delay to the same animation will cause it to start later.

Although the current and previous recipes are two working solutions, I think you'll agree that they are more of a workaround than a proper pattern.

Let's hope that the next version of SwiftUI will bring us a proper way to chain animations together.

Applying multiple animations to a view

SwiftUI allows us to animate multiple features at the same time, and they can also be animated using different durations and different animation curves.

In this recipe, we'll learn how to animate two sets of features and how to make the result look like one single, smooth animation.

Getting ready

Let's create a SwiftUI project called `MultipleAnimations`.

How to do it...

To illustrate how you can apply multiple animations to a view, we are going to create a rectangle that has two sets of animations:

- One set with the color, the vertical offset, and the rotation around the X axis
- One set with the scale and a rotation around the Z axis

We are using an `.easeInOut` curve for the former and `.linear` for the latter. To do this, follow these steps:

1. Let's start by adding the rectangle and the button to trigger the change:

```
struct ContentView: View {  
    @State  
    var initialState = true  
  
    var body: some View {  
        VStack(spacing: 30) {
```

```
Rectangle()
Button {
    initialState.toggle()
} label: {
    Text("Animate")
        .fontWeight(.heavy)
        .foregroundColor(.white)
        .padding()
        .background(.green)
        .cornerRadius(5)
}
}
```

2. We can now add the first set of changes to the Rectangle instance, with a `.easeInOut` animation:

```
//...
Rectangle()
    .fill(initialState ? .blue : .red)
    .cornerRadius(initialState ? 50 : 0)
    .offset(y: initialState ? 0 : -200)
    .rotation3DEffect(initialState ? .degrees(0)
        : .degrees(45),
        axis: (x: 1, y: 0, z: 0))
    .animation(.easeInOut(duration: 2), value:
        initialState)
```

3. Finally, we add the second set of changes, with a `.linear` animation:

```
Rectangle()
//...
.scaleEffect(initialState ? 1 : 0.8)
.rotationEffect(initialState ? Angle(degrees:0) :
    Angle(degrees:-90))
.animation(.linear(duration: 1), value:
```

```
        initialState)  
.frame(width: 300, height: 200)
```

Running the app, we can see how the two animation sets interact:

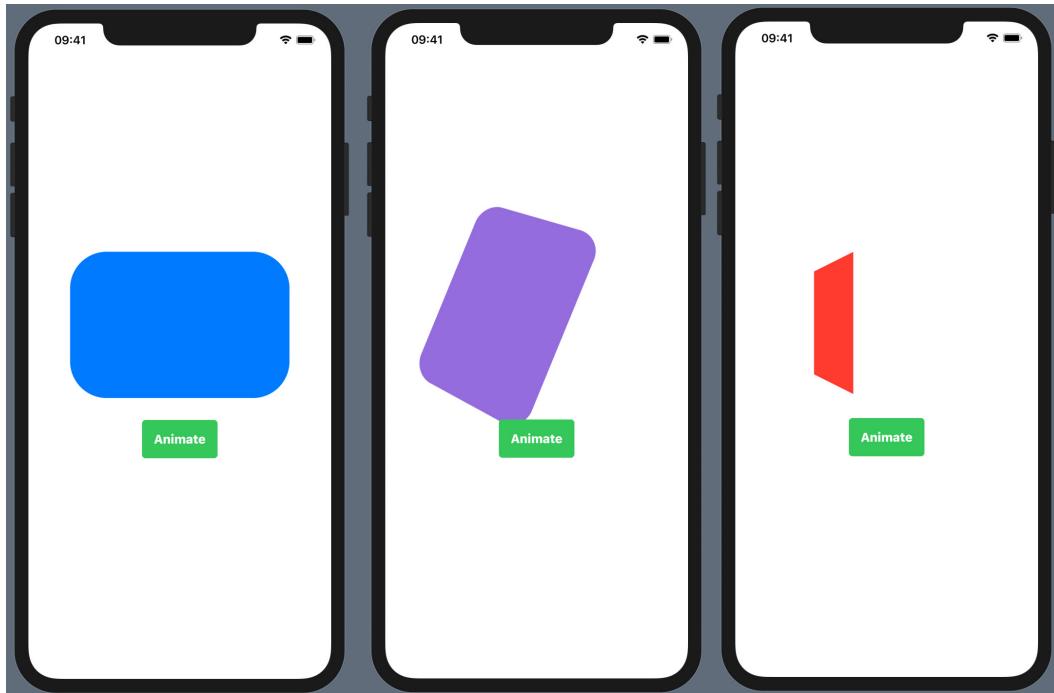


Figure 8.7 – Multiple animations on the same components

How it works...

Remember in the introduction how we mentioned the three steps of an animation – a trigger, a change of data, and a change of UI?

Basically, changing multiple features of a component is considered a single change for SwiftUI. For each step, SwiftUI calculates the intermediary value for each of them and then applies all the changes at the same time for every single step.

Creating custom view transitions

SwiftUI has a nice feature that gives us the possibility to add an animation when a view appears or disappears. It is called a **transition**, and it can be animated with the usual degree of customization.

In this recipe, we'll see how to create custom appearing and disappearing transitions by combining different transitions.

Getting ready

This recipe uses two images courtesy of *Erika Wittlieb* from *Pixabay* (<https://pixabay.com/users/erikawittlieb-427626/>).

You can find the images in the GitHub repo at the following link, <https://github.com/PacktPublishing/SwiftUI-Cookbook-2nd-Edition/tree/main/Resources/Chapter08/recipe7>, but you can also use your own images for this recipe. If you don't want to use your own images, you can replace them with two symbols from SF Symbols, such as "gamecontroller" or "car."

Then, create a new SwiftUI project in Xcode called **CustomViewTransition**, and copy the **ch8-r7-i1.jpg** and **ch8-r7-i2.jpg** images in the **Assets** catalog:

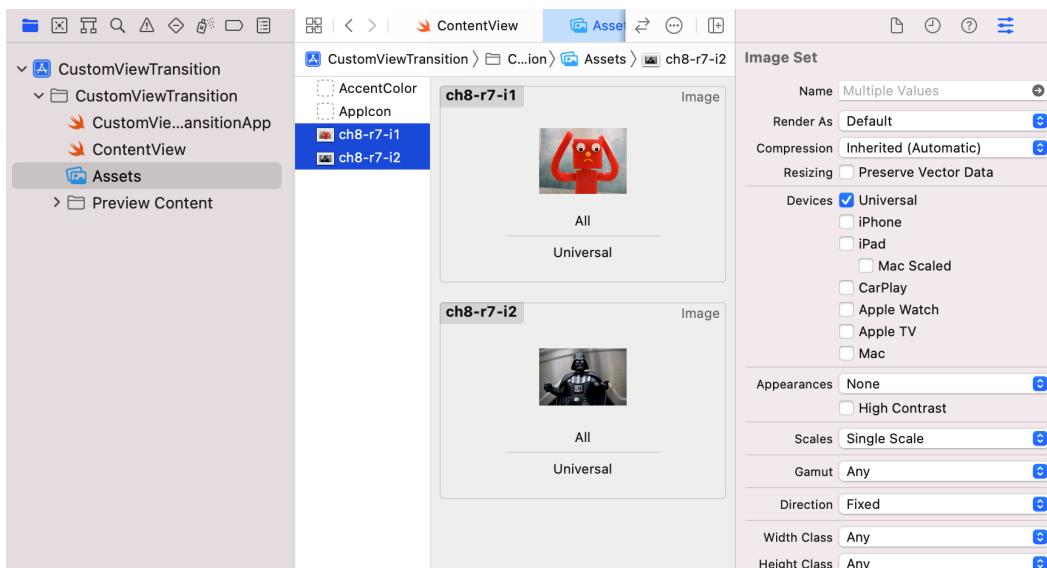


Figure 8.8 – Adding the images to the Assets catalog

How to do it...

We are going to implement a simple app where we are switching two views when we tap on a button. For simplicity, the two components are just wrappers around an `Image` component, but this will work for any kind of component:

1. Firstly, we will implement the two views that wrap around an `Image` component:

```
extension Image {  
    func custom() -> some View {  
        self  
            .resizable()  
            .aspectRatio(contentMode: .fit)  
            .cornerRadius(20)  
            .shadow(radius: 10)  
    }  
}  
  
struct FirstComponent: View {  
    var body: some View {  
        Image("ch8-r7-i1")  
            .custom()  
    }  
}  
  
struct SecondComponent: View {  
    var body: some View {  
        Image("ch8-r7-i2")  
            .custom()  
    }  
}
```

2. Let's now put the components in the `ContentView`, selecting which one to present depending on a flag that will be toggled by a button:

```
struct ContentView: View {  
    @State var showFirst = true  
    var body: some View {  
        VStack(spacing: 24) {
```

```
        if showFirst {
            FirstComponent()

        } else {
            SecondComponent()
        }
    Button {
        showFirst.toggle()
    } label: {
        Text("Change")
    }
}
.animation(Animation.easeInOut, value:
    showFirst)
.padding(.horizontal, 20)
}
}
```

3. If you run the app now, the two images will be swapped using a crossfade animation, which is the default animation for the images.
4. Let's introduce the concept of transitions by creating a new type of transition and modifying the images to respect the transition animation when appearing or disappearing:

```
extension AnyTransition {
    static var moveScaleAndFade: AnyTransition {
        let insertion = AnyTransition
            .scale
            .combined(with: .move(edge: .leading))
            .combined(with: .opacity)
        let removal = AnyTransition
            .scale
            .combined(with: .move(edge: .top))
            .combined(with: .opacity)
        return .asymmetric(insertion: insertion,
                           removal: removal)
    }
}
```

```
}

struct ContentView: View {
    //...
    FirstComponent()
        .transition(.moveScaleAndFade)
    //...
    SecondComponent()
        .transition(.moveScaleAndFade)
    //...
}
```

Running the app now, we can see that the components appear by scaling up and moving from the left, and disappear by scaling down and moving to the top:

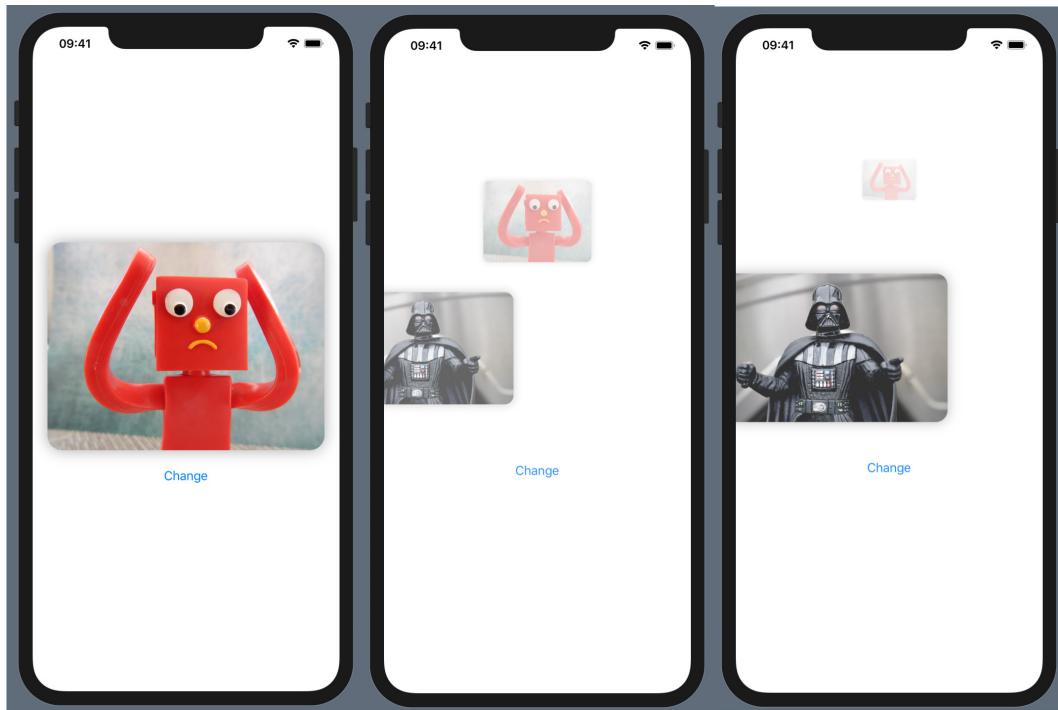


Figure 8.9 – Custom view transitions

How it works...

A transition is basically a list of transformations that can be applied when a view appears or disappears.

Using the `.combined` function, a transition can be combined with others to create more sophisticated animations.

In our example, we used a `.asymmetric` transition. It means that the removal will look different than the insertion. But of course, we could also have a symmetric transition returning the insertion or removal transition – try it and see how the app behaves.

Creating a hero view transition with `.matchedGeometryEffect`

Do you know what a hero transition is? If you don't know the term, you have still probably seen it many times – maybe in an e-commerce app, where, with a list of products, each product also has a thumbnail to show the product. Selecting a product flies it to a details page, with a big image of the same product. The smooth animation from the thumbnail to the big image is called a **hero transition**.

Another example is the cover image animation in the Apple Music player – transitioning from the mini player to the full player. SwiftUI provides a modifier, `.matchedGeometryEffect`, which makes it very easy to implement this kind of animation almost without any effort.

Getting ready

This recipe uses a few images, courtesy of *Pixabay* (<https://pixabay.com>).

You can find the images in this GitHub link, <https://github.com/PacktPublishing/SwiftUI-Cookbook-2nd-Edition-/tree/main/Resources/Chapter08/recipe8>, but you can also use your own images for this recipe.

Create a new SwiftUI project in Xcode called HeroViewTransition and copy the images to the **Assets** catalog:

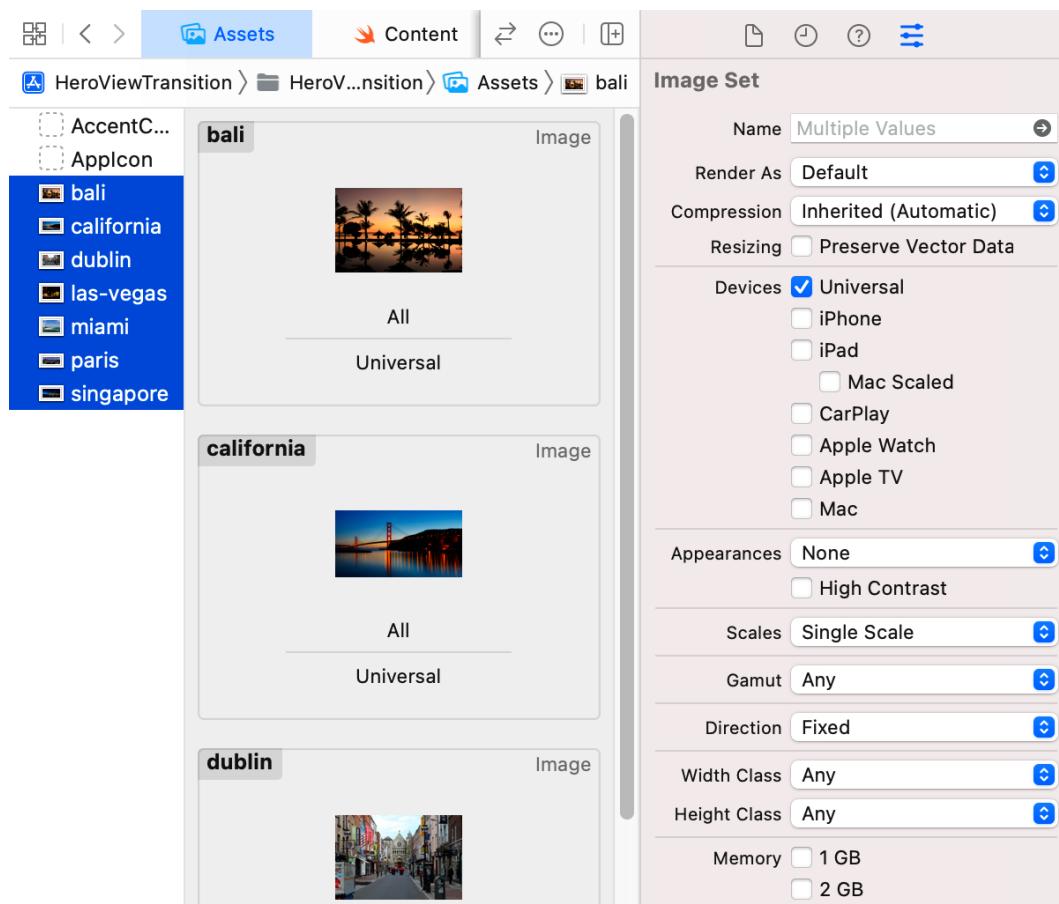


Figure 8.10 – Adding the images to the Assets catalog

How to do it...

We are going to implement a simple app with a list of holiday destinations. When the user selects one destination, the thumbnail flies to the details page. A cross mark button on the page allows the user to close it. When the page is closing, the big image flies back to the original position in the list view. To do this, follow these steps:

- Firstly, we implement the model for our holiday destination app:

```
struct Item: Identifiable {
    let id = UUID()
```

```
let image : String  
let title : String  
let details : String  
}
```

- With this model, we can create an array of items:

```
let data = [  
    Item(image: "california",  
        title: "California",  
        details: "California, the most populous state in  
        the United States and the third most extensive by  
        area, is located on the western coast of the USA  
        and is bordered by Oregon to the north."),  
    Item(image: "miami",  
        title: "Miami",  
        details: "Miami is an international city at  
        Florida's  
        south-eastern tip. Its Cuban influence is  
        reflected in the cafes"),  
    //...  
]
```

- Since this code is long and not so interesting, you can find the Data.swift file with it in the GitHub repo: <https://github.com/PacktPublishing/SwiftUI-Cookbook-2nd-Edition-/blob/main/Resources/Chapter08/recipe8/Data.swift>. Copy this into your project.
- We will move now to the ContentView component, where we add the property to drive the animation:

```
struct ContentView: View {  
    @State private var selectedItem: Item!  
    @State private var showDetail = false  
    @Namespace var animation  
    var body: some View {  
        //...  
    }  
}
```

5. Inside the body, we add the two components: DestinationListView, with the list of the holiday destinations, and DestinationDetailView, with the information for the selected item:

```
var body: some View {
    ZStack {
        DestinationListView(selectedItem: $selectedItem,
                            showDetail: $showDetail,
                            animation: animation)
            .opacity(showDetail ? 0 : 1)
        if showDetail {
            DestinationDetailView(selectedItem: selectedItem,
                                showDetail: $showDetail,
                                animation: animation)
        }
    }
}
```

6. Let's start implementing DestinationListView, defining the properties to drive its layout:

```
struct DestinationListView: View {
    @Binding var selectedItem: Item!
    @Binding var showDetail: Bool
    let animation: Namespace.ID

    var body: some View {
        }
    }
}
```

7. In the body function, add a ScrollView component, where we iterate on the data array to show the thumbnails:

```
var body: some View {
    ScrollView(.vertical) {
        VStack(spacing: 20) {
            ForEach(data) { item in
            }
    }
}
```

```
        }
        .padding(.all, 20)
    }
}
```

8. Inside the `ForEach` loop body, we present an `Image` component:

```
ForEach(data) { item in
    Image(item.image)
        .resizable()
        .aspectRatio(contentMode: .fill)
        .cornerRadius(10)
        .shadow(radius: 5)
}
```

9. After the `.shadow` modifier, we apply the secret ingredient:

`.matchedGeometryEffect!` This will be explained in the *How it works...* section, but for the moment, just apply it to the `Image` component:

```
Image(item.image)
//...
.shadow(radius: 5)
.matchedGeometryEffect(id: item.image,
    in: animation)
```

10. Finally, a `.onTapGesture` callback will select the item to open and display the detail page:

```
Image(item.image)
//...
.onTapGesture {
    selectedItem = item
    withAnimation {
        showDetail.toggle()
    }
}
```

11. Let's move on to implementing the details page, where we define View and its properties:

```
struct DestinationDetailView: View {  
    var selectedItem: Item  
    @Binding var showDetail: Bool  
    let animation: Namespace.ID  
  
    var body: some View {  
    }  
}
```

12. In the body function, add a ZStack component, with the details of the holiday destination, and a Button component to close the page:

```
var body: some View {  
    ZStack(alignment: .topTrailing){  
        VStack{  
        }  
        .ignoresSafeArea(.all)  
        Button {  
            //...  
        }  
        .background(Color.white  
                    .ignoresSafeArea(.all))  
    }  
}
```

13. Let's start with the Button component, where we simply dismiss the page toggling the showDetail flag:

```
Button {  
    withAnimation {  
        showDetail.toggle()  
    }  
} label: {  
    Image(systemName: "xmark")  
        .foregroundColor(.white)  
        .padding()  
        .background(.black.opacity(0.8))
```

```
        .clipShape(Circle())
    }
.padding(.trailing,10)
```

14. Add the following code to the `VStack` component, which contains the information of the product, notably the big image at the top:

```
 VStack{
    Image(selectedItem.image)
        .resizable()
        .aspectRatio(contentMode: .fit)
    Text(selectedItem.title)
        .font(.title)
    Text(selectedItem.details)
        .font(.callout)
        .padding(.horizontal)
    Spacer()
}
.ignoresSafeArea(.all)
```

15. Finally, apply `.matchedGeometryEffect` to the `Image` component on the details page:

```
 Image(selectedItem.image)
    .resizable()
    .aspectRatio(contentMode: .fit)
    .matchedGeometryEffect(id: selectedItem.image,
                           in: animation)
```

Running the app now, when we select a picture from the list, the image smoothly animates, moving to the top of the screen when the details page is fully visible.

When we close the page, the image flies back to its original position in the list:

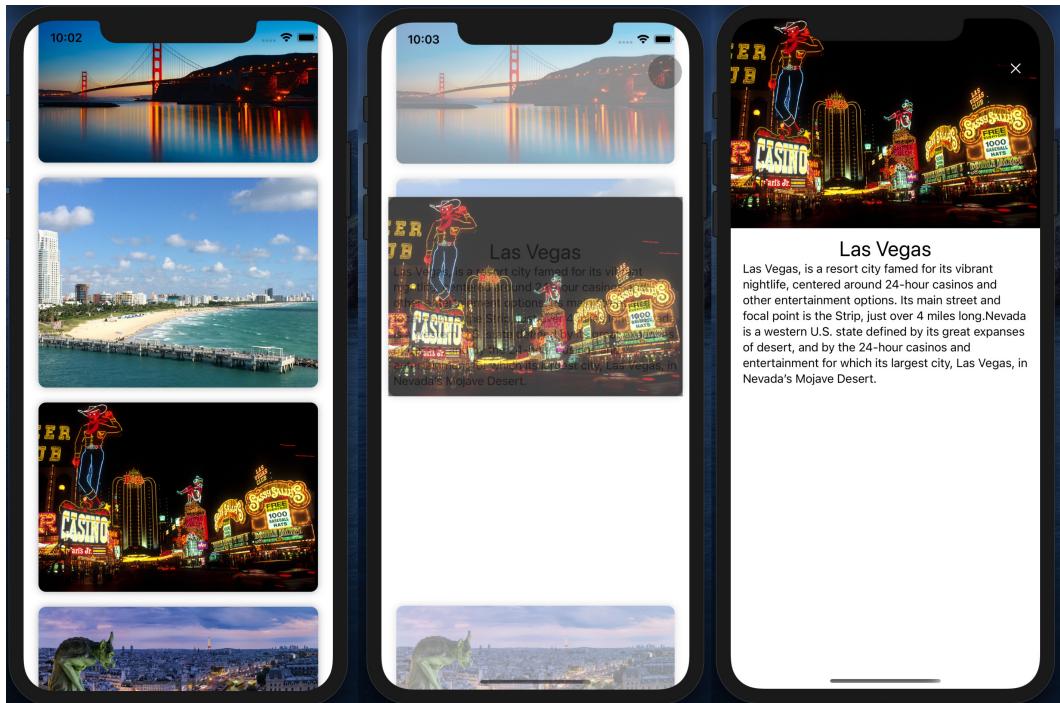


Figure 8.11 – Hero view transition

How it works...

It has been said that `.matchedGeometryEffect` is a sort of **Keynote Magic Move** for SwiftUI, where you set the start component and the end component, and SwiftUI magically creates an animation for you.

The `.matchedGeometryEffect` modifier sets the relationship between the initial and the final component so that SwiftUI can create the animation to transform the initial to the final component. To store this relationship, `.matchedGeometryEffect` needs an identifier – in this case, the name of the image – and a place to save the identifiable relationship. In our code, this is done in the following line:

```
.matchedGeometryEffect(id: item.image, in: animation)
```

SwiftUI provides a property wrapper, `@Namespace`, which transforms the property of a view in a global place to save the animations that SwiftUI must render. The animation engine will use this property under the hood to create the various steps of the animation.

We can simplify this to say that by using `.matchedGeometryEffect`, we define a starting component and an ending component in different views, and then we say to SwiftUI that those components are the same. After that, SwiftUI will figure out what transformations it has to apply.

In our case, the transformations are as follows:

- The *y* offset position (from *inside the scroll view* to the *top of the screen*)
- The size (from the *thumbnail* to the *big picture*)
- Rounded corners (from *with rounded corners* to *without rounded corners*)

When the `showDetail` Boolean flag is `false`, only the list is presented, and the thumbnail images are the initial state of the animation. Toggling the value of `showDetail` makes the details page appear animated. In this case, the big picture on the page is the final state of the animation. Since the identifier is the same in both the components, SwiftUI assumes it is the same component in two different moments in time and renders an animation to transform it from one moment to another.

When `showDetail` is toggled again when showing the details page, the roles of the initial and final state are inverted, and the animation is rendered in reverse.

To visually appreciate the transformation of the animation, I invite you to enable **Debug | Slow Animations** from the menu in the simulator, run the app, and see the various steps of the transition, from the thumbnail to the big picture and back.

Lottie animations in SwiftUI

You probably already know about Lottie (<https://airbnb.design/lottie/>), which is a library for embedding animations made in Adobe After Effects in iOS.

Lottie fills the gap between motion designers and developers; designers can implement their animations using their favorite tool and then export them in JSON format to be used by developers, reproduced in high quality on a device. In this recipe, we'll use an animation downloaded from <https://lottiefiles.com/>, where you can find thousands of animations, either free or paid, so it is definitely a website to keep an eye on.

Getting ready

In this recipe, we'll use a resource, `filling-heart.json`, that we can find in the GitHub repo, and we'll import Lottie using Xcode's Swift Package Manager integration.

Let's start by creating a SwiftUI project called `LottieInSwiftUI`.

Then, add the `filling-heart.json` file, which you can find in the GitHub repo at <https://github.com/PacktPublishing/SwiftUI-Cookbook-2nd-Edition-/blob/main/Resources/Chapter08/recipe9/filling-heart.json>. Remember to tick the **Copy items if needed** box when dragging the file into the Xcode project.

Now, we must include the **Lottie** framework in the project using the Swift Package Manager Xcode integration:

1. Select the project, then the **Package Dependencies** tab, followed by the *plus* button:

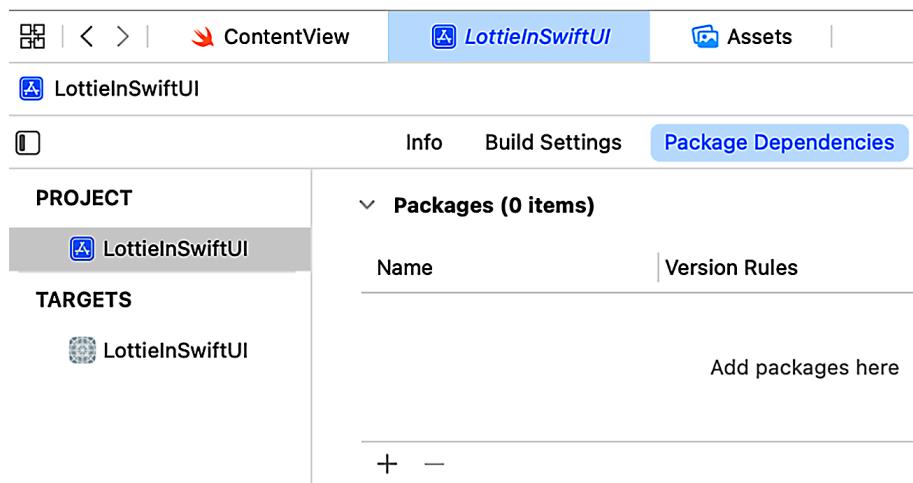


Figure 8.12 – The Package Dependencies tab

2. In the package repository view, search for the <https://github.com/airbnb/lottie-ios> URL and then add the package:

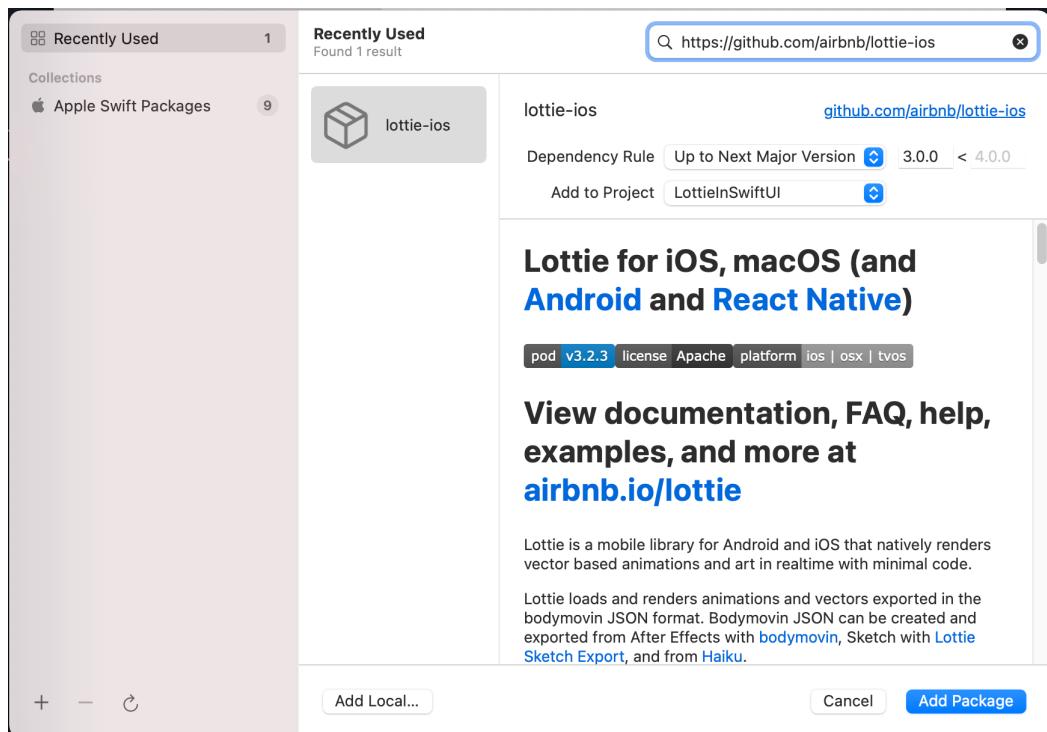


Figure 8.13 – The package repository view

3. Add the **Lottie** package to the project target:

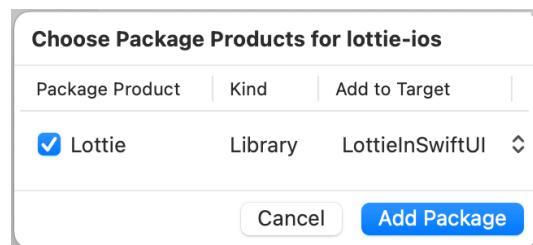


Figure 8.14 – Adding Lottie to the app target

- After downloading the framework, Lottie should be ready to use:

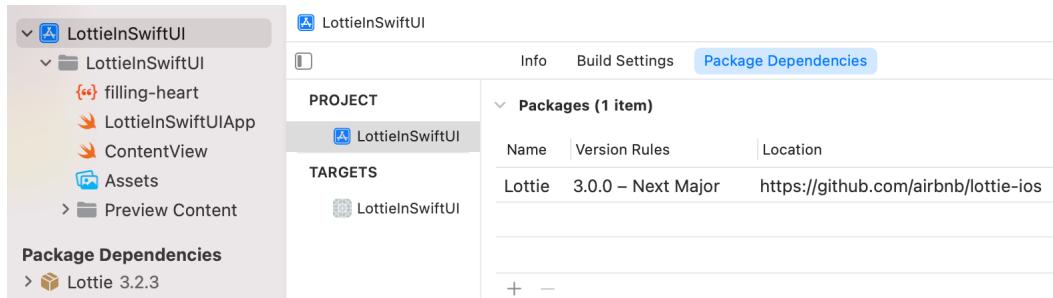


Figure 8.15 – Lottie ready in the project

How to do it...

We are going to implement a simple player of an animation. There will be just two components on the screen: an animation view and a button to trigger it. To do so, follow these steps:

- Firstly, create a placeholder component for Lottie:

```
struct LottieAnimation: View {
    var animationName: String

    @Binding var play: Bool
    var body: some View {
        Circle()
    }
}
```

- Add the `LottieAnimation` component in `ContentView`, along with a `Button` component to trigger the animation:

```
struct ContentView: View {
    @State
    private var play = false

    var body: some View {
        ZStack {
            Color.yellow
                .edgesIgnoringSafeArea(.all)
```

```
    VStack {
        LottieAnimation(animationName:
            "filling-heart",
            play: $play)
        .padding(.horizontal, 30)
        Button {
            play = true
        } label: {
            Text("Fill me!")
                .fontWeight(.heavy)
                .padding(15)
                .background(.white)
                .cornerRadius(10)
        }
    }
}
```

3. Let's now implement the `LottieAnimation` view, without forgetting to import `Lottie`. Replace the placeholder for the `LottieAnimation` view with the following code:

```
import Lottie
struct LottieAnimation: UIViewRepresentable {
    private let animationView = AnimationView()
    var animationName = ""
    @Binding var play: Bool

    func makeUIView(context: Context) -> UIView {
        let view = UIView()
        animationView.animation =
            Animation.
        named(animationName)
        animationView.contentMode = .scaleAspectFit
        animationView.scalesLargeContentImage = true
        animationView
```

```
        .translatesAutoresizingMaskIntoConstraints
        = false
    view.addSubview(animationView)
    NSLayoutConstraint.activate([
        animationView.heightAnchor
            .constraint(equalTo: view.heightAnchor),
        animationView.widthAnchor
            .constraint(equalTo: view.widthAnchor)
    ])
    return view
}
}
```

4. Finally, implement the `updateUIView()` function to start the animation when the `play` variable changes:

```
struct LottieAnimation: UIViewRepresentable {
//...
func updateUIView(_ uiView: UIView, context:
Context) {
    guard play else { return }

    animationView.play { _ in
        play = false
    }
}
}
```

Running the app, we can see how, with a few lines of code, we were able to use a nice Lottie animation:



Figure 8.16 – A gorgeous Lottie animation

How it works...

Lottie exposes a `UIView` subclass called `AnimationView`, where we set the animation name and can customize the animation itself – for example, by changing the speed.

Given that it is a `UIView` subclass, we need to wrap it in a `UIViewRepresentable` class to use it in SwiftUI.

The `updateUIView()` function is called every time the view needs to be refreshed – in our case, when the `play` bound variable changes – so that we can call the `play()` function in the `AnimationView` class.

Implementing a stretchable header in SwiftUI

A stretchable header is a well-known effect where, on top of a scroll view, there is an image that scales up when the user slides down the entries.

An example can be found on the artist page of the Spotify app. But in general, it is so common that you are expecting it when a page has a big image as a banner on top of a list of items.

In this recipe, we'll implement a skeleton of the artist page on the Spotify app, and we'll see that this effect is easy to implement in SwiftUI too.

Getting ready

Let's create a SwiftUI app called `StretchableHeader` and add the two images – `avatar.jpg` and `header.jpg` – which you can find in the GitHub repo at <https://github.com/PacktPublishing/SwiftUI-Cookbook-2nd-Edition-/tree/main/Resources/Chapter08/recipe10>:

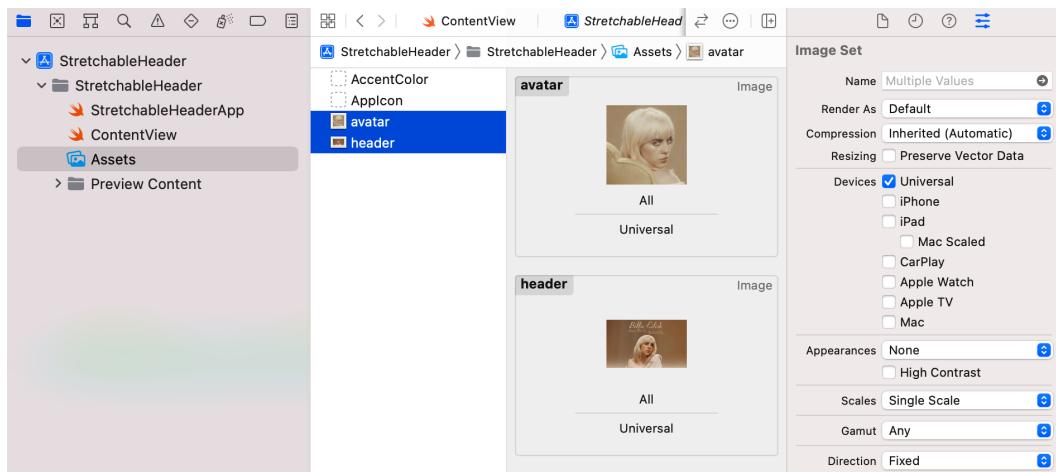


Figure 8.17 – Importing the images

How to do it...

We are going to implement two components, one for each row and the other for the header. The former is a simple horizontal stack with a few components, while the latter is an `Image` component with some tricks to stick it to the top and scale it up when we drag the list of items. To do so, follow these steps:

1. Create a Row component:

```
struct Row: View {
    var body: some View {
        HStack {
            Image("avatar")
                .resizable()
                .frame(width: 50, height: 50)
```

```
        .clipShape(Circle())
        Spacer()
    VStack(alignment: .trailing) {
        Text("Billie Eilish")
            .fontWeight(.heavy)
        Text("Happier Than Ever")
    }
}
.padding(.horizontal, 15)
}
}
```

2. Add the rows to ContentView:

```
struct ContentView: View {
    var body: some View {
        ScrollView(.vertical, showsIndicators: false) {
            VStack {
                ForEach(0..<6) { _ in
                    Row()
                    Divider()
                }
            }
            .edgesIgnoringSafeArea(.all)
        }
    }
}
```

3. Create a StretchableHeader View, which is just a wrapper around an image:

```
struct StretchableHeader: View {
    let imageName: String

    var body: some View {
        GeometryReader { geometry in
            Image(self.imageName)
                .resizable()
                .scaledToFill()
        }
    }
}
```

```
        .frame(width: geometry.size.width,
               height: geometry.height)
        .offset(y: geometry.verticalOffset)
    }
    .frame(height: 350)
}
}
```

4. The code doesn't compile yet. To fix it, add the following functions, such as `GeometryProxy`:

```
extension GeometryProxy {
    private var offset: CGFloat {
        frame(in: .global).minY
    }

    var height: CGFloat {
        size.height + (offset > 0 ? offset : 0)
    }

    var verticalOffset: CGFloat {
        offset > 0 ? -offset : 0
    }
}
```

5. Finally, we can add the header to `ContentView`:

```
struct ContentView: View {
//...
    VStack {
        StretchableHeader(imageName: "header")
        ForEach(0..<6) { _ in
//...
    }
```

Running the app, you can see that the header moves to the top when you slide up, but sticks to the top and increases in size when you slide down:



Figure 8.18 – Drag to stretch the header

How it works...

As you can see, the trick is in the way we calculate the vertical offset: when we try to slide the image down, the top of the image becomes greater than 0, so we apply a negative offset to compensate for the offset caused by the slide, and the image sticks to the top.

It is the same for the height: when the image moves up, the height is normal, but when it moves down after being stuck to the top, the height increases following the drag.

Note that the image is modified with `.scaledToFit()` so that when the height increases, the width increases proportionally, and we have a scale-up effect.

Creating floating hearts in SwiftUI

When Periscope was introduced in 2015, it was the first app to popularize the live-streaming feature that is now a part of other apps, such as Facebook, Instagram, and YouTube.

A way of appreciating the streamer was to send a bunch of floating hearts, a feature that is now also implemented on Instagram and Facebook.

In this recipe, we'll see how to implement this feature in SwiftUI, as an example of a fairly complex real-world animation that can still be done easily.

Getting ready

Let's start by creating a SwiftUI app called `FloatingHearts`.

For this recipe, we are going to use a third-party framework that provides a function to create an interpolation when given a small set of points.

The library is called `SwiftCubicSpline` and can be found at <https://github.com/gscalzo/SwiftCubicSpline>; please refer to the *Lottie Animations in SwiftUI* recipe to see how to install an SPM package using Xcode.

After downloading the framework, `SwiftCubicSpline` should be ready to be used, as you can see in the following screenshot:

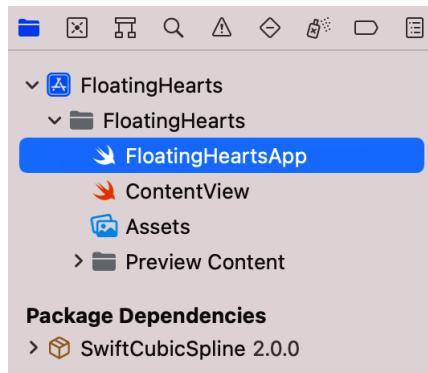


Figure 8.19 – SwiftCubicSpline ready in the project

How to do it...

In our simplified version of the floating hearts animation, we are going to trigger the animation via a bottom-left button.

The general idea is to have a sort of collection of heart components where we add a new heart every time we tap on the button. The heart will be removed from the list after the animation finishes – that is, when the heart reaches the top of the screen.

We are now going to add the code. It's a pretty long and sophisticated recipe, so don't worry if you don't precisely understand every step; everything will be explained in the *How it works...* section.

Here we go:

1. Let's start by adding the trigger button to `ContentView`:

```
struct ContentView: View {
    var body: some View {
        VStack {
            Spacer()
            HStack {
                Button {
                    hearts.new()
                } label: {
                    Image(systemName: "heart")
                        .font(.title)
                        .frame(width: 80, height: 80)
                        .foregroundColor(.white)
                        .background(.blue)
                        .clipShape(Circle())
                        .shadow(radius: 10)
                }
                Spacer()
            }.padding(.horizontal, 30)
        }
        .overlay(HeartsView(hearts: hearts))
    }
}
```

Running the app, you should see the button, which does nothing when tapped.

2. To keep the app runnable, introduce a placeholder for the Heart view, which we will implement properly later.

However, the view must be marked as `Identifiable` to be rendered as a list and `Equatable` so that it can be found and removed from the list:

```
// Placeholder
struct Heart: View, Identifiable {
    let id = UUID()
    var body: some View {
        Circle()
    }
}
extension Heart: Equatable {
    static func == (lhs: Heart, rhs: Heart) -> Bool {
        lhs.id == rhs.id
    }
}
extension Array where Element: Equatable {
    mutating func remove(object: Element) {
        guard let index = firstIndex(of: object)
        else { return }
        remove(at: index)
    }
}
```

3. Now, add the container for the animation and a class to model the list of hearts.

The `Hearts` class is an observable object, so it raises an event when its `@Published` properties change.

The `Hearts` view simply iterates on the list of hearts to render them:

```
class Hearts: ObservableObject {
    @Published
    private(set) var all: [Heart] = []

    func new() {
        let heart = Heart()
        all.append(heart)
        DispatchQueue.main
```

```
        .asyncAfter(deadline: .now() + 10.0) {
            self.all.remove(object: heart)
        }
    }
}

struct HeartsView: View {
    @ObservedObject
    var hearts: Hearts

    var body: some View {
        ForEach(hearts.all) { $0 }
    }
}
```

4. Add the Hearts model and HeartsView to ContentView:

```
struct ContentView: View {
    var hearts = Hearts()

    var body: some View {
        VStack {
            //...
        }
        .overlay(HeartsView(hearts: hearts))
    }
}
```

Running the app, a big black circle appears in the center of the screen. We don't know it yet, but our stream of floating hearts is almost ready to flow!

5. Before implementing the flying hearts, let's add a convenience function to the Color class to create a random value:

```
extension Color {
    init(r: Double, g: Double, b: Double) {
        self.init(red: r/255, green: g/255, blue: b/255)
    }
    static func random() -> Color {
```

```
        Color(r: .random(in: 100...144),  
              g: .random(in: 10...200),  
              b: .random(in: 200...244))  
    }  
}
```

6. Replace the Heart placeholder with the proper implementation:

```
struct Heart: View, Identifiable {  
    let id = UUID()  
  
    @State  
    private var opacity = 1.0  
    @State  
    private var scale: CGFloat = 1.0  
    @State  
    private var toAnimate = false  
  
    var body: some View {  
        Image(systemName: "heart.fill")  
            .foregroundColor(.random())  
            .opacity(opacity)  
            .modifier(MoveShakeScale(pct: toAnimate ?  
                1 : 0))  
            .animation(Animation.easeIn(duration:  
                5.0),  
                value: toAnimate)  
            .task {  
                toAnimate.toggle()  
                withAnimation(.easeIn(duration: 5)) {  
                    opacity = 0  
                }  
            }  
    }  
}
```

7. The last thing missing is the `MoveShakeScale` view modifier, which basically implements the three animations. Let's implement the curves needed for the scale and horizontal movement of animations. For this modifier, we need `SwiftCubicSpline`, so don't forget to import it. Add the following code:

```
import SwiftCubicSpline

struct MoveShakeScale: GeometryEffect {
    private(set) var pct: CGFloat
    private let xPosition =
        UIScreen.main.bounds.width/4 +
        CGFloat.random(in: -20..<20)

    private let scaleSpline = CubicSpline(points: [
        Point(x: 0, y: 0.0),
        Point(x: 0.3, y: 3.5),
        Point(x: 0.4, y: 3.1),
        Point(x: 1.0, y: 2.1),
    ])

    private let xSpline = CubicSpline(points: [
        Point(x: 0.0, y: 0.0),
        Point(x: 0.15, y: 20.0),
        Point(x: 0.3, y: 12),
        Point(x: 0.5, y: 0),
        Point(x: 1.0, y: 8),
    ])
}
```

8. `GeometryEffect` has only one function, `effectValue()`, which receives a `size` value and returns the transformation to apply. In our case, we are applying all the intermediate transformations at the place indicated by the `pct` value, which is also the animated value that goes from 0 to 1 in the given duration. Add the following code:

```
struct MoveShakeScale: GeometryEffect {
    //...
    var animatableData: CGFloat {
        get { pct }
        set { pct = newValue }
    }

    func effectValue(size: CGSize) ->
        ProjectionTransform {
        let scale = scaleSpline[x: Double(pct)]
        let xOffset = xSpline[x: Double(pct)]

        let yOffset = UIScreen.main.bounds.height/2 -
            pct * UIScreen.main.bounds.height/4*3

        let transTrasf = CGAffineTransform(
            translationX: xPosition + CGFloat(xOffset),
            y: yOffset)
        let scaleTrasf = CGAffineTransform(
            scaleX: CGFloat(scale),
            y: CGFloat(scale))
        return ProjectionTransform(
            scaleTrasf.concatenating(transTrasf))
    }
}
```

Finally, running the app, we can see our nice stream of floating hearts:

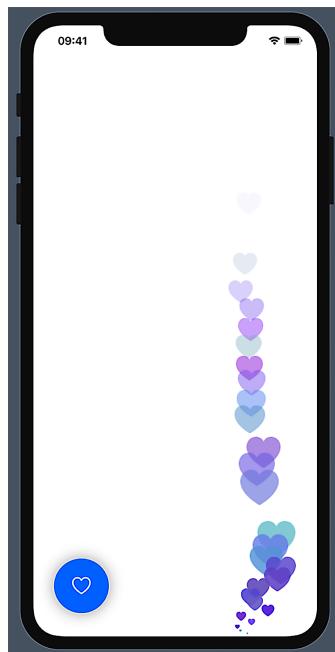


Figure 8.20 – A stream of floating hearts

How it works...

The basic idea of this recipe is to have four animations in parallel:

- Moving from the bottom to the top
- Fading
- Scaling up and down
- Slightly shaking while moving up

Also, we want to have the scaling and the shaking following a curve that we create given a few known points. The technique to create a curve given a set of points is called **interpolation**, and for that, we use the `SwiftCubicSpline` library.

For example, the scale curve starts from zero, it grows to a maximum, and then it settles down to an intermediate value. You can see the curve in the following figure:

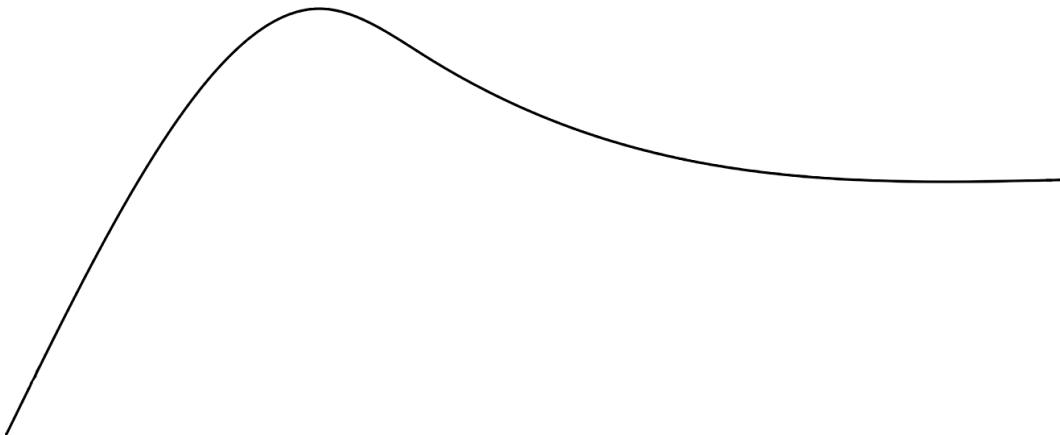


Figure 8.21 – Scaling curve

On the other hand, the curve for the horizontal movement is a sort of zigzag, which you can see in the following figure:

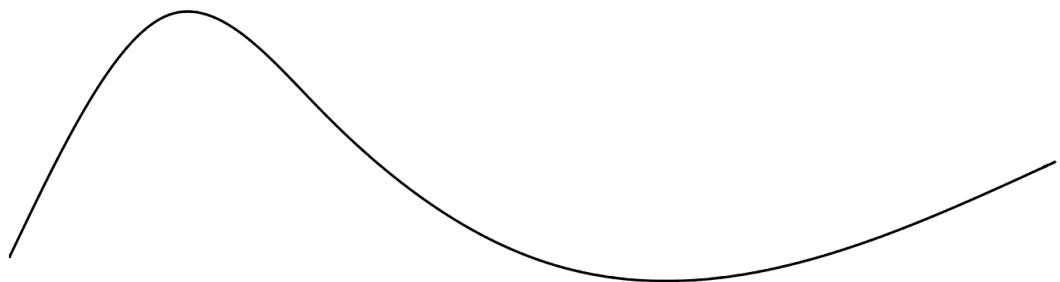


Figure 8.22 – Horizontal movement curve

While it is simple to animate the opacity because the curve is linear, there isn't a way to use a custom curve.

We circumvent this limitation using `GeometryEffect`. `GeometryEffect` is a view modifier that returns a transformation to apply. The trick here is to consider the *time* (called `pct` as a variable) as an animatable entity so that it will be called at every step of the animation.

In the `effectValue()` function, we are then calculating the required transformations – scaling, vertical moving, and horizontal shaking – at a given point in time, and we return it as a single transformation to apply to the component it is attached to.

There is one last thing to note, which is the way we remove the hearts from the list. As you can see in *step 4*, and noted here in the following code, we use a delayed callback to remove the hearts:

```
func new() {
    let heart = Heart()
    all.append(heart)
    DispatchQueue.main.asyncAfter(deadline: .now() + 10.0) {
        self.all.remove(object: heart)
    }
}
```

The goal is to remove the hearts when the animation has finished. But in this version of SwiftUI, there is no way to attach a callback to be called on animation completion, so we dispatch a delayed callback after some time, making it big enough to be sure that it is removed after the animation has finished.

I'll admit that it is a bit hacky, but it is effective.

See also

You can find more information on the use of `GeometryEffect` in this article at [objc.io](https://objc.io/blog/2019/09/26/swiftui-animation-timing-curves/), <https://www.objc.io/blog/2019/09/26/swiftui-animation-timing-curves/>, and in the second part of a series on advanced SwiftUI animation at [swiftui-lab.com](https://swiftui-lab.com/swiftui-animations-part2/), <https://swiftui-lab.com/swiftui-animations-part2/>.

Implementing a swipeable stack of cards in SwiftUI

Every now and then, an app solves a common problem in such an elegant and peculiar way that it becomes a sort of de facto way to do it in other apps as well.

I am referring to a pattern such as *pull to refresh*, which started in the **Twitter** app and then became part of iOS itself.

A few years ago, **Tinder** introduced the pattern of swipeable cards to solve the problem of indicating which cards we like and which we dislike, in a list of cards.

From then on, countless apps have applied the same visual pattern, not just in the dating sector but in every sector that needed a way to make a match between different users, including anything from business purposes, such as coupling mentors and mentees, to indicating which clothes we like for a fashion e-commerce app.

In this recipe, we are going to implement a bare-bones version of Tinder's swipeable stack of cards.

Getting ready

This recipe doesn't need any external resources, so just create a SwiftUI app called `SwipeableCards`.

How to do it...

For our simple swipeable card recipe, we are not using images but a simple gradient.

In our `User` struct, we are then going to record the gradient for that user. To calculate the width and the offset of a card, we are going to use a trick, where each user has an incremental ID; the greater IDs are at the front and the lower ones are at the back.

In that way, we can calculate the width and offset of the card using a mathematical formula and give the impression of stacked cards without using a 3D UI that is too complicated:

1. Create the `User` struct:

```
struct User: Identifiable, Equatable {  
    var id: Int  
    let firstName: String  
    let lastName: String  
    let start: Color  
    let end: Color  
}
```

2. Given that struct, add a list of users in `ContentView`:

```
struct ContentView: View {  
    @State  
    private var users: [User] = [  
        User(id: 0, firstName: "Mark",  
            lastName: "Bennett",
```

```
        start: .red, end: .green),
User(id: 1, firstName: "John",
lastName: "Lewis",
start: .green, end: .orange),
User(id: 2, firstName: "Joan",
lastName: "Mince",
start: .blue, end: .green),
User(id: 3, firstName: "Liz",
lastName: "Garret",
start: .orange, end: .purple),
]

var body: some View {
}
}
```

3. If you think that three users are not enough, add some more:

```
@State
private var users: [User] = [
//...
User(id: 4, firstName: "Lola",
lastName: "Pince",
start: .yellow, end: .gray),
User(id: 5, firstName: "Jim",
lastName: "Beam",
start: .pink, end: .yellow),
User(id: 6, firstName: "Tom",
lastName: "Waits",
start: .purple, end: .blue),
User(id: 7, firstName: "Mike",
lastName: "Rooney",
start: .black, end: .gray),
User(id: 8, firstName: "Jane",
lastName: "Doe",
```

```
        start: .red, end: .green),
    ]
```

4. Then, in the `ContentView` body, add the list of `CardView`:

```
var body: some View {
    GeometryReader { geometry in
        ZStack {
            ForEach(users) { user in
                if user.id > users.maxId - 4 {
                    CardView(user: user, onRemove: { removedUser in
                        users.removeAll { $0.id == removedUser.id }
                    })
                        .animation(.spring(), value: users)
                        .frame(width: users
                            .cardWidth(in: geometry,
                                userId: user.id),
                                height: 400)
                        .offset(x: 0,
                            y: users.cardOffset(
                                userId: user.id))
                }
            }
        } .padding()
    }
}
```

5. Add an extension method to the array of users to calculate `maxId`, which is used to limit the number of visible cards to four, and the width and offset of a card given an ID:

```
extension Array where Element == User {
    var maxId: Int { map { $0.id }.max() ?? 0 }

    func cardOffset(userId: Int) -> Double {
        Double(count - 1 - userId) * 8.0
    }
}
```

```

func cardWidth(in geometry: GeometryProxy,
               userId: Int) -> Double {
    geometry.size.width - cardOffset(userId: userId)
}
}

```

6. Finally, implement CardView, starting with the private variables:

```

struct CardView: View {
    @State
    private var translation: CGSize = .zero
    private var user: User
    private var onRemove: (_ user: User) -> Void

    private var threshold: CGFloat = 0.5

    init(user: User, onRemove: @escaping (_ user:
        User)
        -> Void) {
        self.user = user
        self.onRemove = onRemove
    }

    var body: some View {
    }
}

```

7. Then, let's add VStack with the various components in body, wrapping it in GeometryReader, which will be used in the DragGesture object:

```

var body: some View {
    GeometryReader { geometry in
        VStack(alignment: .leading, spacing: 20) {
            Rectangle()
                .fill(LinearGradient(gradient:
                    Gradient(colors: [user.start,
                                     user.end]),
                startPoint: .topLeading,

```

```
        endPoint: .bottomTrailing))
    .cornerRadius(10)
    .frame(width: geometry.size.width - 40,
           height: geometry.size.height * 0.65)
    Text("\(user.firstName) \(user.lastName)")
        .font(.title)
        .bold()
    }
    .padding(20)
    .background(Color.white)
    .cornerRadius(8)
    .shadow(radius: 5)
    .animation(.spring(), value: translation)
    .offset(x: translation.width, y: 0)
}
```

8. After the `.offset()` modifier, we must add a `.rotationEffect()` modifier to give the effect of rotating the card when it is dragged toward the border of the screen, and `DragGesture` to change the translation and remove the card or release it when the finger leaves the screen:

```
.rotationEffect(.degrees(
    Double(translation.width /
           geometry.size.width) * 20),
    anchor: .bottom)
.gesture(
    DragGesture()
        .onChanged {
            translation = $0.translation
        }.onEnded {
            if $0.percentage(in: geometry) >
                self.threshold {
                    onRemove(user)
                } else {
                    translation = .zero
                }
        }
)
```

```
    }  
)  
}
```

9. The last thing that is missing is the percentage convenience function in the `DragGesture` value. Add it with the following code:

```
extension DragGesture.Value {  
    func percentage(in geometry: GeometryProxy) ->  
        Double {  
        abs(translation.width / geometry.size.width)  
    }  
}
```

Running the app, we can smoothly swipe to the left or right, and when a swipe is more than 50%, the card disappears from the stack:

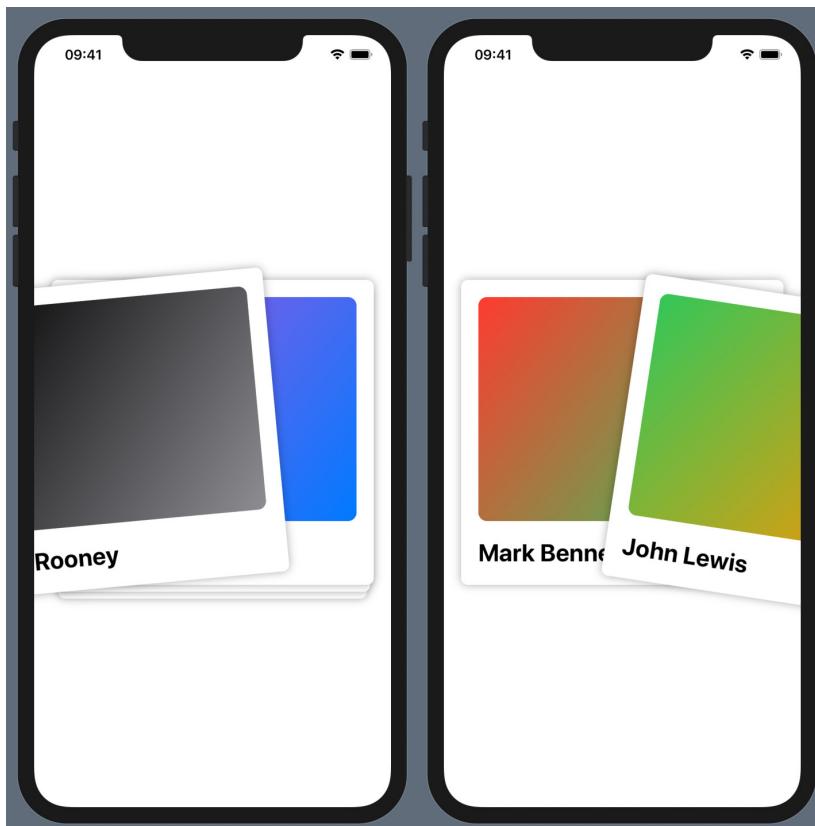


Figure 8.23 – Stack of swipeable cards

How it works...

Even though the code looks a bit long, in reality, it is very simple.

Let's start from *step 4* where the `ContentView` body is populated. Note that we are showing only the last 4 cards, using the following condition:

```
if user.id > self.users.maxId - 4 {
```

If you want to show more cards, change 4 to something else.

Another thing to notice is that in the callback we are passing to `CardView`, we are removing the card from our list. The `onRemove` parameter is called by `CardView` when it is released, and it is translated from the origin to more than 50% of its size. In that case, the card is very close to the left or right border of the device and so it must be removed.

Finally, note that the vertical offset and the width are calculated depending on the ID of the user.

In *step 5*, we are calculating the width and offset as a simple proportion of the ID of the user, assuming that they are discrete and incremental.

Step 6 is where most of the logic resides. Let's skip almost all the code, which is basically configuring the layout of the card, and concentrate on the gesture. The `onChange` callback in the `DragGesture` object sets the `@State` variable `translation` to the value of the dragged object. The `translation` variable is then used to move the card horizontally and to rotate it around the Z axis slightly. When the drag terminates, we calculate the current percentage: if it is greater than 50%, we call the `onRemove` callback; otherwise, we set the `translation` variable back to 0.

That is pretty much it. If you understand the parts we have just discussed, you'll be able to apply the same concepts to many other animations.

9

Driving SwiftUI with Data

In this chapter, we'll learn how to manage the state of single or multiple Views. In the declarative world of SwiftUI, you should consider Views as functions of their state, whereas in UIKit, in the imperative world, you are telling a view what it has to do depending on the state. In SwiftUI, Views react to changes. SwiftUI has two ways of reaching these goals:

- Using the binding property wrappers that SwiftUI provides
- Using **Combine**, the framework that Apple introduced to implement functional reactive programming

Property wrappers are a way to decorate a property and were introduced in Swift 5.1. There are four different ways of using them in SwiftUI:

- **@State**: To change state variables that belong to the same view. These variables should not be visible outside the view and should be marked as **Private**.
- **@ObservedObject**: To change state variables for multiple but connected Views; for example, when there is a parent-child relationship. These properties must have reference semantics.

- `@StateObject`: To encapsulate the mutable state of a View in an external class. It behaves like an `@ObservedObject`, but its life cycle isn't tied to the life cycle of the View that created it. It will stay allocated if the View is destroyed and then recreated because of a re-rendering.
- `@EnvironmentObject`: When the variables are shared between multiple and unrelated Views and defined somewhere else in the app. For example, you can set the color and font themes as common objects that will be used everywhere in the app.

Under the hood, some of these property wrappers use Combine to implement their functionalities. However, for more complex interactions, you can use Combine directly by exploiting its capabilities, such as chaining streams and filtering.

In this chapter, we'll look at recipes on how to use the **state binding** property wrappers, with some simple, but realistic recipes. If you have already read the recipes in the previous chapters, you will have already encountered these property wrappers, but now is the time to concentrate on them and understand them better.

Using the previous property wrappers, you should be able to create stateless Views that change when their stateful model changes.

By the end of the chapter, you should be able to decide which property wrappers to use to bind the state to the components, depending on the relationships between the state and multiple or single Views.

In *Chapter 10, Driving SwiftUI with Combine*, we'll learn how to use Combine to drive changes in SwiftUI Views.

In this chapter, we are going to learn how to use the SwiftUI binding mechanism to populate and change the Views when data changes. We will cover the following recipes:

- Using `@State` to drive a View's behavior
- Using `@Binding` to pass a state variable to child Views
- Implementing a `CoreLocation` wrapper as `@ObservedObject`
- Using `@StateObject` to preserve the model's life cycle
- Sharing state objects with multiple Views using `@EnvironmentObject`

Technical requirements

The code in this chapter is based on Xcode 13, with iOS 15 as the minimum iOS target.

You can find the code for this chapter in this book's GitHub repository at <https://github.com/PacktPublishing/SwiftUI-Cookbook-2nd-Edition/tree/main/Chapter09-Driving-SwiftUI-with-data>.

Using @State to drive a View's behavior

As we mentioned in the introduction, when a state variable belongs only to a single view, its changes are bound to the components using the `@State` property wrapper.

To understand this behavior, we are going to implement a simple to-do list app, where a static set of to-dos are changed to *done* when we tap on the row.

In the next recipe, *Using @Binding to pass a state variable to child Views*, we'll expand on this recipe, adding the possibility of adding new to-dos.

Getting ready

Let's start this recipe by creating a SwiftUI app called `StaticTodoList`.

How to do it...

To demonstrate the use of the `@State` variable, we are going to create an app that holds its state in a list of `Todo` structs: each `Todo` can be either `undone` or `done`, and we can change its state by tapping on the related row.

When the user taps on one row in the UI, they change the `done` state in the related `Todo` struct:

1. Let's start by adding the basic `Todo` struct:

```
struct Todo: Identifiable {
    let id = UUID()
    let description: String
    var done: Bool
}
```

2. Then, in ContentView, add the list of to-dos, marking them with @State:

```
struct ContentView: View {  
    @State  
    private var todos = [  
        Todo(description: "review the first chapter",  
              done: false),  
        Todo(description: "buy wine", done: false),  
        Todo(description: "paint kitchen", done:  
              false),  
        Todo(description: "cut the grass", done:  
              false),  
    ]  
    //..  
}
```

3. Now, render the to-dos, striking a line through the description if it is complete, and changing the checkmark accordingly. Do this by adding the following code:

```
var body: some View {  
    List($todos) { $todo in  
        HStack {  
            Text(todo.description)  
                .strikethrough(todo.done)  
            Spacer()  
            Image(systemName:  
                  todo.done ?  
                  "checkmark.square" :  
                  "square")  
        }  
    }  
}
```

4. Then, add a tap gesture to the Todo component to change the state, which shows whether a task has been completed:

```
HStack {  
    //...  
}  
.contentShape(Rectangle())  
.onTapGesture {  
    todo.done.toggle()  
}
```

By running the app, you can see that the way the `todo` items look changes when you tap on a row:

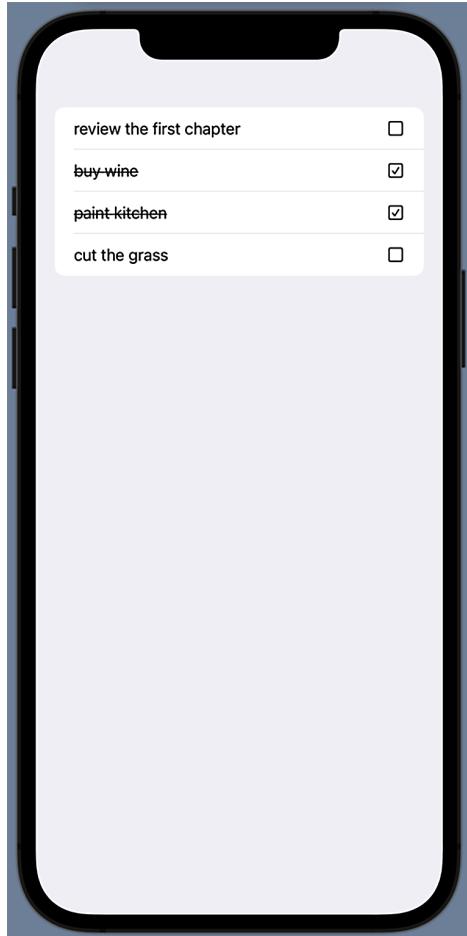


Figure 9.1 – Different to-do renderings, depending on their doneness

How it works...

If there is only one recipe that you understand and remember from this chapter, I believe that this is it.

Even if it is really simple, it depicts the change of mindset needed to understand the dynamic behavior of SwiftUI. Instead of dictating the changes to the UI, we are changing the state, and this is reflected in the UI.

Another thing to note is that every property that's used in the `body` function of a `View` is immutable, and we cannot change it in place. If we must mutate its value – with a `.toggle()`, in our case – we need to use a special syntax to refer to the properties in a mutable way.

In our example, the intuitive way of iterating over a list of `todos` would have been something like the following:

```
List(todos) { todo in
    //...
    Text(todo.description)
    //...
    .onTapGesture {
        todo.done.toggle()
    }
}
```

However, the instruction to change the value of the `done` property would have failed to compile:

```
todo.done.toggle()
//error: cannot use mutating member on immutable value:
'todo' is a 'let' constant
```

As we mentioned previously, `todo` is a constant, so we can't mutate its value.

Fortunately, SwiftUI provides a way of passing a value as a reference, even if it is a `struct`. The original container must be decorated with a `$`, and the internal variable must also have this `$`. Our code will look like this:

```
List($todos) { $todo in
```

In this way, `todo` is now mutable – or, even better – a reference to the original entry in the array, and we can safely change its value.

Finally, you probably noticed the `.contentShape()` modifier, which we applied to the row. This is necessary to give the whole row as a hit area to the gesture. Otherwise, only the text would have been sensitive to the touches.

See also

You can find more information in the Apple tutorial, *State and Data Flow*, at https://developer.apple.com/documentation/swiftui/state_and_data_flow.

It is also worthwhile watching the WWDC 2019 video *Data Flow through SwiftUI*, which you can find here: <https://developer.apple.com/videos/play/wwdc2019/226/>.

Using @Binding to pass a state variable to child Views

In the *Using @State to drive a View's behavior* recipe, you saw how to use an `@State` variable to change a UI. But what if we want to have another view that changes that `@State` variable?

Given that an array has a value-type semantic, if we pass down the variable, Swift creates a copy of the variable, and if the variable is mutated, changes are not reflected in the original.

SwiftUI solves this with the `@Binding` property wrapper, which, in a certain way, creates a reference semantic for specific structs.

To explore this mechanism, we are going to create an extension of the `TodoList` app that we created in the *Using @State to drive a View's behavior* recipe, where we are going to add a child view that allows us to add a new to-do to the list.

Getting ready

The starting point for this project is the final code of the previous recipe, so you can use the same `StaticTodoList` project you used previously.

If you want to keep the recipes separate, you can create a new SwiftUI project called `DynamicTodoList`.

How to do it...

The main difference compared with the previous `StaticTodoList` project is the addition of an `InputView`, which allows the user to add a new task to perform.

The first steps are the same as the previous recipe, so feel free to skip them if you are starting your code from the source of that recipe:

1. Let's start by adding the basic `Todo` struct:

```
struct Todo: Identifiable {  
    let id = UUID()  
    let description: String  
    var done: Bool  
}
```

2. Then, in `ContentView`, add the list of to-dos, marking them with `@State`:

```
struct ContentView: View {  
    @State  
    private var todos = [  
        Todo(description: "review the first chapter",  
              done: false),  
        Todo(description: "buy wine", done: false),  
        Todo(description: "paint kitchen", done:  
              false),  
        Todo(description: "cut the grass", done:  
              false),  
    ]  
    //..  
}
```

3. Now, render the to-dos, striking a line through the description if they are complete, and changing the checkmark accordingly. Do this by adding the following code:

```
var body: some View {  
    List($todos) { $todo in  
        HStack {  
            Text(todo.description)  
                .strikethrough(todo.done)  
            Spacer()  
        }  
    }  
}
```

```
        Image(systemName:  
            todo.done ?  
                "checkmark.square" :  
                "square")  
    }  
}  
}
```

4. Then, add a tap gesture to the Todo component to change the state of doneness:

```
HStack {  
    //...  
}  
.contentShape(Rectangle())  
.onTapGesture {  
    todo.done.toggle()  
}
```

5. Create a new component that we will use to add a new to-do task:

```
struct InputTodoView: View {  
    @State  
    private var newTodoDescription: String = ""  
  
    @Binding  
    var todos: [Todo]  
  
    var body: some View {  
        HStack {  
            TextField("Todo", text:  
                $newTodoDescription)  
.textFieldStyle(RoundedBorderTextFieldStyle())  
            Spacer()  
            Button {  
                //...  
            } label: {  
                Text("Add")  
            .padding(.horizontal, 16)
```

```
        .padding(.vertical, 8)
        .foregroundColor(.white)
        .background(.green)
        .cornerRadius(5)
    }
}
.frame(height: 60)
.padding(.horizontal, 24)
.padding(.bottom, 30)
.background(Color.gray)
}
}
```

6. We have finished with the component layout. Now, add the action to the button. Here, we will create a new task if the description is not empty:

```
//...
Button {
    guard !newTodoDescription.isEmpty else {
        return
    }
    todos.append(Todo(description: newTodoDescription,
                      done: false))
    newTodoDescription = ""
} label: {
    //...
}
```

7. If you run the app now, you won't see any difference compared to the previous app: can you spot what's wrong?

Exactly – we haven't added `InputTodoView` to `ContentView` yet!

Before adding it, embed the `List` view in a `ZStack` that ignores the safe area at the bottom:

```
var body: some View {
    ZStack(alignment: .bottom) {
        List($todos) { $todo in
            //...
        }
    }
}
```

```
    }
    .edgesIgnoringSafeArea(.bottom)
}
```

8. Finally, put `InputTodoView` in `ZStack`, along with the `List` view:

```
ZStack(alignment: .bottom) {
    List($todos) { $todo in
        // ...
    }
    InputTodoView(todos: $todos)
}
```

Finally, the app is improved – not only can we change the state of the to-dos, but we can also add new to-dos:

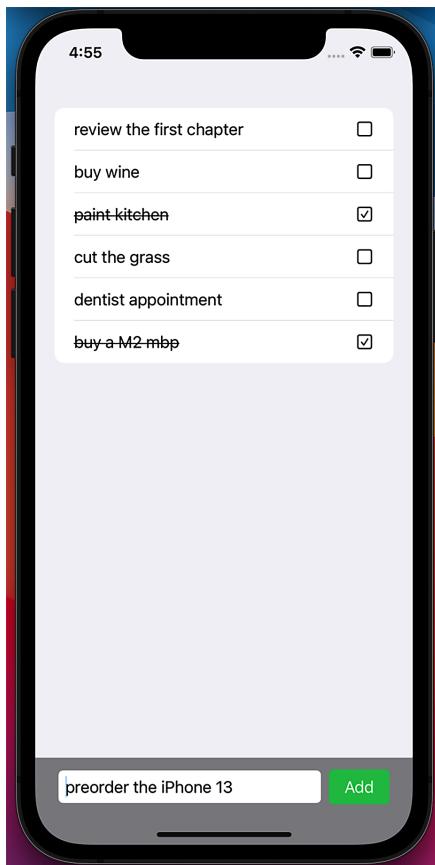


Figure 9.2 – Adding new to-dos to the list

How it works...

As we mentioned in the introduction to the recipe, when a child view has to change the state of a parent view, passing the variable directly is not enough. Because of the value type semantic, a copy of the original variable is made.

SwiftUI uses the `@Binding` property wrapper, which creates a two-way binding:

- Any changes in the parent's variable are reflected in the child's variable.
- Any changes in the child's variable are reflected in the parent's variable.

This is achieved by using the `@Binding` property wrapper in the child's definition and by using the `$` symbol when the parent passes the variable. This means that it isn't passing the variable itself, but its reference, similar to using `&` for the `inout` variables in normal Swift code.

Implementing a **CoreLocation** wrapper as `@ObservedObject`

We mentioned in the introduction to this chapter that `@State` is used when the state variable has value-type semantics. This is because any mutation of the property creates a new copy of the variable, which triggers a rendering of the view's body, but what about a property with reference semantics?

In this case, any mutation of the variable is applied to the variable itself and SwiftUI cannot detect the variation by itself.

In this case, we must use a different property wrapper, `@ObservedObject`, and the observed object must conform to `ObservableObject`. Furthermore, the properties of this object that will be observed in the view must be decorated with `@Published`. In this way, when the properties mutate, the view is notified, and the body is rendered again.

This will also help if we want to bridge iOS foundation objects to the new SwiftUI model, such as **CoreLocation** functionalities. **CoreLocation** is the iOS framework that determines the device's geographic location. Location determination requires always having the mechanism on, even if it's in the background, and comes with associated privacy concerns. For this reason, explicit permission must be granted by the user.

In this recipe, we'll see how simple it is to wrap `CLLocationManager` in an `ObservableObject` and consume it in a SwiftUI view.

Getting ready

Let's create a SwiftUI project called `SwiftUICoreLocation`.

For privacy reasons, every app that needs to access the location of the user must seek permission before accessing it. To allow this, a message should be added to the *Privacy – Location When in Use Usage Description* key in the **Info** part of the project:

1. Select the main target (**SwiftUICoreLocation**) in the **PROJECT > Info** tab in the detail view:

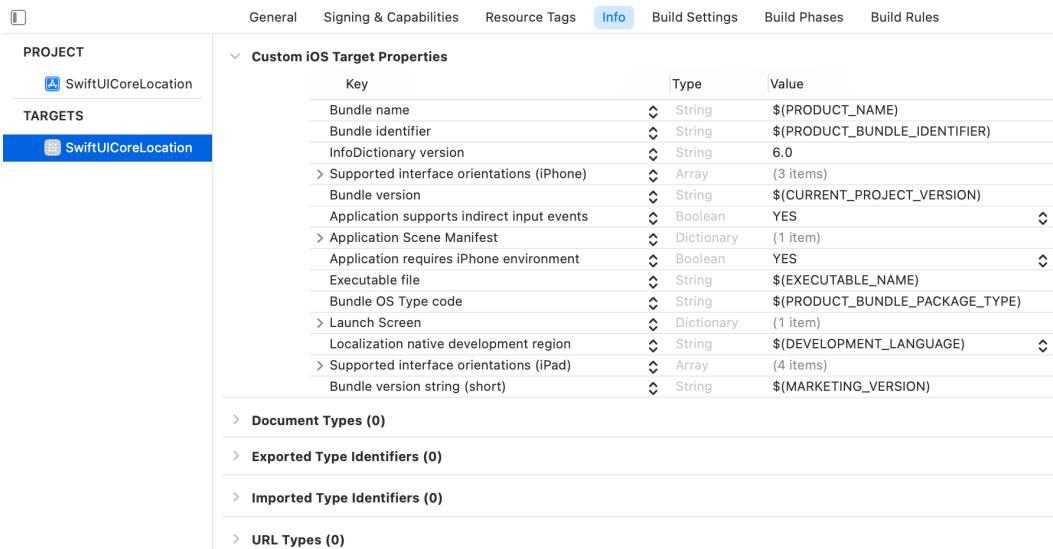


Figure 9.3 – Selecting the project Info tab

2. Right-click the table to select **Raw Keys & Values**:

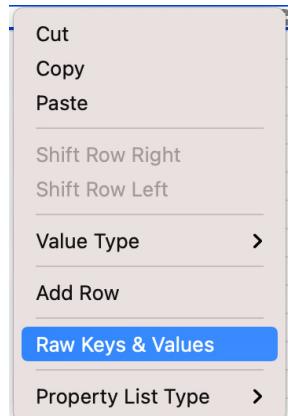


Figure 9.4 – Selecting Raw Keys & Values in the Info tab

3. Add the **NSLocationWhenInUseUsageDescription** key and a value with a message to the user:

CFBundleExecutable	String	\$(EXECUTABLE_NAME)
NSLocationWhenInUseUsageDescription	String	Enable Location Services please
CFBundlePackageType	String	\$(PRODUCT_BUNDLE_PACKAGE_TYPE)

Figure 9.5 – Message to the user to enable location services

How to do it...

In this recipe, we are simply wrapping `CLLocationManager` in an `ObservableObject` and publishing two properties:

- The location's status
- The current location

A SwiftUI view will present the user location and subsequent location updates when the user moves more than 10 meters:

1. Let's start by creating `ObservableObject`, which owns `CLLocationManager` and exposes the two state variables:

```
import CoreLocation
class LocationManager: NSObject, ObservableObject {
    private let locationManager = CLLocationManager()
    @Published
    var status: CLAuthorizationStatus?
    @Published
    var current: CLLocation?
}
```

2. Then, implement the `init()` function, where we will configure `CLLocationManager` to react when the position changes by 10 meters. We will begin by updating the location's detection:

```
class LocationManager: NSObject, ObservableObject {
    //...
    override init() {
        super.init()
        self.locationManager.delegate = self
        self.locationManager.distanceFilter = 10
    }
}
```

```
    self.locationManager.desiredAccuracy =  
  
    kCLLocationAccuracyBest  
    self.locationManager.  
requestWhenInUseAuthorization()  
    self.locationManager.startUpdatingLocation()  
}  
}
```

3. The app doesn't compile because we are setting LocationManager as a delegate to CLLocationManager without conforming it to the appropriate protocol. Undertake this task using the following code:

```
extension LocationManager: CLLocationManagerDelegate {  
    func locationManagerDidChangeAuthorization(_  
        manager:  
        CLLocationManager) {  
        status = manager.authorizationStatus  
    }  
  
    func locationManager(_ manager: CLLocationManager,  
                        didUpdateLocations  
                        locations: [CLLocation]) {  
        guard let location = locations.last else {  
            return  
        }  
        self.current = location  
    }  
}
```

4. Before using the class in ContentView, add a convenient extension to CLAuthorizationStatus to format the status in a descriptive way:

```
extension Optional where Wrapped ==  
CLAuthorizationStatus {  
    var description: String {  
        guard let self = self else {  
            return "unknown"  
        }  
    }  
}
```

```
        switch self {
            case .notDetermined:
                return "notDetermined"
            case .authorizedWhenInUse:
                return "authorizedWhenInUse"
            case .authorizedAlways:
                return "authorizedAlways"
            case .restricted:
                return "restricted"
            case .denied:
                return "denied"
            @unknown default:
                return "unknown"
        }
    }
}
```

5. Do the same for the `CLLocation` class by adding the following extension:

```
extension Optional where Wrapped == CLLocation {
    var latitudeDescription: String {
        guard let self = self else {
            return "-"
        }
        return "\((self.coordinate.latitude))"
    }
}

var longitudeDescription: String {
```

```
guard let self = self else {
    return "-"
}
return "\(self.coordinate.longitude)"
}
}
```

6. Finally, we must set LocationManager as an observed object in ContentView:

```
struct ContentView: View {
    @ObservedObject
    private var locationManager = LocationManager()

    var body: some View {
        VStack {
            Text("Status:
                \(locationManager.status.description)")
            HStack {
                Text("Latitude:
                    \(locationManager.current
                        .latitudeDescription))")
                Text("Longitude:
                    \(locationManager.current
                        .longitudeDescription))")
            }
        }
    }
}
```

When running the app for the first time, after granting permission for the location services, you should see your coordinates printed in the view. If you move more than 10 meters, you should see the values update:

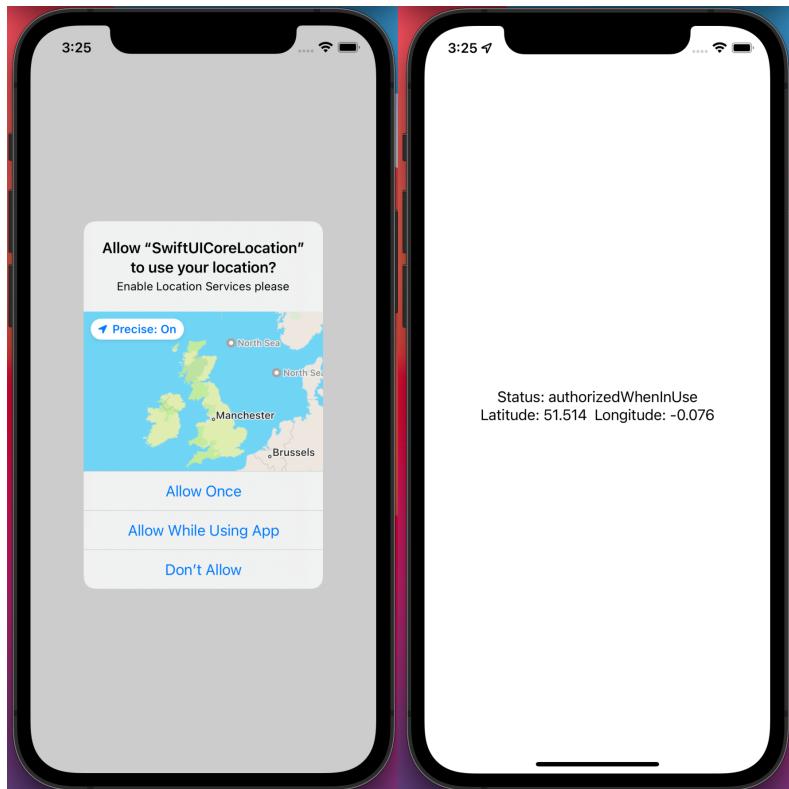


Figure 9.6 – Showing user locations with SwiftUI

How it works...

Using a class as a state variable for SwiftUI is a matter of creating a subclass of `ObservableObject` and deciding what properties will trigger a re-rendering of a view.

You can consider the `@Published` property wrapper as a sort of automatic property observer that triggers a notification when its value changes. The view, with the `@ObservedObject` decoration, registers the changes in the published properties. Every time one of these mutates, it renders the body again.

`@Published` is syntactic sugar for a **Combine** publisher. If you are curious to see what you can do with **Combine**, we'll introduce it in *Chapter 10, Driving SwiftUI with Combine*, in the *Introducing Combine in a SwiftUI project* recipe.

Using @StateObject to preserve the model's life cycle

Even though it has solid foundations, SwiftUI is a relatively new framework. Apple adds new features in every release to adapt to the usage done by the community of developers.

In the first release of SwiftUI, `@ObservedObject` was provided to separate the model logic from the view logic. The usage that Apple was assuming was that the object would be injected from an external class, not created directly inside a View.

Creating an `@ObserveObject` in a View ties the life cycle of the object to the life cycle of the view.

However, the View can be destroyed and created several times while still appearing on the screen, where it should present the content of the view. When the View is destroyed, `@ObservedObject` is destroyed too, resetting its internal state.

This was a counter-intuitive behavior and most of the developers were assuming that `@ObservedObject` would keep its state until the owning view was shown on the screen.

Apple, being aware of that, added a new property wrapper called `@StateObject` to fix this problem.

In this recipe, we are going to show you this unexpected and counter-intuitive effect, and how `@StateObject` must be used to solve it.

Getting ready

Create a new SwiftUI app called Counter.

How to do it...

In this recipe, we will be implementing a simple Counter app where the counter is managed by a child View, which is contained in a Container View, along with a Text View that changes with the current time when the *Refresh* button is tapped:

1. Start creating the model for CounterView:

```
class Counter: ObservableObject {  
    @Published var value: Int = 0  
  
    func inc() {  
        value += 1  
    }  
}
```

```
    }

    func dec() {
        value -= 1
    }
}
```

2. Create a CounterView, presenting a Text with the value of its model, and two buttons to mutate it in the model:

```
struct CounterView: View {
    var body: some View {
        VStack(spacing: 12) {
            Text("\(counter.value)")
            HStack(spacing: 12) {
                Button {
                    counter.dec()
                } label: {
                    Text("-")
                        .padding()
                        .foregroundColor(.white)
                        .background(.red)
                }
                Button {
                    counter.inc()
                } label: {
                    Text("+")
                        .padding()
                        .foregroundColor(.white)
                        .background(.green)
                }
            }
        }
    }
}
```

3. To test the behavior with the model as `@ObservedObject`, as well as `@StateObject`, add both definitions, with the latter commented out:

```
struct CounterView: View {
    @ObservedObject var counter = Counter()
    // @StateObject var counter = Counter()
    var body: some View {
        //...
    }
}
```

4. In `ContentView`, add a `@State` `Date` property and present it along with `CounterView` in a `VStack` container:

```
struct ContentView: View {
    @State var refreshedAt: Date = Date()
    var body: some View {
        VStack(spacing:12) {
            Text("Refresh at ") +
            Text(refreshedAt.formatted(date: .omitted,
                time: .standard) )
            CounterView()
        }
    }
}
```

5. Finally, in the same `VStack`, add a `Button` to change the refresh date property:

```
VStack(spacing:12) {
    //...
    CounterView()
    Button {
        refreshedAt = Date()
    } label: {
        Text("Refresh")
        .padding()
        .foregroundColor(.white)
        .background(.blue)
    }
}
```

While running the app, we can increment or decrement the counter and refresh the whole Container view:

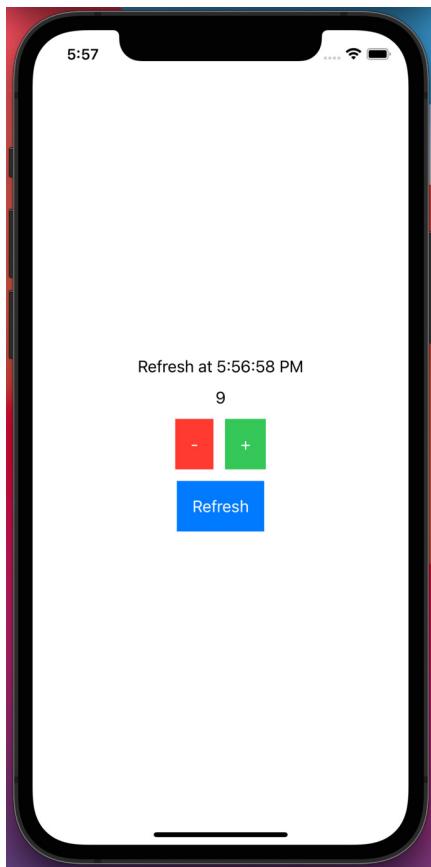


Figure 9.7 – Incrementing or decrementing a contained View

How it works...

Now, let's see how the app behaves with an `@StateObject` and with a `@ObservedObject` model.

We will start with the Counter model, which we defined as an `@ObservedObject`, by enabling the following code:

```
@ObservedObject var counter = Counter()
```

Run the app and tap on the + button a few times. When the value of the counter is different than zero, tap on the **Refresh** button to change the Text value on the top of the View.

Unexpectedly, the value of the counter becomes zero, though we were expecting to see a new time while keeping the same value of the counter. The reason for this is that when we update the `@State` property, `refreshedAt`, the change triggers a render of the body of `ContentView`. Rendering the body means that SwiftUI recreates the graph of the objects to be presented on screen, destroying and recreating them accordingly.

During the rendering, `CounterView` is recreated, and with it, the `Counter` object is destroyed and instantiated again, resetting its value.

Now, change the model to be an `@StateObject`:

```
@StateObject var counter = Counter()
```

Tap on the `+` button a few times and then on the **Refresh** button. At this point, the value of the counter will be preserved when the View that *owns* the `Counter` object is destroyed and created again.

With an `@StateObject`, we can decouple the model life cycle from the view life cycle, and it is the pattern to follow when a model is created by a View.

Sharing state objects with multiple Views using `@EnvironmentObject`

In many cases, there are dependent collaborators that must be shared between several Views, even without having a tight relationship between them. Think of a `ThemeManager`, or a `NetworkManager`, or a `UserProfileManager`.

Passing them through the chain of Views can be annoying, without even thinking of the coupling we could create. If a view doesn't need the `NetworkManager` instance, for example, it should still have it as a property in case one of its child Views needs it.

SwiftUI solves this with the concept of `Environment`, a place to add common objects – usually `ObservableObject` objects – that will be shared between a chain of Views. An `Environment` is started in the root ancestor of the view graph and can be changed further down in the chain by us adding new objects.

To present this feature, we are going to implement a basic song player with three Views:

- A View for the list of songs
- A mini player View (always on top of the Views when a song is played)
- A View for the song details

All three of these Views always have an indication of the song currently playing and a button to play or stop a song.

It's important that the Views are in sync with the actual audio player, and we must also decouple the view logic from the actual song playing.

For that, we will use a simple `MusicPlayer` object that simulates the actual song that's playing, which is saved in `Environment`.

By the end of the recipe, you will have a grasp of what to put in `Environment` and how, as well as how to separate the view logic from the business logic.

Getting ready

Let's start by creating a new SwiftUI app and calling it `SongPlayer`.

To present the songs in the player, we need some images, `cover0.png` to `cover5.png`, which you can find in this book's GitHub repository at <https://github.com/PacktPublishing/SwiftUI-Cookbook-2nd-Edition-/tree/main/Resources/Chapter09/recipe5>.

Add them to the `Assets` images and we will be ready to go:

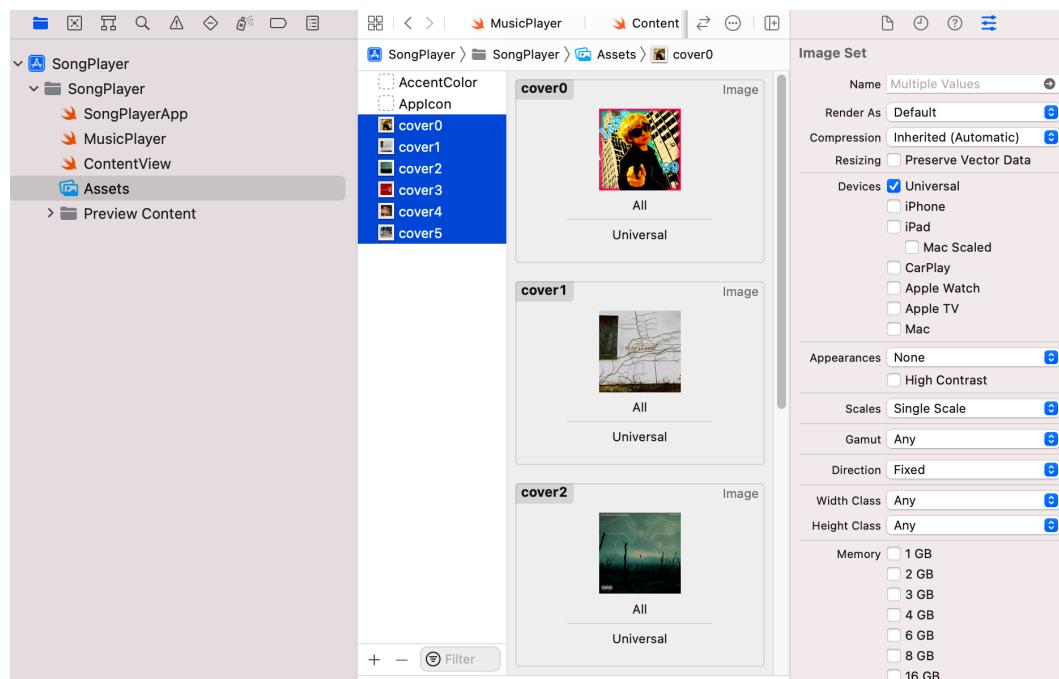


Figure 9.8 – Importing the images

How to do it...

We are going to implement three Views to present a playable song and a `MusicPlayer` class that simulates a real audio player:

1. Model the song with the following `Song` struct:

```
struct Song: Identifiable, Equatable {
    var id = UUID()
    let artist: String
    let name: String
    let cover: String
}
```

2. Before moving to the Views implementation, let's make the `MusicPlayer` class (a class that we could eventually extend to use the **Core Audio** framework to play audio) with an optional current playing song:

```
class MusicPlayer: ObservableObject {
    @Published
    var currentSong: Song?

    func pressButton(song: Song) {
        if currentSong == song {
            currentSong = nil
        } else {
            currentSong = song
        }
    }
}
```

3. Add the list of songs to the `ContentView` struct and present them in a List View:

```
struct ContentView: View {
    @EnvironmentObject
    private var musicPlayer: MusicPlayer

    private let songs = [
        Song(artist: "Luke", name: "99", cover: "cover0"),
        Song(artist: "Luke", name: "99", cover: "cover1"),
        Song(artist: "Luke", name: "99", cover: "cover2"),
        Song(artist: "Luke", name: "99", cover: "cover3"),
        Song(artist: "Luke", name: "99", cover: "cover4"),
        Song(artist: "Luke", name: "99", cover: "cover5"),
        Song(artist: "Luke", name: "99", cover: "cover6"),
        Song(artist: "Luke", name: "99", cover: "cover7"),
        Song(artist: "Luke", name: "99", cover: "cover8"),
        Song(artist: "Luke", name: "99", cover: "cover9")
    ]
}
```

```
        Song(artist: "Foxing", name: "No Trespassing",
              cover: "cover1"),
        Song(artist: "Breaking Benjamin", name: "Phobia",
              cover: "cover2"),
        Song(artist: "J2", name: "Solenoid",
              cover:"cover3"),
        Song(artist: "Wildflower Clothing",
              name: "Lightning Bottles", cover: "cover4"),
        Song(artist: "The Broken Spirits", name: "Rubble",
              cover: "cover5"),
    ]
}

var body: some View {
    NavigationView {
        List(self.songs) { song in
            //...
        }
        .listStyle(.plain)
        .navigationTitle("Music Player")
    }
}
```

4. Each row in the list must have a play button whose image depends on the song currently being played. Create a `PlayButton` component that has a connection to the shared `MusicPlayer` object:

```
struct PlayButton: View {
    @EnvironmentObject
    private var musicPlayer: MusicPlayer
    let song: Song

    private var buttonText: String {
        if song == musicPlayer.currentSong {
            return "stop"
        } else {
            return "play"
        }
    }
}
```

```

        }

    }

    var body: some View {
        Button {
            musicPlayer.pressButton(song: song)
        } label: {
            Image(systemName: buttonText)
                .font(.system(.largeTitle))
                .foregroundColor(.black)
        }
    }
}

```

5. For each song in the list, we will add a proper SongView:

```

NavigationView {
    List(self.songs) { song in
        SongView(song: song)
    }
}

```

6. The layout of each view presents the cover to the left, the author and song title in the center, and the play/stop button to the right. Also, when we tap on the cover image, we will navigate to a full-screen player view. Create the SongView struct with the following code:

```

struct SongView: View {
    @EnvironmentObject
    private var musicPlayer: MusicPlayer
    let song: Song

    var body: some View {
        HStack {
            NavigationLink(destination:
                PlayerView(song: song)) {
                Image(song.cover)
            }
        }
    }
}

```

```
        .renderingMode(.original)
        .resizable()
        .aspectRatio(contentMode: .fill)
        .frame(width: 100, height: 100)
    VStack(alignment: .leading) {
        Text(song.name)
        Text(song.artist).italic()
    }
}

Spacer()
PlayButton(song: song)
}.buttonStyle(PlainButtonStyle())
}
}
```

7. The full-screen PlayerView is a view with a bigger cover image and a **Play/Stop** button. Create it with the following code:

```
struct PlayerView: View {
    @EnvironmentObject
    private var musicPlayer: MusicPlayer
    let song: Song

    var body: some View {
        VStack {
            Image(song.cover)
                .renderingMode(.original)
                .resizable()
                .aspectRatio(contentMode: .fill)
                .frame(width: 300, height: 300)
            HStack {
                Text(song.name)
```

```

        Text(song.artist).italic()
    }
    PlayButton(song: song)
}
}
}
}
```

8. The app is almost complete, but if we try to run it, we will get a runtime error. The reason is that we are reading a value from Environment with @EnvironmentObject, but we don't initialize its value anywhere.

Inject MusicPlayer into Environment in the main scene:

```

@main
struct SongPlayerApp: App {
    var body: some Scene {
        WindowGroup {
            ContentView()
                .environmentObject(MusicPlayer())
        }
    }
}
```

9. The app now runs properly, but we still have an error when we try to open it in preview mode. Can you spot the reason?

Yes, it is because we are not running the app from the @main struct in the preview, but from the ContentView_Previews struct.

Inject a MusicPlayer instance into the preview:

```

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

10. We're almost there. The final missing component is `MiniPlayerView`, which is a view that simply wraps a `SongView` component. Add the following component:

```
struct MiniPlayerView: View {
    @EnvironmentObject
    private var musicPlayer: MusicPlayer

    var body: some View {
        if let currentSong = musicPlayer.currentSong {
            SongView(song: currentSong)
                .padding(.all, 24)
        } else {
            EmptyView()
        }
    }
}
```

11. Finally, in the `ContentView` struct, embed `NavigationView` with the songs into a `ZStack` and add `MiniPlayerView` at the bottom:

```
var body: some View {
    ZStack(alignment: .bottom) {
        NavigationView {
            //...
        }
        MiniPlayerView()
            .background(.gray)
            .offset(y: 44)
    }
}
```

By running the app, we can see that all the Views are in sync. Upon selecting different songs to play in different Views, they always present the correct **Play/Stop** button:

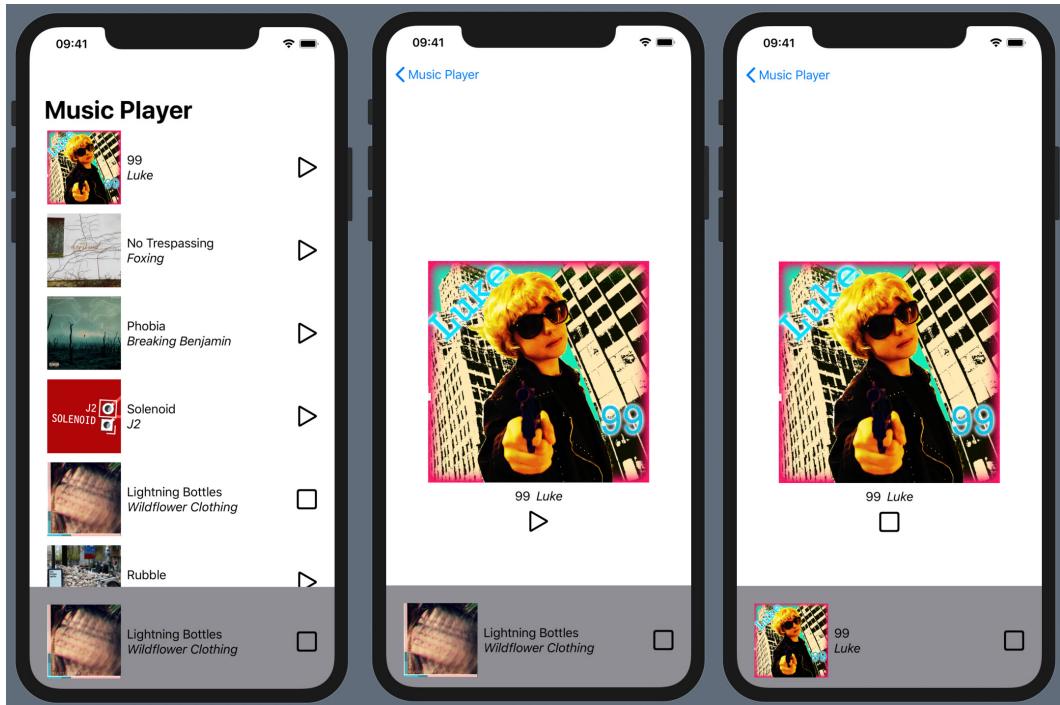


Figure 9.9 – Playing a song in different Views

How it works...

The `@Environment` feature is a nice (and official way) of resolving a problem that has often been solved using singletons or global variables: the problem of common dependencies. If you have something that could be needed in multiple unrelated Views, storing it in `Environment` could be a viable solution.

Every view has access to `Environment` set by their ancestors, and it can retrieve a value from it, marking the property with the `@EnvironmentObject` wrapper.

You can imagine this feature as being a sort of shared `HashMap`, where the keys are the type of object, and the values are an instance of the object itself.

From this, we can deduce that we cannot have two objects of the same type in `Environment`, but this shouldn't usually prove to be a problem.

Moreover, there is a way to have different objects of the same type there, but this goes beyond the scope of this book (hint: define custom keys).

See also

Although not explicitly declared by Apple, this talk by *Stephen Celis* could have been an inspiration for this feature: *How to Control the World* (<https://vimeo.com/291588126>).

10

Driving SwiftUI with Combine

In this chapter, we'll learn how to manage the state of SwiftUI Views using Combine. In the **Worldwide Developers Conference (WWDC)** 2019, Apple not only introduced SwiftUI but also Combine, a perfect companion to SwiftUI for managing the declarative change of state in Swift.

In recent years, given the success of **Functional Reactive Programming (FRP)** in different sectors of the industry, the same concept is being used in the iOS ecosystem. It was first implemented with **ReactiveCocoa**, the original framework in Objective-C. That framework was converted into Swift with **ReactiveSwift**. Finally, it was converted into **RxSwift**, which became the default framework for performing FRP in iOS.

In typical Apple way, Apple took the best practices that have matured over years of trial and error from the community, and instead of acquiring either **ReactiveSwift** or **RxSwift**, Apple decided to reimplement their concepts, simplify their version, and specialize it for mobile development. That's how **Combine** was born!

An in-depth examination of Combine is outside the scope of this book, but in the recipes in this chapter, we'll provide a shallow introduction to Combine and how to use it.

Just as SwiftUI is a declarative way of describing a **User Interface (UI)** layout, Combine is a declarative way of describing the flow of changes.

The basic foundation of Combine is the concept of *Publisher* and *subscriber*. A *Publisher* emits events, pushed by something else that can be a part of an iOS **Software Development Kit (SDK)**, such as the network layer, which pushes events into the `dataTaskPublisher` publisher of the `URLSession` instance, or another part of the app that you are implementing; for example, the publisher of a *ViewModel*. One or more *subscribers* can observe that publisher and react when a new event is published. You can consider this mechanism as a sort of *Observable/Observer* pattern on steroids.

The main difference with a plain *Observable/Observer* pattern is that in the case of the *Observer* pattern, the interface of the *Observable* object is custom and depends on the particular object, whereas in the Combine case, the interface is standard, and the only difference is the type of events and errors that a publisher emits.

Given that the interface is standard, it is possible to *combine* (hence the name) multiple *Publishers* into one or apply modifiers such as filters, retry-ers, and so on.

I admit that this may sound confusing and complicated, but if you try out a few recipes in this chapter, you will understand the philosophy of reactive programming and, in particular, the way it is implemented in Combine.

By the end of the chapter, you will know when and how to use Combine (or if it is overkill for your needs), and when you need to learn about Combine in more depth.

In this chapter, we are going to learn how to use the SwiftUI binding mechanism and Combine to populate and change the views when data changes.

We are going to cover the following recipes:

- Introducing Combine in a SwiftUI project
- Managing the memory in Combine to build a timer app
- Validating a form using Combine
- Fetching remote data using Combine and visualizing it in SwiftUI
- Debugging an app based on Combine
- Unit testing an app based on Combine

Technical requirements

The code in this chapter is based on Xcode 13 and iOS 15.

You can find the code for this chapter in this book's GitHub repository at <https://github.com/PacktPublishing/SwiftUI-Cookbook-2nd-Edition/tree/main/Chapter10-Driving-SwiftUI-with-Combine>.

Introducing Combine in a SwiftUI project

In this recipe, we are going to add publish support to **CoreLocation**, which is an Apple framework that provides location functionalities such as the current position of the user. The `CLLocationManager` framework class will emit status and location updates, which will be observed by a SwiftUI View. It is a different implementation of the problem that was presented in the *Implementing a CoreLocation wrapper as @ObservedObject* recipe of *Chapter 9, Driving SwiftUI with Data*.

Usually, when a reactive framework is used, instead of the common **Model-View-Controller (MVC)** architecture, people tend to use **Model-View-ViewModel (MVVM)**.

In this architecture, the view doesn't have any logic. Instead, this is encapsulated in the intermediate object, the *ViewModel*, which has the responsibility of being the model for the view and talking to the business logic model, services, and so on to update that model. For example, it will have a property for the current location that the view will use to present it to the user, and at the same time, the *ViewModel* will talk to the *LocationService* hiding that relationship from the View. In this way, the UI is completely decoupled from the business logic, and the *ViewModel* acts as an adapter between the two.

The *ViewModel* is bound to the view with a one-to-one relationship between the basic components and the properties of the *ViewModel*.

If this is the first time you have heard of MVVM and you are confused by this description, then don't worry – this recipe is really simple, and you'll understand it while implementing it. I promise!

Getting ready

Let's create a SwiftUI project called `CombineCoreLocationManager`.

For privacy reasons, every app that needs to access the location of the user must seek permission before accessing it. To allow this, a message should be added to the *Privacy – Location When in Use Usage Description* key in the **info** part of the project:

1. Select the main target in the **PROJECT > Info** tab in the detail view:

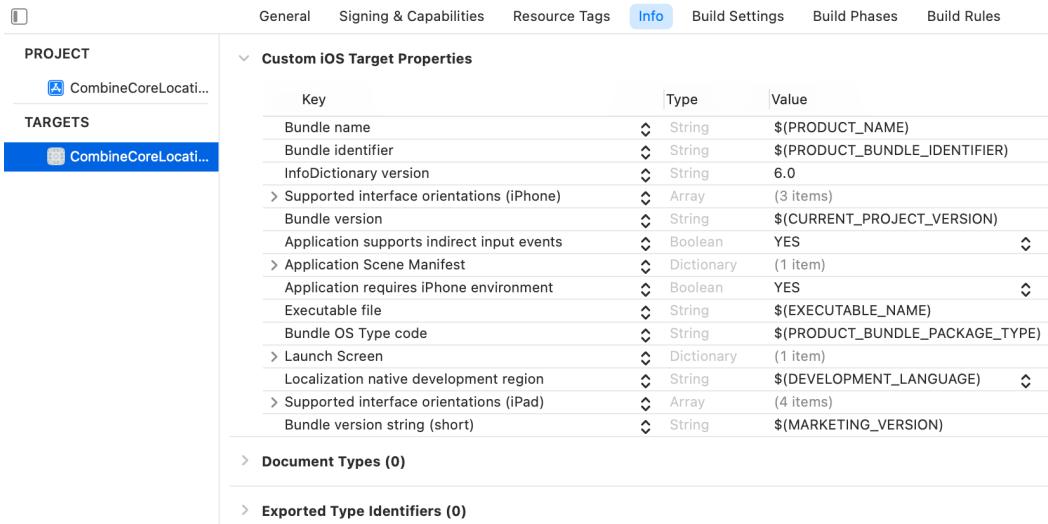


Figure 10.1 – Selecting the Info tab

2. Right-click the table to select **Raw Keys & Values**:

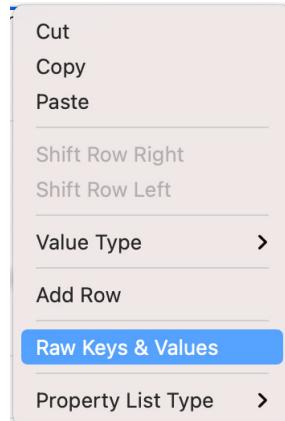


Figure 10.2 – Selecting Raw Keys & Values in the Info tab

3. Add the **NSLocationWhenInUseUsageDescription** key and a value with a message to the user:

CFBundleExecutable	String	<code>\$(EXECUTABLE_NAME)</code>
NSLocationWhenInUseUsageDescription	String	Enable Location Services please
CFBundlePackageType	String	<code>\$(PRODUCT_BUNDLE_PACKAGE_TYPE)</code>

Figure 10.3 – Message to the user to enable location services

How to do it...

As I mentioned in the introduction to this recipe, we are going to implement this recipe while following the MVVM architecture.

We are going to implement three components:

- The status and location view in `ComponentView`: The **View**
- A `LocationViewModel` class that interacts with the model and updates the view: The **ViewModel**
- A `CLLocationManager` service that listens to the location's changes and sends events when they happen: The **Model**

Let's move on to the code, starting with the Model:

1. First, import `CoreLocation` and `Combine` and implement a wrapper for `CLLocationManager`:

```
import CoreLocation
import Combine

class LocationManager: NSObject {
    enum LocationError: String, Error {
        case restricted
        case denied
        case unknown
    }

    private let locationManager = CLLocationManager()

    override init() {
        super.init()
    }
}
```

```
        self.locationManager.delegate = self
        self.locationManager.desiredAccuracy =
kCLLocationAccuracyBest
        self.locationManager.
requestWhenInUseAuthorization()
    }

    func start() {
        locationManager.startUpdatingLocation()
    }
}
```

2. So far, there is nothing particularly *reactive*. So, let's add two publishers: one for the status and the other for the location updates. As you can see, they are called *Subjects*: we'll explain what they are in the *How it works...* section. For the time being, just consider them as *Publishers*. Their declaration shows the types of events and errors:

```
class LocationManager: NSObject {
    //...
    let statusPublisher =
        PassthroughSubject<CLAuthorizationStatus,
                           LocationError>()

    let locationPublisher =
        PassthroughSubject<CLLocation?,
                           Never>()
    //...
}
```

3. Now, let's send the events to the publishers. We will implement the `CLLocationManagerDelegate` functions, where we push events in the publishers:

```
extension LocationManager: CLLocationManagerDelegate {
    func locationManagerDidChangeAuthorization(_
        manager:
        CLLocationManager) {
```

```
        switch manager.authorizationStatus {
            case .restricted:
                statusPublisher
                    .send(completion: .failure(.restricted))
            case .denied:
                statusPublisher
                    .send(completion: .failure(.denied))
            case .notDetermined, .authorizedAlways,
                 .authorizedWhenInUse:
                statusPublisher.send(manager.
                    authorizationStatus)
            @unknown default:
                statusPublisher
                    .send(completion: .failure(.unknown))
        }
    }

func locationManager(_ manager: CLLocationManager,
                     didUpdateLocations
                     locations: [CLLocation]) {
    guard let location = locations.last else {
        return }
    locationPublisher.send(location)
}
}
```

4. Add the `LocationViewModel` class, starting from its public interface:

```
class LocationViewModel: ObservableObject {
    @Published
    private var status: CLAuthorizationStatus =
        .notDetermined
    @Published
    private var currentLocation: CLLocation?
    @Published
    var errorMessage = ""
    private let locationManager = LocationManager()
```

```
func startUpdating() {
    locationManager.start()
}
```

5. The public interface is not complete. Add a few more computed properties to fulfill all the UI's needs:

```
class LocationViewModel: ObservableObject {
    //...
    var thereIsAnError: Bool {
        !errorMessage.isEmpty
    }

    var latitude: String {
        currentLocation.latitudeDescription
    }

    var longitude: String {
        currentLocation.longitudeDescription
    }

    var statusDescription: String {
        switch status {
        case .notDetermined:
            return "notDetermined"
        case .authorizedWhenInUse:
            return "authorizedWhenInUse"
        case .authorizedAlways:
            return "authorizedAlways"
        case .restricted:
            return "restricted"
        case .denied:
            return "denied"
        @unknown default:
            return "unknown"
        }
    }
}
```

```
    }

    func startUpdating() {
        locationManager.start()
    }
    //...
}
```

6. To create a meaningful description for the optional `CLLocation` object, add a convenience extension to the `Optional` protocol definition:

```
extension Optional where Wrapped == CLLocation {
    var latitudeDescription: String {
        guard let self = self else {
            return "-"
        }
        return String(format: "%0.4f",
                      self.coordinate.latitude)
    }

    var longitudeDescription: String {
        guard let self = self else {
            return "-"
        }
        return String(format: "%0.4f",
                      self.coordinate.longitude)
    }
}
```

7. To finish this class, let's add the `subscribers` to the `init()` function:

```
class LocationViewModel: ObservableObject {
    //...

    private var cancellableSet: Set<AnyCancellable> = []

    init() {
        locationManager
            .statusPublisher
```

```
        .debounce(for: 0.5, scheduler: RunLoop.main)
        .removeDuplicates()
        .sink { completion in
            switch completion {
            case .finished:
                break
            case .failure(let error):
                self.errorMessage = error.rawValue
            }
        } receiveValue: { self.status = $0 }
        .store(in: &cancellableSet)

    locationManager.locationPublisher
        .debounce(for: 0.5, scheduler: RunLoop.main)
        .removeDuplicates(by: lessThanOneMeter)
        .assign(to: \.currentLocation, on: self)
        .store(in: &cancellableSet)
    }
}
```

8. As you can see, we removed all the location event updates where the distance is less than a meter. The following function removes the locations closest to the previous location:

```
class LocationViewModel: ObservableObject {
    //...
    private func lessThanOneMeter(_ lhs: CLLocation?,
                                 _ rhs: CLLocation?) -> Bool {
        if lhs == nil && rhs == nil {
            return true
        }
        guard let lhr = lhs,
              let rhr = rhs else {
            return false
        }
    }
}
```

```
        return lhr.distance(from: rhr) < 1
    }
}
```

9. After finishing the reactive *ViewModel*, let's complete the app with the *View*. First, create *ContentView* with the model and a `.task()` modifier to trigger the location update:

```
struct ContentView: View {
    @StateObject
    var locationViewModel = LocationViewModel()

    var body: some View {
        Group {
            .padding(.horizontal, 24)
            .task {
                locationViewModel.startUpdating()
            }
        }
    }
}
```

10. Inside the *Group* block, add the components for the error message or the actual coordinates:

```
Group {
    if locationViewModel.thereIsAnError {
        Text("Location Service terminated with error:
              \((locationViewModel.errorMessage))")
    } else {
        Text("Status:
              \((locationViewModel.statusDescription))")
        HStack {
            Text("Latitude: \((locationViewModel.latitude))")
            Text("Longitude:
                  \((locationViewModel.longitude))")
        }
    }
}
```

Upon running the app, after allowing the location service, you will see your current location:

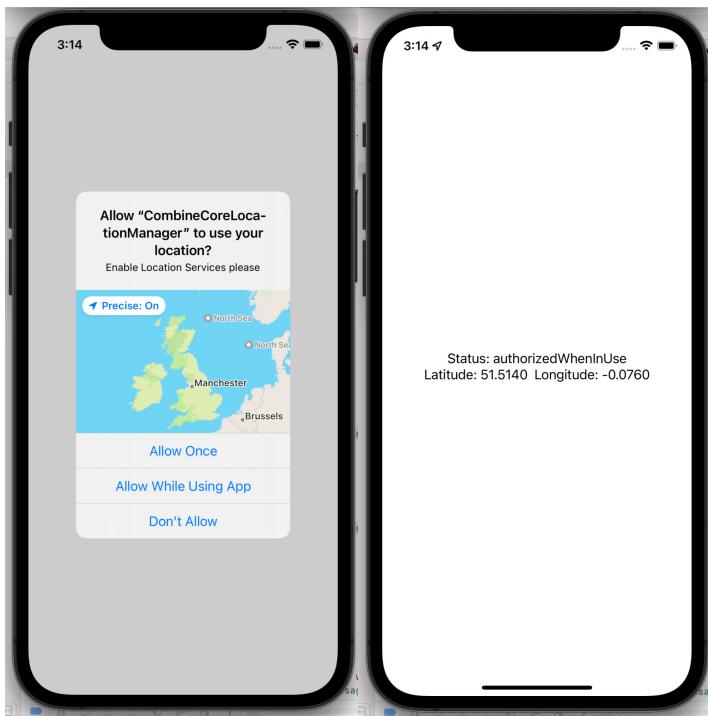


Figure 10.4 – A reactive location manager

Congratulations – you've implemented your first Combine app!

How it works...

Something to notice before diving into the *Publishers* description is that we set two `@Published` variables to `private`. This must seem counterintuitive because usually, the `@Published` variables are the variables that the View uses to render something. But in this case, the *ViewModel* exposes other variables – `latitude`, `longitude`, and `statusDescription` – that are derived from the `@Published` variables, and the View uses them to render their components.

By now, it should be clear that the goal of the `@Published` variables is to trigger a re-render, even though the View won't directly use the variables that are decorated with `@Published`.

Now, let's have a quick look at the two most important parts of this recipe: *Publishers* and *Subscriptions*.

Publishers in Combine

In our example, the *Publishers* are two *Subjects*. In the Combine world, a Publisher can be considered as a function, with one *Input* and one *Output*.

`Publisher` is the protocol that manages the *Output* that a client can subscribe to, whereas the protocol that manages the *Input* is the `Subject`, which provides a `send()` function to emit events.

Two *Subjects* are already implemented: `CurrentValueSubject` and `PassthroughSubject`.

The former has an initial value and maintains the changed value, even if it doesn't have any subscribers. The latter doesn't have an initial value and drops changes if nobody is observing it.

To send an event, the client has to call the `send()` function, as we do for changes in `status` and `location`:

```
statusPublisher.send(status)
//...
locationPublisher.send(location)
```

In Combine, an error scenario is a *completion case*. A *Publisher* sends an event until it completes, either successfully or with an error. After it completes, the stream is closed, and if we want to send more events, we need to create a new stream and the client must subscribe to the new one.

Back to the error case, the `send(completion:_)` function has either `.finished` or `.failure` as its parameter:

```
statusPublisher.send(completion: .failure(.restricted))
//...
statusPublisher.send(completion: .failure(.denied))
//...
statusPublisher.send(completion: .failure(.unknown))
```

In a nutshell, we just saw the publishing capabilities of Combine.

Subscriptions in Combine

The subscription part of Combine is more complex and sophisticated. A full discussion of subscriptions is beyond the scope of this book, but let's what we have done in our recipe.

The whole subscription part is in the `init()` function of the `LocationViewModel` class, where we subscribe to our two subscriptions.

The first thing to notice is that we connect different functions to create a combination of effects, in the same way that we are modifying the views in SwiftUI.

Let's dissect the `locationPublisher` subscription:

```
locationManager.locationPublisher
    .debounce(for: 0.5, scheduler: RunLoop.main)
    .removeDuplicates(by: lessThanOneMeter)
    .assign(to: \.currentLocation, on: self)
    .store(in: &cancellableSet)
```

First, we subscribe with the `debounce()` function, which discards all the events that happen too quickly and allows the changes to be passed every 0.5 seconds. It also moves the computation in `mainThread`: the event can be pushed by a function in a background thread, but the UI must always be updated in `mainThread`.

The `removeDuplicates()` function receives a subscription and modifies it, removing all the duplicates according to the predicate we pass, which, in our case, considers duplicating all the locations with a distance of less than 1 meter.

The `assign()` function is the final modifier that puts the event in a property of an object – in this case, the `currentLocation` property in the `LocationViewModel` class.

The last step, `.store(in:)`, puts the subscription in a set that has the same life cycle as the container, and it will be *dealloc-ed* when the container is *dealloc-ed* (we'll learn more about Combine memory management in the *Managing the memory in Combine to build a timer app* recipe).

Let's have a look at the subscriber to the `statusPublisher` publisher:

```
locationManager
    .statusPublisher
    .debounce(for: 0.5, scheduler: RunLoop.main)
    .removeDuplicates()
    .sink { completion in
        switch completion {
        case .finished:
            break
        case .failure(let error):
```

```
    self.errorMessage = error.rawValue
}
} receiveValue: { self.status = $0 }
.store(in: &cancellableSet)
```

We already know that the `debounce()` function and `removeDuplicates()` don't have parameters because the event type is an `equatable` enum.

The `sink()` function is a final and extended step, similar to the `assign()` function; it receives the values but also the completion so that it can apply different logic, depending on the received value.

This concludes our Combine primer. If you are feeling confused, don't worry: reactive programming is a big change of mindset. But after trying it in a few sample apps, you should start to see that when using it properly, programming a UI is even simpler than doing it in the usual imperative way.

See also

You can find more information regarding ReactiveX at <https://reactivex.io/>.

To gain a visual understanding of the Rx operators, and similarly for the Combine operators, the *RxMarbles* website (<https://rxmarbles.com/>) has a nice collection of pictures explaining them.

Managing the memory in Combine to build a timer app

When a client subscribes to a publisher, the result should be held somewhere. Usually, it is stored in a `Set` of `AnyCancellable`. If the client subscribes to multiple publishers, the code is a bit repetitive: it would be better to have a way to wrap all subscriptions and put all the results in the same set.

In this recipe, we'll use a feature that was experimentally introduced in Swift 5.1 that was then promoted as part of the language in Swift 5.4, the **result builders**, to create an extension of `Set` of `AnyCancellable`, wrap all the subscriptions, and store them in the set.

In this recipe, we are going to create a `StopWatch` app using three publishers: one for the deciseconds, one for the seconds, and one for the minutes.

We will also learn how to use timers in Combine.

Getting ready

Let's create a SwiftUI app called `StopWatch`.

How to do it...

The goal of the app is to have a stopwatch that can be started and stopped using a button. We are going to implement the counter using three timed publishers that emit events at different intervals: one every decisecond, one every second, and one every minute.

We are going to wrap them in a single closure to store them in the usual `Set` of `AnyCancellable`. Let's get started:

1. We will start by importing `Combine` and creating our reactive timer:

```
class StopWatchTimer: ObservableObject {  
    @Published  
    var deciseconds: Int = 0  
    @Published  
    var seconds: Int = 0  
    @Published  
    var minutes: Int = 0  
    @Published  
    var started = false  
  
    private var cancellableSet: Set<AnyCancellable> = []  
    func start() {  
    }  
    func stop() {  
    }  
}
```

2. Before implementing the `start()` function, let's quickly create the `stop()` function:

```
func stop() {  
    cancellableSet = []  
    started = false  
}
```

3. Add the `start()` function, where we will reset the counters and create the publishers and subscribe to them:

```
func start() {
    deciseconds = 0
    seconds = 0
    minutes = 0

    cancellableSet.store {
        Timer.publish(every: 0.1, on: RunLoop.main,
                      in: .default)
            .autoconnect()
            .sink { _ in
                self.deciseconds = (self.deciseconds +
                    1)%10
            }
        Timer.publish(every: 1.0, on: RunLoop.main,
                      in: .default)
            .autoconnect()
            .sink { _ in
                self.seconds = (self.seconds + 1)%60
            }
        Timer.publish(every: 60.0, on: RunLoop.main,
                      in: .default)
            .autoconnect()
            .sink { _ in
                self.minutes = (self.minutes + 1)%60
            }
    }
    started = true
}
```

4. The code is terse and looks good, but it doesn't compile because the `.store()` function is undefined. Let's define it:

```
typealias CancellableSet = Set<AnyCancellable>
extension CancellableSet {
    mutating func store(@CancellableBuilder _
```

```
        cancellables: () ->
[AnyCancellable]) {
    formUnion(cancellables())
}

@resultBuilder
struct CancellableBuilder {
    static func buildBlock(_ cancellables:
AnyCancellable...) -> [AnyCancellable] {
        return cancellables
    }
}

}
```

5. Now that we are finished with the timer, let's move on to `View`. Add the following code to render the digits:

```
struct ContentView: View {
    @StateObject
    private var timer = StopWatchTimer()

    var body: some View {
        VStack(spacing: 12) {
            HStack(spacing: 0) {
                Text("\(timer.minutes.formatted)")
                    .font(.system(size: 80))
                    .frame(width: 100)
                Text(":")
                    .font(.system(size: 80))
                Text("\(timer.seconds.formatted)")
                    .font(.system(size: 80))
                    .frame(width: 100)
                Text(":")
                    .font(.system(size: 80))
                Text("\(timer.deciseconds.formatted)")
                    .font(.system(size: 80))
            }
        }
    }
}
```

```
        .frame(width: 100)
    }
}
}
}
```

6. Then, add a `Button` to start/stop the timer:

```
var body: some View {
    VStack(spacing: 12) {
        //...
        Button {
            if timer.started {
                timer.stop()
            } else {
                timer.start()
            }
        } label: {
            Text(timer.started ? "Stop" : "Start")
                .foregroundColor(.white)
                .padding(.horizontal, 24)
                .padding(.vertical, 16)
                .frame(width: 100)
                .background(timer.started ?
                    Color.red : Color.green)
                .cornerRadius(5)
        }
    }
}
```

7. The only bit that's missing is an `extension` to `Int` to format it properly. Add it with the following code:

```
extension Int {
    var formatted: String {
        String(format: "%02d", self)
    }
}
```

Now, we can run the app and see that it measures the time in a nice and precise way:

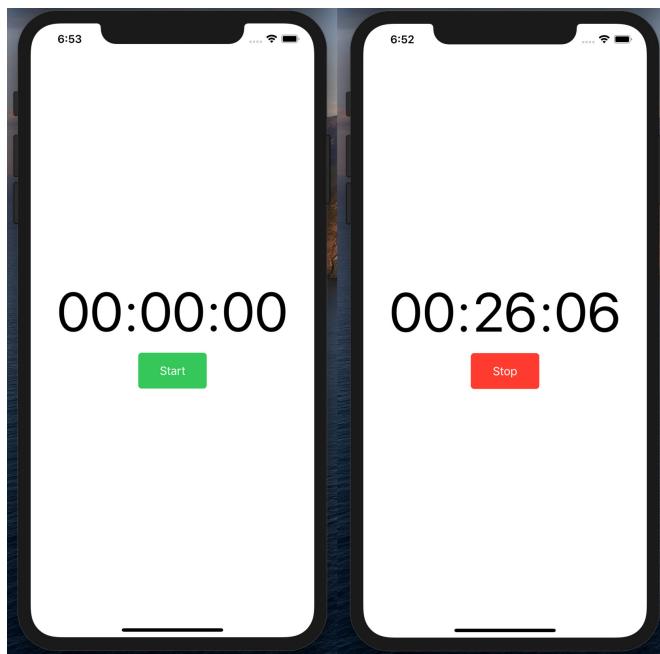


Figure 10.5 – A reactive StopWatch application in action

How it works...

In this recipe, we introduced a couple of new things: a timer publisher and a way to collect all the cancellable chain of subscriptions in one go. Let's remember that a chain of subscriptions means subscribers that connect to a publisher, possibly with a series of modifiers in between. A chain of subscriptions returns a token of the `AnyCancellable` type, which can be used to cancel that chain. If that cancellable token is collected in a `Set`, then when this `Set` is cleared, all the tokens are canceled.

Combine adds a publisher function to the `Timer` foundation class, where you can set the interval and the thread where it will emit the events. The `.autoconnect()` modifier will make the publisher start immediately.

Regarding the `.store()` function extension, it uses the `CancellableBuilder` result builder, which, in practice, produces a closure where all the values created inside it are returned as elements of an array. So, for example, we may have to define a builder from `Int`, such as the following:

```
@resultBuilder  
struct IntBuilder {
```

```
static func buildBlock(_ values: Int...) -> [Int] {  
    values.map { 2*$0 }  
}  
}
```

We would then give it a few functions returning an `Int`:

```
func functionReturningOne() -> Int { 1 }  
func functionReturningTwo() -> Int { 2 }  
func functionReturningThree() -> Int { 3 }
```

We can define a function that prints the double of the `Int` property that is passed as a parameter:

```
func printDoubleInt(@IntBuilder builder: () -> [Int]) {  
    print(builder())  
}
```

We can run the app with the following code:

```
printDoubleInt {  
    functionReturningOne()  
    functionReturningTwo()  
    functionReturningThree()  
    4  
    5  
}
```

We should see the following result:

```
[2, 4, 6, 8, 10]
```

Given this example, it is pretty clear that in our extension, the builder takes the array of `AnyCancellable` and adds its content to the original set.

See also

This pattern was proposed by Alexey Naumov and you can find the original blog post, *Variadic DisposeBag for Combine Subscriptions*, here: <https://nalexn.github.io/cancelbag-for-combine/>.

Validating a form using Combine

Sometimes, the reactive way of thinking feels academic and not connected to the usual problems a developer has to solve. In reality, most of the programs we usually write would benefit from using reactive programming, but some problems fit better than others.

As an example, let's consider a part of an app where the user has to fill in a form. The form has several fields, each one with a different way of being validated; some can be validated on their own while others have validation that depends on different fields, and all together concur to validate the whole form.

Imperatively, this usually creates a mess of spaghetti code, but by switching to the reactive declarative way, the code becomes natural.

In this recipe, we'll implement a simple signup page with a username text field and two password fields, one for the password and the other for password confirmation.

The username has a minimum number of characters, and the password must be at least eight characters long, comprising mixed numbers and letters, with at least one uppercase letter and a special character such as !, #, \$, and so on. Also, the password and the confirmation password must match. When all the fields are valid, the form will be valid, and we can proceed to the next page.

Each field will be a publisher, and the validations will be subscribers of those publishers.

By the end of this recipe, you'll be able to model each form validation using a reactive architecture and move a step further into understanding this way of building programs.

Getting ready

Create a SwiftUI app with Xcode called `FormValidation`.

How to do it...

To explore the validation pattern for a form, we are going to implement a simple signup page.

As usual, when dealing with the UI and a reactive framework, we will separate the View from the business logic using MVVM.

In our case, the business logic is the validation logic, and it will be encapsulated in a class called `SignupViewModel`:

1. Let's start by defining the public interface of `SignupViewModel`, indicating the `@Published` properties for the input and output:

```
class SignupViewModel: ObservableObject {
    // Input
    @Published
    var username = ""

    @Published
    var password = ""

    @Published
    var confirmPassword = ""

    // Input
    @Published
    var isValid = false

    @Published
    var usernameMessage = " "

    @Published
    var passwordMessage = " "

    private var cancellableSet: Set<AnyCancellable> = []
}
```

2. The whole validation logic will be in the `init()` function, separated into three streams: one for the `username`, one for the `password`, and one for the `form`.

Let's start by adding the one for the `username`:

```
init() {
    usernameValidPublisher
        .receive(on: RunLoop.main)
        .map { $0 ? " "
            : "Username must be at least 6 characters
long" }
        .assign(to: \.usernameMessage, on: self)
        .store(in: &cancellableSet)
}
```

3. Similarly, add the validation for the password, noting that the *Publisher* emits an enum and not a simple `Bool`:

```
init() {
    //...
    passwordValidPublisher
        .receive(on: RunLoop.main)
        .map { passwordCheck in
            switch passwordCheck {
            case .invalidLength:
                return "Password must be at least
                    8 characters long"
            case .noMatch:
                return "Passwords don't match"
            case .weakPassword:
                return "Password is too weak"
            default:
                return " "
            }
        }
        .assign(to: \.passwordMessage, on: self)
        .store(in: &cancellableSet)
}
```

4. Finally, add the *Publisher* form validation subscription:

```
init() {
    //...
    formValidPublisher
        .receive(on: RunLoop.main)
        .assign(to: \.isValid, on: self)
        .store(in: &cancellableSet)
}
```

5. For convenience, add the *Publishers* as computed properties in a private extension of `SignupViewModel`. The first one is the publisher for the username, whose only logic is checking if the length of `username` is correct:

```
private extension SignupViewModel {  
    var usernameValidPublisher: AnyPublisher<Bool,  
        Never> {  
        $username  
            .debounce(for: 0.5, scheduler:  
                RunLoop.main)  
            .removeDuplicates()  
            .map { $0.count >= 5 }  
            .eraseToAnyPublisher()  
    }  
}
```

6. The password validation is a bit more complex since it needs to check three properties: the valid length, the strength of the password, and if the password and the confirmed password match.

Let's start with validating the password's length:

```
private extension SignupViewModel {  
    //...  
    var validPasswordLengthPublisher:  
        AnyPublisher<Bool, Never> {  
        $password  
            .debounce(for: 0.5, scheduler:  
                RunLoop.main)  
            .removeDuplicates()  
            .map { $0.count >= 8 }  
            .eraseToAnyPublisher()  
    }  
}
```

7. Then, add the check for the strength:

```
private extension SignupViewModel {  
    //...  
    var strongPasswordPublisher: AnyPublisher<Bool,  
        Never> {  
        $password  
            .debounce(for: 0.2, scheduler:  
                RunLoop.main)  
            .removeDuplicates()  
            .map(\.isStrong)  
            .eraseToAnyPublisher()  
    }  
}
```

8. Note that here, we are checking if a string is strong by using an `isStrong` function that doesn't exist yet. Let's add it as an extension of `String`. It basically checks if the string contains letters, digits, uppercase letters, and special characters such as £, \$, !, and so on:

```
extension String {  
    var isStrong: Bool {  
        containsACharacter(from: .lowercaseLetters) &&  
        containsACharacter(from: .uppercaseLetters) &&  
        containsACharacter(from: .decimalDigits) &&  
        containsACharacter(from:  
            .alphanumerics.inverted)  
    }  
  
    private func containsACharacter(from set:  
        CharacterSet) -> Bool {  
        rangeOfCharacter(from: set) != nil  
    }  
}
```

9. While we are here, let's also add the enum property that was returned by the password validations:

```
enum PasswordCheck {
    case valid
    case invalidLength
    case noMatch
    case weakPassword
}
```

10. Let's go back to the password validation stream, adding the one that checks whether the password and confirmed passwords match:

```
private extension SignupViewModel {
    //...
    var matchingPasswordsPublisher: AnyPublisher<Bool,
        Never> {
        Publishers
            .CombineLatest($password,
            $confirmPassword)
            .debounce(for: 0.2, scheduler:
            RunLoop.main)
            .map { password, confirmPassword in
                password == confirmPassword
            }
            .eraseToAnyPublisher()
    }
}
```

11. The final password validation stream combines all the previous validators. We can do this with the following code:

```
private extension SignupViewModel {
    //...
    var passwordValidPublisher:
        AnyPublisher<PasswordCheck,
        Never> {
        Publishers
            .CombineLatest3(validPasswordLengthPublisher,
```

```
        strongPasswordPublisher,
        matchingPasswordsPublisher)
    .map { validLength, strong, matching in
        if (!validLength) {
            return .invalidLength
        }
        if (!strong) {
            return .weakPassword
        }
        if (!matching) {
            return .noMatch
        }
        return .valid
    }
    .eraseToAnyPublisher()
}
}
```

12. The last validator we will implement is the *form* validator, which combines the *username* and *password* validators:

```
private extension SignupViewModel {
    //...
    var formValidPublisher: AnyPublisher<Bool, Never> {
        Publishers
            .CombineLatest(usernameValidPublisher,
                           passwordValidPublisher)
            .map { usernameIsValid, passwordIsValid in
                usernameIsValid && (passwordIsValid ==
                    .valid)
            }
            .eraseToAnyPublisher()
    }
}
```

13. Before we render the Views in `ContentView`, we will add a bunch of convenience modifiers for `TextField` and `SecureField`:

```
struct CustomStyle: ViewModifier {
    func body(content: Content) -> some View {
        content
            .frame(height: 40)
            .background(Color.white)
            .cornerRadius(5)
    }
}

extension TextField {
    func custom() -> some View {
        modifier(CustomStyle())
        .autocapitalization(.none)
    }
}

extension SecureField {
    func custom() -> some View {
        modifier(CustomStyle())
    }
}
```

14. `ContentView` is a simple vertical stack of `TextField` and `Text` components that represent the fields to add and the possible error messages. We will modify `ContentView` so that it has a yellow background and a `VStack` containing the fields:

```
struct ContentView: View {
    @ObservedObject
    private var signupViewModel = SignupViewModel()

    var body: some View {
        ZStack {
            Color.yellow.opacity(0.2)
            VStack(spacing: 24) {
```

```
//...
}
.padding(.horizontal, 24)
}
.edgesIgnoringSafeArea(.all)
}
}
```

15. Then, we will add the field for the username:

```
VStack(spacing: 24) {
    VStack(alignment: .leading) {
        Text(signupViewModel.usernameMessage)
            .foregroundColor(.red)
        TextField("Username", text:
            $signupViewModel.username)
            .custom()
    }
}
```

16. Now, let's move on to the password, where we will add two Textfield parameters, one for the password and another to confirm it. This will help the user ensure they have inserted it without misspelling it:

```
VStack(spacing: 24) {
//...
    VStack(alignment: .leading) {
        Text(signupViewModel.passwordMessage)
            .foregroundColor(.red)
        SecureField("Password",
            text: $signupViewModel.password)
            .custom()
        SecureField("Repeat Password",
            text: $signupViewModel.confirmPassword)
            .custom()
    }
}
```

17. Finally, let's add the Register button, which will only be enabled when the form is fully valid:

```
 VStack(spacing: 24) {  
     //...  
     Button {  
         print("Successfully registered!")  
     } label: {  
         Text("Register")  
             .foregroundColor(.white)  
             .frame(width: 100, height: 44)  
             .background(signupViewModel.isValid ?  
                     Color.green : Color.red)  
             .cornerRadius(10)  
     }.disabled(!signupViewModel.isValid)  
 }
```

I admit that this looks like a long recipe, but you can observe how nicely the logic flows. Now, it's time to run our app:

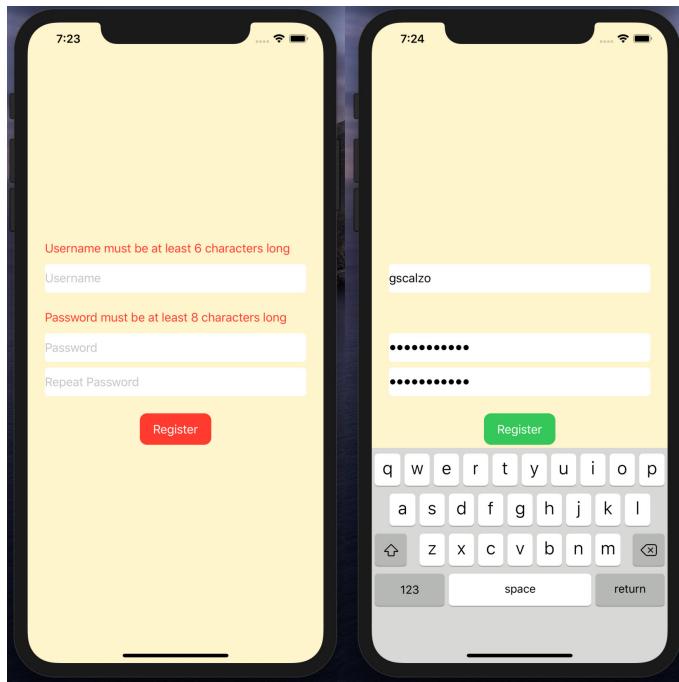


Figure 10.6 – A reactive signup page

How it works...

This is a kind of problem where Combine shines: splitting the flow of the logic into simple sub-steps and then combining them.

There are a few things to notice here:

- At the end of each publisher, there is the `eraseToAnyPublisher()` modifier. The reason for this is that the composition of the modifiers on the publishers creates some complex nested types, wherein the end the subscriber only needs an object of the `Publisher` type. The function does some `eraseToAnyPublisher()` magic to flatten these types and just returns a generic `Publisher`.
- The `.debounce()` function only sends events if they happen in the interval and are passed as parameters; that way, the subscriber is not bombarded by events. In this case, if the user is a fast typist, we are only evaluating the password or username after a few characters have been inserted.
- The `.map(\.isStrong)` function is a nice shortcut that was added in Swift 5.2 and it is equivalent to this:

```
.map { password in
    return password.isStrong
}
```

- Finally, `Publishers.CombineLatest` and `Publishers.CombineLatest3` create a new publisher that emits a single tuple, or triple, with the latest events of each publisher passed as parameters.

There's more...

Now that have you seen how easy and obvious it is to create validators, why not improve our simple signup page a little?

For example, you may have noticed that at the beginning, both the error messages are present when you should probably be concentrating on adding a valid username.

The improvement that I suggest you should make is change the validators in a manner that the password is validated only after the username is correct.

Do you think you'll be able to do it?

Fetching remote data using Combine and visualizing it in SwiftUI

A common characteristic that most mobile apps have is that they fetch data from a remote web service. Given the asynchronous nature of the problem, it is often problematic when this is implemented in the normal imperative world. However, it suits the reactive world nicely, as we'll see in this recipe.

We are going to implement a simple weather app, fetching the current weather and a 5-day forecast from **OpenWeather**, a famous service that also has a free tier.

After fetching the forecast, we will present the results in a list view, with the current weather fixed on the top.

Getting ready

We will start by creating a SwiftUI app called `Weather`.

To use this service, we must create an account on **OpenWeather**:

- Go to the **OpenWeather** signup page (https://home.openweathermap.org/users/sign_up) and fill in your data:

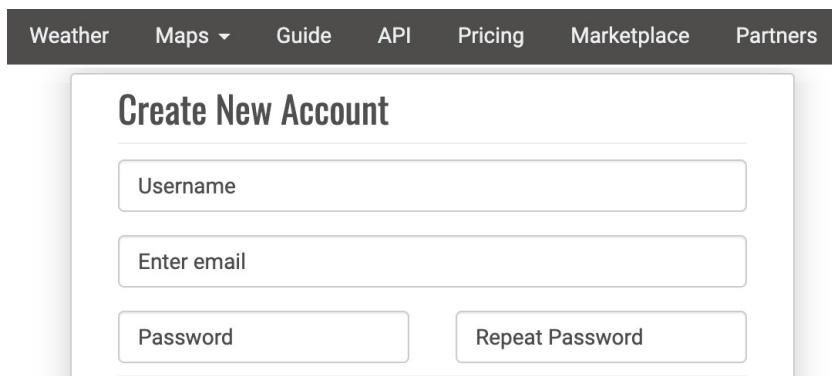


Figure 10.7 – OpenWeather signup page

- After confirming the login, you must create a new API key from the **API keys** page (https://home.openweathermap.org/api_keys):

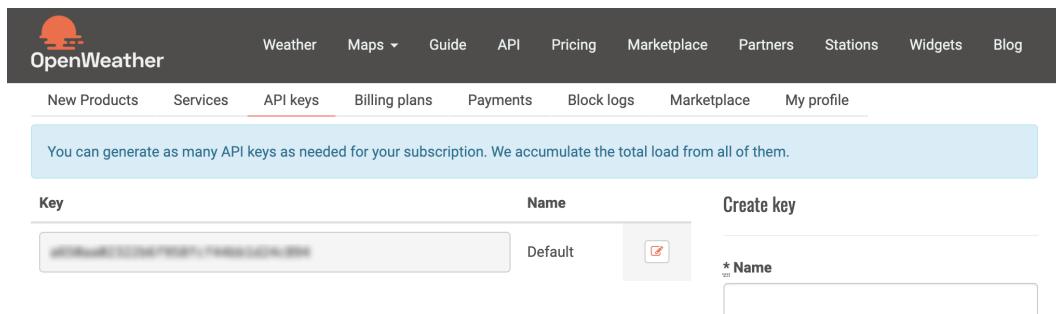


Figure 10.8 – OpenWeather API keys

- Note that you have to validate your email address and that the activation could take a couple of hours. Check your spam folder and be patient.

We'll use the key we created when calling the endpoints to fetch the weather and the forecast.

How to do it...

We are going to separate the implementation into two parts: the web service and the UI.

Because **OpenWeather** is exposing two different endpoints for the current weather and the forecast, the web service will use two streams.

The View will observe the two variables – the current weather and the forecast – and it will update accordingly:

- Before we implement the objects, let's decide on the model we want to have and how to create it from the JSON response we get from the service. The model to create for the weather is simple:

```
struct Weather: Decodable, Identifiable {
    var id: TimeInterval { time.timeIntervalSince1970 }
    let time: Date
    let summary: String
    let icon: String
    let temperature: Double
}
```

2. The following is an example that's returned from the service:

```
{  
    //...  
    "weather": [  
        {  
            "id": 803,  
            "main": "Clouds",  
            "description": "broken clouds",  
            "icon": "04d"  
        }  
    ],  
    "main": {  
        "temp": 19.24,  
        "feels_like": 19.23,  
        "temp_min": 16.82,  
        "temp_max": 21.65,  
        "pressure": 999,  
        "humidity": 77  
    },  
    "dt": 1628265544,  
    //...  
}
```

The highlighted lines are the attributes we implemented in our model. We have to extract them in the `init()` function.

Define the keys and the `init()` function from the decoder so that the model can be decoded from the JSON response:

```
struct Weather: Decodable, Identifiable {  
    //...  
    enum CodingKeys: String, CodingKey {  
        case time = "dt"  
        case weather = "weather"  
        case summary = "description"  
        case main = "main"  
        case icon = "icon"  
        case temperature = "temp"
```

```
    }

    init(from decoder: Decoder) throws {
        let container = try decoder.container(keyedBy:
            CodingKeys.self)
        time = try container.decode(Date.self, forKey:
            .time)
        var weatherContainer = try container
            .nestedUnkeyedContainer(forKey: .weather)
        let weather = try weatherContainer
            .nestedContainer(keyedBy: CodingKeys.self)
        summary = try weather
            .decode(String.self, forKey: .summary)
        icon = try weather.decode(String.self,
            forKey: .icon)
        let main = try container
            .nestedContainer(keyedBy: CodingKeys.self,
                forKey: .main)
        temperature = try main.decode(Double.self,
            forKey: .temperature)
    }
}
```

3. Create the forecast struct as a simple list of the Weather records:

```
struct ForecastWeather: Decodable {
    let list: [Weather]
}
```

4. Now, let's move on to ObservableObject of the WeatherService class, whose goal is to connect to the service and fetch the data. We are exposing three variables: the current weather, the forecast, and a message if there is an error. Replace the <INSERT YOUR KEY> string (highlighted in the following code) with the API key we created in the *Getting ready* section. The API key should be added in double quotes:

```
class WeatherService: ObservableObject {
    @Published var errorMessage: String = ""
```

```
@Published var current: Weather?  
@Published var forecast: [Weather] = []  
  
private let apiKey = <INSERT YOUR KEY>  
private var cancellableSet: Set<AnyCancellable> = []  
}
```

5. Given that we have two endpoints, we are going to use them in a `load()` function, starting with the current weather. Add the following function to the `WeatherService` class:

```
func load(latitude: Float, longitude: Float) {  
    let decoder = JSONDecoder()  
    decoder.dateDecodingStrategy = .secondsSince1970  
  
    let currentURL = URL(string:  
        "https://api.openweathermap.org/data/2.5/weather?  
        lat=\(latitude)&lon=\(longitude)&  
        appid=\(apiKey)&units=metric")!  
    URLSession  
        .shared  
        .dataTaskPublisher(for: URLRequest(url:  
            currentURL))  
        .map(\.data)  
        .decode(type: Weather.self, decoder: decoder)  
        .receive(on: RunLoop.main)  
        .sink { completion in  
            switch completion {  
            case .finished:  
                break  
            case .failure(let error):  
                self.errorMessage =  
                    error.localizedDescription  
            }  
        }  
        .receiveValue: {  
            self.current = $0  
        }  
}
```

```
        .store(in: &cancellableSet)
    }
```

6. In the same function, add the call to the forecast endpoint:

```
func load(latitude: Float, longitude: Float) {
    //...
    let forecastURL = URL(string:
        "https://api.openweathermap.org/data/2.5/forecast?
        lat=\(latitude)&lon=\(longitude)&
        appid=\(apiKey)&units=metric")!
    URLSession
        .shared
        .dataTaskPublisher(for: URLRequest(url:
            forecastURL))
        .map(\.data)
        .decode(type: ForecastWeather.self, decoder:
            decoder)
        .receive(on: RunLoop.main)
        .sink { completion in
            switch completion {
            case .finished:
                break
            case .failure(let error):
                self.errorMessage =
                    error.localizedDescription
            }
        } receiveValue: {
            self.forecast = $0.list
        }
        .store(in: &cancellableSet)
}
```

7. Before implementing the views, let's add a few extensions to format some values of the weather:

```
extension Double {  
    var formatted: String {  
        String(format: "%.0f", self)  
    }  
}
```

8. If you go to **OpenWeather** and look at the **Weather icons** page (<https://openweathermap.org/weather-conditions>), you will find the relationship between the icon code and a proper image. Using that page, let's create a map to translate the code into an equivalent **SF Symbols** icon, starting with the icon to use during the day:

```
extension String {  
    var weatherIcon: String {  
        switch self {  
            case "01d":  
                return "sun.max"  
            case "02d":  
                return "cloud.sun"  
            case "03d":  
                return "cloud"  
            case "04d":  
                return "cloud.fill"  
            case "09d":  
                return "cloud.rain"  
            case "10d":  
                return "cloud.sun.rain"  
            case "11d":  
                return "cloud.bolt"  
            case "13d":  
                return "cloud.snow"  
            case "50d":  
                return "cloud.fog"  
        }  
    }  
}
```

```
    }
}
```

9. Now, let's add the icons for the forecast during the night:

```
extension String {
    var weatherIcon: String {
        //...
        case "01n":
            return "moon"
        case "02n":
            return "cloud.moon"
        case "03n":
            return "cloud"
        case "04n":
            return "cloud.fill"
        case "09n":
            return "cloud.rain"
        case "10n":
            return "cloud.moon.rain"
        case "11n":
            return "cloud.bolt"
        case "13n":
            return "cloud.snow"
        case "50n":
            return "cloud.fog"
        default:
            return "icloud.slash"
    }
}
```

10. Create a `CurrentWeather`: View component to present the weather that's been fetched by the service:

```
struct CurrentWeather: View {
    let current: Weather
```

```
var body: some View {
    VStack(spacing: 28) {
        Text(current.time
            .formatted(date: .long, time:
                .standard))
        HStack {
            Image(systemName:
                current.icon.weatherIcon)
                .font(.system(size: 98))
            Text("\(current.temperature.formatted) °")
                .font(.system(size: 46))
        }
        Text("\(current.summary)")
    }
}
```

11. Each hourly forecast is presented in a `WeatherRow: View` class with the weather icon, the time, and the temperature. Add the following code:

```
struct WeatherRow: View {
    let weather: Weather

    var body: some View {
        HStack() {
            Image(systemName:
                weather.icon.weatherIcon)
                .frame(width: 40)
                .font(.system(size: 28))
            VStack(alignment: .leading) {
                Text(weather.summary)
                Text(weather.time.formatted(date: .long,
                    time: .standard))
                    .font(.system(.footnote))
            }
            Spacer()
            Text("\(weather.temperature.formatted) ° ")
        }
    }
}
```

```
        .frame(width: 40)
    }
    .padding(.horizontal, 16)
}
}
```

12. Let's move our attention to `ContentView`, where we will present the error message, which is empty by default, and then the current weather and the forecast as a vertical list.

Create `ContentView` with a `VStack` container and the error message:

```
struct ContentView: View {
    @StateObject
    var weatherService = WeatherService()

    var body: some View {
        VStack {
            Text(weatherService.errorMessage)
                .font(.largeTitle)
        }
    }
}
```

13. Add the current weather component and the forecast as a list to the body of `ContentView`:

```
var body: some View {
    VStack {
        //...
        if let currentWeather = weatherService.current {
            VStack {
                CurrentWeather(current:
                    currentWeather)
                List(weatherService.forecast) {
                    WeatherRow(weather: $0)
                }
            }.listStyle(.plain)
        }
    }
}
```

```
        }
    }
}
```

14. Finally, add the initial task to fetch the current weather and the hourly forecast:

```
var body: some View {
    VStack {
        //...
    }
    .task {
        weatherService.load(latitude: 51.5074,
                             longitude: 0.1278)
    }
}
```

This has been a long recipe, but now, you finally have your reward! Upon launching the app, you will see the precise forecast for 5 days, with a granularity of 3 hours:

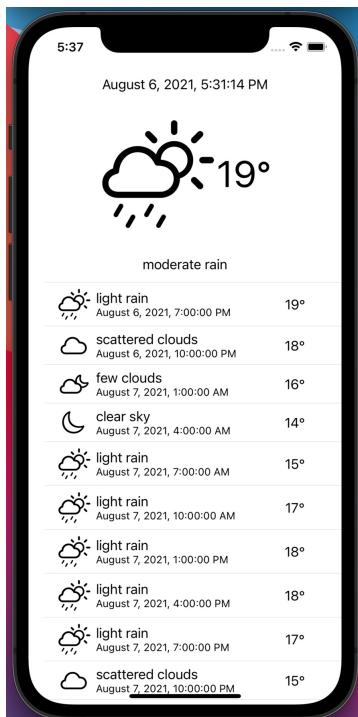


Figure 10.9 – WeatherApp with a 5-day forecast

How it works...

After a long session of coding, let's look at the fetching part in detail:

```
URLSession
    .shared
    .dataTaskPublisher(for: URLRequest(url: currentURL))
    .map(\.data)
    .decode(type: Weather.self, decoder: decoder)
    .receive(on: RunLoop.main)
    .sink { completion in
        switch completion {
        case .finished:
            break
        case .failure(let error):
            self.errorMessage = error.localizedDescription
        }
    } receiveValue: {
        self.current = $0
    }
    .store(in: &cancellableSet)
```

First of all, we can see that Combine adds a reactive function to the normal URLSession, sparing us from having to implement an adapter.

The .dataTaskPublisher() function returns a publisher where we extract the .data() field with .map(\.data).

Here, we can see the power of Swift, where we can pass the keypath to extract instead of doing a longer .map { \$0.data }; it's not just a matter of sparing a bunch of keystrokes, but the former expresses the intent of the code better.

The data blob is then processed by .decode(type: Weather.self, decoder: decoder), which decodes the Weather object.

After that, the publisher transmits a Weather object and .receive(on: RunLoop.main) moves the computation in MainThread, ready to be used for UI changes.

The final computational step, the .sink() function, extracts the error or the values and puts them in the correct @Published variable.

Finally, the publisher is put in the cancellable set, to be removed when `WeatherService` is disposed of.

As you can see, all the steps of the computation are natural and obvious.

Just as an exercise, try to implement the same logic using the conventional declarative way without Combine, and try to understand the differences.

Unrelated to Combine and SwiftUI is the decodable process, where the model doesn't match the structure of the JSON object.

To solve this, we can use the `.nestedContainer` function, which returns a keyed sub-container such as a dictionary, and the `.nestedUnkeyedContainer` function, which returns a sub-container such as an array.

The JSON we will receive is similar to the following:

```
{  
    "weather": [  
        {  
            "description": "broken clouds"  
        }  
    ],  
    "main": {  
        "temp": 19.24  
    },  
    "dt": 1628265544  
}
```

Using the following code, we can reach the two sub-containers: `weather` and `main`. From here, we can extract the necessary values:

```
var weatherContainer = try container  
    .nestedUnkeyedContainer(forKey: .weather)  
let weather = try weatherContainer  
    .nestedContainer(keyedBy: CodingKeys.self)  
summary = try weather.decode(String.self, forKey: .summary)  
icon = try weather.decode(String.self, forKey: .icon)  
  
let main = try container  
    .nestedContainer(keyedBy: CodingKeys.self,
```

```
        forKey: .main)
temperature = try main.decode(Double.self,
                                forKey: .temperature)
```

As I said, this is unrelated to Combine, but it is a good refresher on how Decodable works, so I thought it was worth mentioning it.

There's more...

I believe that what you've learned in this recipe will be one of the Combine concepts that you'll use more in your apps. So, it's worth trying to understand it.

As we mentioned in the previous section, a good exercise would be to implement it in the normal Foundation way using delegate functions and so on, to see the differences of this approach.

Given that we have two endpoints, one for the current weather and the other for the forecast, we implemented two asynchronous and separated streams. How about combining them in a third stream that finishes when both of them finish, and then creates a single object with both variables?

Finally, we hardcoded the coordinates, but how about using `CLLocationManager`, which we implemented in the *Introducing Combine in a SwiftUI project* recipe, to get the current location of the user?

Debugging an app based on Combine

It's a common idea that debugging reactive code is more difficult than debugging imperative code. Unfortunately, this is not completely wrong; partly because of the nature of the code and partly because the development tools are not sophisticated enough to follow this new paradigm.

Combine, however, implements a few convenient ways to help us understand what happens in our streams.

In this recipe, we'll learn about three techniques we can use to debug a Combine stream. I admit that all three are a bit basic; however, they are a starting point and should be enough to help us understand how to deal with errors in the streams.

Getting ready

Create a SwiftUI app called `DebuggingCombine` in Xcode.

How to do it...

Given the limited possibilities of debugging Combine, we will not be implementing a sophisticated app. Instead, we will be implementing a trivial three-button app that calls the three possible ways of debugging Combine:

- Handling events
- Printing events
- Conditional breakpoint

The Combine publishers we are going to test are plain transmitters that emit a string or an integer. Let's get started:

1. Start by implementing the simple object with three functions:

```
import Combine
class ReactiveObject {
    private var cancellableSet: Set<AnyCancellable> = []
    func handleEvents() {
    }
    func printDebug() {
    }
    func breakPoint() {
    }
}
```

2. ContentView just contains three buttons, one for each function, but before adding them, let's prepare ContentView:

```
struct ContentView: View {
    var reactiveObject = ReactiveObject()
    var body: some View {
        VStack(spacing: 24) {
        }
    }
}
```

3. Add the buttons to VStack:

```
VStack(spacing: 24) {
    Button {
        reactiveObject.handleEvents()
    } label: {
        Text("HandleEvents")
            .foregroundColor(.white)
            .frame(width: 200, height: 50)
            .background(Color.green)
    }
    Button {
        reactiveObject.printDebug()
    } label: {
        Text("Print")
            .foregroundColor(.white)
            .frame(width: 200, height: 50)
            .background(Color.orange)
    }

    Button {
        reactiveObject.breakPoint()
    } label: {
        Text("Breakpoint")
            .foregroundColor(.white)
            .frame(width: 200, height: 50)
            .background(Color.red)
    }
}
```

4. Move back to the ReactiveObject class and implement the first function, `handleEvents()`:

```
func handleEvents() {
    let subject = PassthroughSubject<String, Never>()
    subject
        .handleEvents(receiveSubscription: {
            print("Receive subscription: \$(\$0)")
        })
}
```

```
        }, receiveOutput: {
            print("Received output: \($0)")
        }, receiveCompletion: {
            print("Receive completion: \($0)")
        }, receiveCancel: {
            print("Receive cancel")
        }, receiveRequest: {
            print("Receive request: \($0)")
        })
        .sink { _ in }
    .store(in: &cancelableSet)
subject.send("New Message!")
}
```

5. The `printDebug()` function is even simpler:

```
func printDebug() {
    let subject = PassthroughSubject<String, Never>()
    subject
        .print("Print").sink { _ in }
    .store(in: &cancelableSet)
    subject.send("New Message!")
}
```

6. The final function, `breakPoint()`, adds a conditional breakpoint when the counter is at 7:

```
func breakPoint() {
    (1..<10).publisher
        .breakpoint(receiveOutput: {
            $0 == 7
        }) { $0 == .finished
    }
    .sink { _ in }
    .store(in: &cancelableSet)
}
```

You can run the app and see the effects of the different strategies. The following screenshot shows the debug using print events:

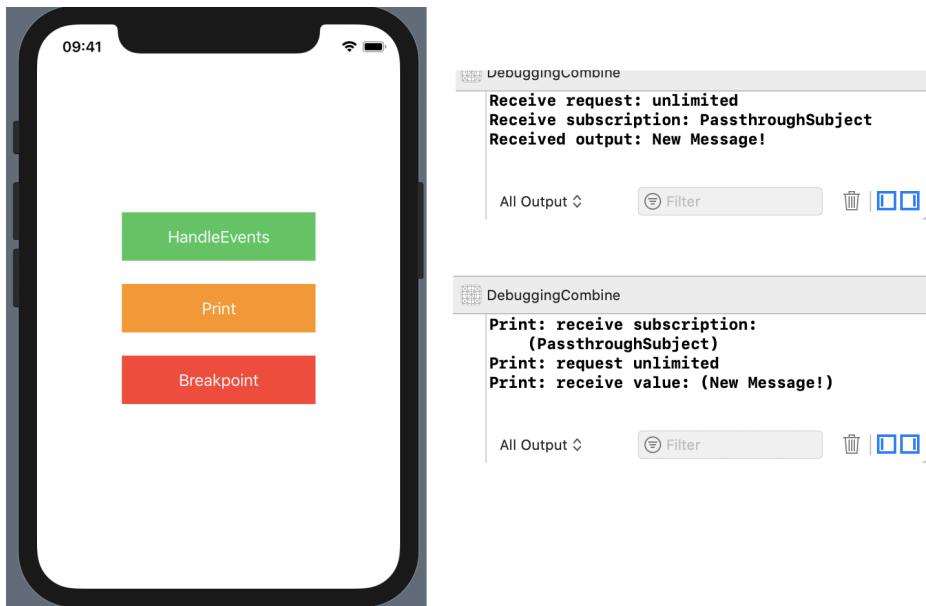


Figure 10.10 – Debugging Combine handling and printing events

In the following events, the Xcode debugger stopped when we used the breakpoint strategy:

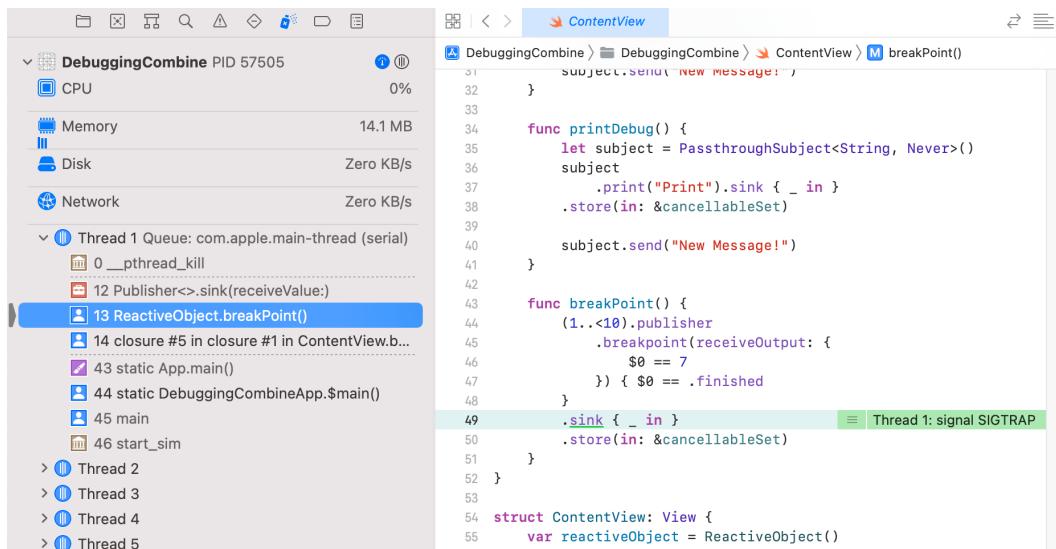


Figure 10.11 – Breakpoint in a Combine app

How it works...

Even if it is quite basic, the debug functionalities that Combine brings to us are quite useful.

The `.handleEvents()` function has a few closures whereby we can perform more sophisticated activities instead of simple printing, as in our example.

The `.print()` function is a shortcut for the previous `.handleEvents()` function implementation, where every event is printed in the Xcode console with the prefix we pass, which we can use to filter those messages in the Xcode console.

Finally, `.breakpoint()` stops the execution of the code when it is called.

There's more...

It's worth signaling a super-useful tool that was implemented by *Marin Todorov* called **Timelane**, which allows you to follow the streams and present the results in **Instruments**. It is a pretty advanced tool, but it is also sophisticated and flexible, and it's worth giving it a try.

It can be found at <http://timelane.tools/> and is illustrated here:

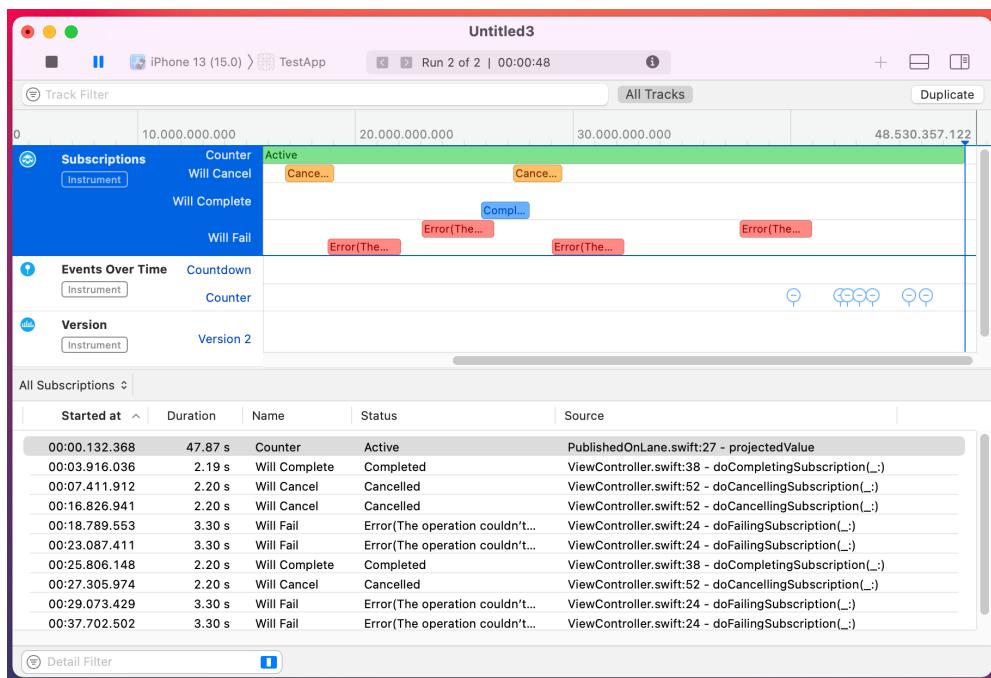


Figure 10.12 – Timelane in action

Unit testing an app based on Combine

I must confess that the topic of this recipe is very close to my heart: unit testing an app based on Combine.

We are going to implement an app that retrieves a list of GitHub users and shows them in a list view.

The code is pretty similar to the one in the *Fetching remote data using Combine and visualizing it in SwiftUI* recipe. But in this case, we'll learn how to unit test it – something that, even if it is very important, isn't well covered in documentation and tutorials.

Getting ready

Let's open Xcode and create a SwiftUI app called `GithubUsers`, paying attention to enabling the tests by checking the **Include Tests** check box:

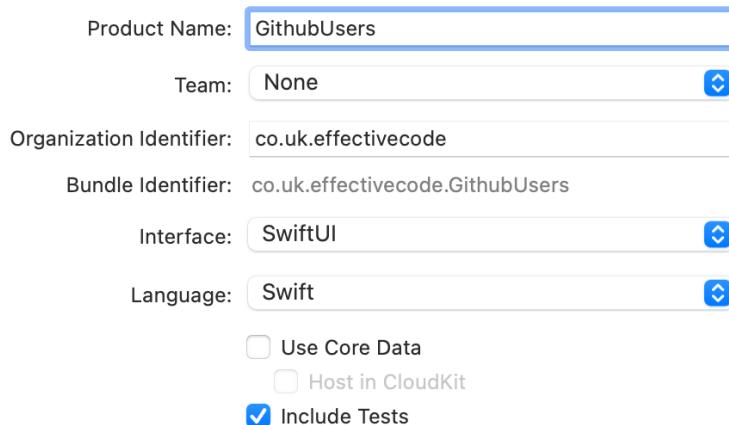


Figure 10.13 – Creating a SwiftUI app with tests enabled

Then, add the `githubUsers.json` file to the `GithubUsersTests` folder. You can find it in this book's GitHub repository at <https://github.com/PacktPublishing/SwiftUI-Cookbook-2nd-Edition-/blob/main/Resources/Chapter10/recipe6/githubUsers.json>:

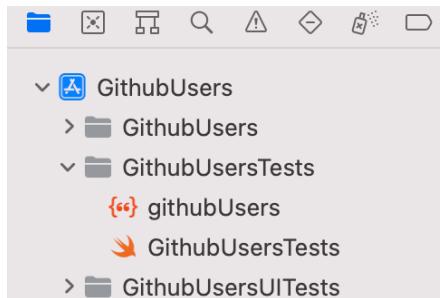


Figure 10.14 – Fixture for the tests

How to do it...

We will divide this recipe into two logic parts: the first part will implement the app retrieval and show the GitHub users, while the second part will add a unit test for fetching the code:

1. Start by importing Combine and creating a model for the user:

```
import SwiftUI
import Combine

struct GithubUser: Decodable, Identifiable {
    let id: Int
    let login: String
    let avatarUrl: String
}
```

2. Implement a `Github` class that fetches the user from `github.com`, creating a publisher from `URLSession`:

```
class Github: ObservableObject {
    @Published
    var users: [GithubUser] = []

    private var cancellableSet: Set = []
    func load() {
        let url = URL(string:
            "https://api.github.com/users")!
        let decoder = JSONDecoder()
        decoder.keyDecodingStrategy =
```

```
        .convertFromSnakeCase
    URLSession.shared
        .dataTaskPublisher(for: URLRequest(url: url))
            .map(\.data)
            .decode(type: [GithubUser].self, decoder:
                decoder)
            .replaceError(with: [])
            .receive(on: RunLoop.main)
            .assign(to: \.users, on: self)
            .store(in: &cancellableSet)
    }
}
```

3. The view will have the previous class as a `@StateObject` and will load the users when `View` is presented:

```
struct ContentView: View {
    @StateObject var github = Github()

    var body: some View {
        List(github.users) {
            GithubUserView(user: $0)
        }
        .task { github.load() }
    }
}
```

4. Finally, for each user, we will show their username and avatar image:

```
struct GithubUserView: View {
    let user: GithubUser

    var body: some View {
        HStack {
            AsyncImage(url: URL(string:
                user.avatarUrl)) { image in
                image
            .resizable()
```

```
        .scaledToFill()
    } placeholder: {
        Color.purple.opacity(0.1)
    }
    .frame(width: 40, height: 40)
    .cornerRadius(20)
    Spacer()
    Text(user.login)
}
}
```

The app runs correctly and presents the users after fetching them, as shown in the following screenshot:

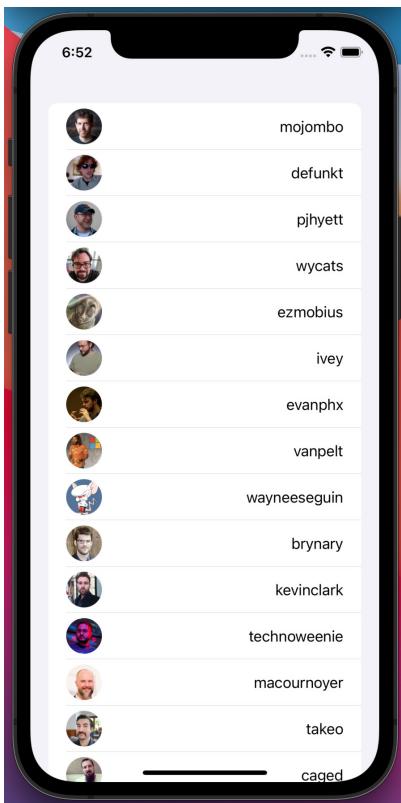


Figure 10.15 – Fetching and presenting GitHub users

How can you be sure that it will always work when we add more features and change the code? The answer is to unit test it!

In the next part of this recipe, we'll learn how to test a Combine publisher.

5. Let's move on to the `GithubUsersTests` file. Here, we will import `Combine` and add a simple function to `XCTestsCase` to read a JSON file that we'll use in our mock:

```
import XCTest
@testable import GithubUsers
import Combine
extension XCTestCase {
    func loadFixture(named name: String) -> Data? {
        let bundle = Bundle(for: type(of: self))
        let path = bundle.url(forResource: name,
                              withExtension: "json")!
        return try? Data(contentsOf: path)
    }
}
```

6. A pretty common way of mocking a `URLSession` call is to create a custom `URLProtocol`. Create a class that conforms to `URLProtocol` by overriding the following functions:

```
final class MockURLProtocol: URLProtocol {
    override class func canInit(with request:
        URLRequest)
        -> Bool {
        return true
    }

    override class func canonicalRequest(for request:
        URLRequest) -> URLRequest {
        return request
    }
}
```

7. Add a property where we can inject a function to handle the request:

```
final class MockURLProtocol: URLProtocol {  
    static var requestHandler: ((URLRequest) throws  
        -> (HTTPURLResponse, Data?))?  
    //...  
}
```

8. Finally, override the functions that were called when a request call starts and finishes loading them:

```
final class MockURLProtocol: URLProtocol {  
    //...  
    override func startLoading() {  
        guard let handler = MockURLProtocol  
            .requestHandler else {  
            fatalError("Handler is unavailable.")  
        }  
  
        do {  
            let (response, data) = try  
                handler(request)  
            client?.urlProtocol(self, didReceive: response,  
                cacheStoragePolicy: .notAllowed)  
            if let data = data {  
                client?.urlProtocol(self, didLoad: data)  
            }  
            client?.urlProtocolDidFinishLoading(self)  
        } catch {  
            client?.urlProtocol(self,  
                didFailWithError: error)  
        }  
    }  
    override func stopLoading() {  
    }  
}
```

9. We want to test that the call returns a list of 30 users and that the first one has an `id` of 1. Let's write our test accordingly:

```
class GithubUsersAppTests: XCTestCase {
    let apiURL = URL(string:
        "https://api.github.com/users")!
    func testUsersCallResult() throws {
        // Arrange
        URLProtocol.registerClass(MockURLProtocol.self)
        MockURLProtocol.requestHandler = { request in
            let response = HTTPURLResponse(url:
                self.apiURL,
                statusCode: 200,
                httpVersion: nil,
                headerFields: nil)!

            return (response, self.loadFixture(named:
                "githubUsers"))!
        }
        let github = Github()

        let exp1 = expectValue(of: github.$users,
            equalsTo: { $0.first?.id == 1 })
        let exp2 = expectValue(of: github.$users,
            equalsTo: { $0.count == 30 })

        // Act
        github.load()

        // Assert
        wait(for: [exp1.expectation,
            exp2.expectation],
            timeout: 1)
    }
}
```

- Finally, implement the `expectValue()` function to verify the result of Publisher:

```
extension XCTestCase {  
    typealias CompletionResult = (expectation:  
        XCTestExpectation,  
        cancellable: AnyCancellable)  
    func expectValue<T: Publisher>(of publisher: T,  
                                    equalsTo closure:  
                                    @escaping (T.Output)  
                                    -> Bool)  
        -> CompletionResult {  
        let exp = expectation(description: "Correct values  
            of " + String(describing:  
                publisher))  
        let cancellable = publisher  
            .sink(receiveCompletion: { _ in },  
                  receiveValue: {  
                      if closure($0) {  
                          exp.fulfill()  
                      }  
                  })  
        return (exp, cancellable)  
    }  
}
```

By running the test, we should see that the test we wrote is green:

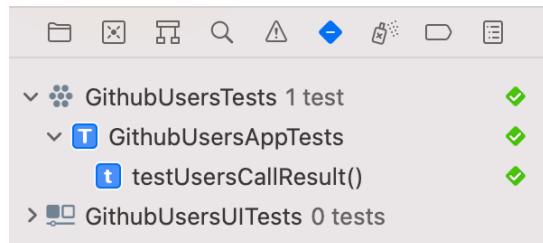


Figure 10.16 – Test result

How it works...

Explaining the benefits of unit testing is beyond the scope of this book, but I think you will agree with me that testing is something that should be part of every professional developer's skill set.

The trick here is to realize that any `@Published` variable is syntactic sugar for a Combine publisher. SwiftUI creates an automatic publisher called `$variable` for any `@Published` variable.

Given that it is a publisher, we can subscribe to it to test its value. For this, we created the `expectValue()` convenience function, to which we pass a closure to verify the expected value. I believe that this could be a useful tool to have in your test toolbox.

To understand it better, try to play with it – for example, see what happens when you change the first check with the following code:

```
let exp1 = expectValue(of: github.$users,  
    equalsTo: { $0.first?.id == 2 })
```

When and where do you think it is going to fail?

Also, try to change the code of the `expectValue()` function and make it fail if the closure is `false`: is it going to be useful?

11

SwiftUI Concurrency with `async await`

One of the most important features of **Swift 5.5**, released together with **Xcode 13**, is the introduction of the `async` and `await` keywords. With `async` and `await`, we can manage asynchronous concurrent code almost as if it was synchronous code.

Concurrency means that different pieces of code run at the same time. Often, we must orchestrate these pieces of code to create sequences of events to present the results in a view.

Before Swift 5.5, the most common way of creating a sequence of concurrent code was by using a **completion block**. When the first part of code finishes, we call a completion block where we start the second piece of code. This works and is manageable if we have only two asynchronous functions to synchronize. But it would become quickly unmaintainable with multiple functions and different ways of synchronizing them. For example, we could have two asynchronous functions to wait before starting the third one. With completion block functions, we must use some other primitive (such as a semaphore) to synchronize them before starting the third function. This would make the code more complicated than it should be.

Taking inspiration from a well-known pattern in other languages such as **JavaScript**, **Kotlin**, and **C#**, Swift provides `async await` to handle this scenario. An asynchronous function must be decorated with the `async` and the calling function must add the `await` before calling the asynchronous function. In this way, we can make our asynchronous code readable and maintainable.

In this chapter, we'll see how the new model fits into the **SwiftUI** model. An obvious scenario for asynchronous code is fetching data from the network – fortunately, **iOS** supports this in an elegant way. What if a package we use (or some of our old code) still has the old completion block way of dealing with concurrency? No worries, because in this chapter, we'll see a recipe to transform the completion block pattern to the `async await` interface.

Finally, we'll implement a list with infinite scrolling using SwiftUI and `async` functions.

This chapter covers the basics of `async await` with the following recipes:

- Integrating SwiftUI and an `async` function
- Fetching remote data in SwiftUI
- Pulling and refreshing data asynchronously in SwiftUI
- Converting a completion block function to `async await`
- Implementing infinite scrolling with `async await`

Technical requirements

The code in this chapter is based on Xcode 13.

You can find the code in the book's Github repository:

<https://github.com/PacktPublishing/SwiftUI-Cookbook-2nd-Edition/tree/main/Chapter11-SwiftUI-concurrency-with-async-await>

Integrating SwiftUI and an `async` function

As mentioned in the introduction of this chapter, the `async await` model fits well in the SwiftUI model.

SwiftUI views offer support for calling asynchronous functions. Also, while a function is concurrently executing, we can change the view without having it blocked.

In this short recipe, we'll integrate an `async` function that suspends its execution for a few seconds, and at the same time, we can use a button to increase a counter shown in the view.

Getting ready

Create an iOS 15 SwiftUI app called `AsyncAwaitSwiftUI`.

How to do it...

We will create a simple app with a button that increases a counter and an asynchronous function that blocks for 5 seconds before returning a value to be presented in the view.

The steps are as follows:

1. Create a `Service` class with the following two functions:

```
class Service {

    func fetchResult() async -> String {
        await sleep(seconds: 5)
        return "Result"
    }

    private func sleep(seconds: Int) async {
        try? await Task.sleep(nanoseconds:
            UInt64(seconds * 1000000000))
    }
}
```

2. In `ContentView`, add two `@State` properties, as shown in the following code:

```
struct ContentView: View {
    let service = Service()
    @State var value: String = ""
    @State var counter = 0

    var body: some View {
    }
}
```

3. Implement the following body in ContentView:

```
struct ContentView: View {  
    //...  
    var body: some View {  
        VStack {  
            Text(value)  
            Text("\(counter)")  
            Button {  
                counter += 1  
            } label: {  
                Text("increment")  
            }  
            .buttonStyle(.bordered)  
        }  
    }  
}
```

4. Finally, add a `.task{ }` modifier to `VStack`, calling the `fetchResult()` function:

```
var body: some View {  
    VStack {  
        //...  
    }  
.task {  
    value = await service.fetchResult()  
}
```

When running the app, we can change the value of the counter while waiting for the function to return the result:



Figure 11.1 – SwiftUI and an async function

How it works...

In this recipe, we saw how to declare an asynchronous function – you just add `async` to the function name. The `Task.sleep()` function is asynchronous, so we have to call it using the `await` keyword, as shown in the following code:

```
try? await Task.sleep(nanoseconds:  
    UInt64(seconds * 1000000000))
```

The `.task{ }` modifier in `VStack` works like `.onAppear{ }`, with the difference that you can have an `async` function to be called inside its block:

```
.task {  
    value = await service.fetchResult()  
}
```

If you replace `.task{ }` with `.onAppear{ }`, the code doesn't compile anymore since in `.onAppear{ }`, you can't call an `async` function:

```

22             Text("increment")
23         }
24     .buttonStyle(.bordered)
25   }
26   .onAppear { [2 ✘] Invalid conversion from 'async' function of type '()...
27   ✘ Invalid conversion from 'async' function of type '() async -> ()' to
28   synchronous function type '() -> Void'
29   ✘ Invalid conversion from 'async' function of type '() async -> ()' to synchronous
30   function type '() -> Void'
31
32 class Service {

```

Figure 11.2 – A compile error for an `async` function call in `.onAppear{}`

A final note on the thread where the functions are executed. In general, an `async` function runs in a background thread. You can see it setting a breakpoint in it, as shown in the following image:

```

32 class Service {
33
34     func fetchResult() async -> String {
35         await sleep(seconds: 5)
36         return "Result"   └─ Thread 2: breakpoint 1.1 (1)
37     }
38
39     private func sleep(seconds: Int) async {
40         try? await Task.sleep(nanoseconds:
41             UInt64(seconds * 1000000000))
42 }

```

Figure 11.3 – The `async` function in a background thread

To update a view, SwiftUI requires that the code runs in the main thread. `.task{ }` does this for us by moving the code execution from the background thread to the main thread, as shown in the following image:

```

25     }
26     .task {
27         value = await service.fetchResult()
28     }   └─ Thread 1: breakpoint 2.1 (1)
29   }
30 }
31
32 class Service {

```

Figure 11.4 – Updating the view in the main thread

Fetching remote data in SwiftUI

One of the operations made easier by the `async await` model is *network operations*. Fetching data from a network resource is an asynchronous operation by definition and until now, we had to manage it with the callback mechanism.

iOS 15 adds an `async await` interface to the `URLSession` class to embrace the new pattern. In this recipe, we'll implement a class to download data from a remote service and present it in a SwiftUI view.

We'll use a free service called **SpamWords** (<https://www.spam-words.com>) that hosts a list of words to detect if a message is spam or not. The services have multiple lists, grouped by language.

The app we'll implement will have a view to select the language and another view to present the spam words for the language we selected.

Getting ready

Create an Xcode 13 SwiftUI app called `SpamWords`. After creating it, be sure to set the target deploy to iOS 15.

How to do it...

We are going to implement the app starting from the service, and then connect it to the views.

Let's proceed as follows:

1. Create the model objects to transform the **JSON** returned from the network call to Swift struct:

```
struct Language: Decodable, Identifiable {
    var id: String { code }
    let code: String
    let label: String
}

struct LanguageList: Decodable {
    let codeLanguages: [Language]
}

struct SpamWords: Decodable {
```

```
    let words: [String]
}
```

2. Create a Service struct to call the network service and decode the response into a Decodable model:

```
struct SpamWordsLanguagesService {
    private let decoder: JSONDecoder = {
        let decoder = JSONDecoder()
        decoder.keyDecodingStrategy =
            .convertFromSnakeCase
        return decoder
    }()

    private func fetch<T: Decodable>(type: T.Type,
                                      from urlString: String) async -> T? {
        guard let url = URL(string: urlString) else {
            return nil
        }
        do {
            let (data, _) = try await URLSession
                .shared
                .data(from: url)

            return try decoder.decode(type, from: data)
        } catch {
            return nil
        }
    }
}
```

3. Finally, add two async functions to the `SpamWordsLanguagesService` struct to call the `language` and `words` endpoints:

```
struct SpamWordsLanguagesService {
    //...
    func fetchLanguages() async -> [Language] {
        await fetch(type: LanguageList.self,
```

```
        from: "https://www.spam-
words.com/api/languages") ?
.codeLanguages ?? []
}

func fetchWords(language: Language) async ->
[String] {
    await fetch(type: SpamWords.self,
from: "https://www.spam-words.com/api/words/"
+ language.code)?
.words ?? []
}
}
```

4. Add a List view in ContentView to present the supported languages:

```
struct ContentView: View {
    let service = SpamWordsLanguagesService()
    @State var languages: [Language] = []

    var body: some View {
        NavigationView {
            List(languages) { language in
                NavigationLink(destination:
                    SpamWordsView(language: language))
                {
                    Text(language.label)
                }
            }
            .navigationTitle("Languages")
        }
        .listStyle(.plain)
    }
}
```

5. To update `languages` `@State` property, add a `.task{ }` modifier to `NavigationView`:

```
struct ContentView: View {  
    //...  
    var body: some View {  
        NavigationView {  
            //...  
            }  
            .listStyle(.plain)  
            .task {  
                languages = await service.fetchLanguages()  
            }  
        }  
    }  
}
```

6. Now implement a view to show the words for the selected language. Create a `SpamWordsView`, as shown in the following code:

```
struct SpamWordsView: View {  
    let language: Language  
  
    let service = SpamWordsLanguagesService()  
    @State var words: [String] = []  
  
    var body: some View {  
        List(words, id: \.self) { word in  
            Text(word)  
        }  
        .navigationTitle(language.label)  
    }  
}
```

7. Similarly, add a `.task{ }` modifier to the `List` instruction, calling the `fetchWords()` function:

```
List(words, id: \.self) { word in  
    Text(word)  
}
```

```
.navigationTitle(language.label)
.task {
    words = await service.fetchWords(language: language)
}
```

The app is now ready, and we can test it by opening and selecting a language. Then, we can see the list of spam words for that language:

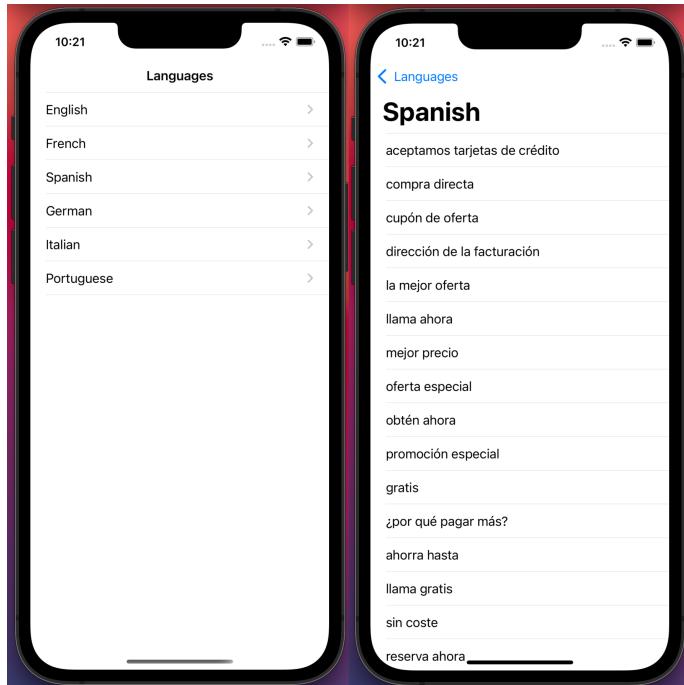


Figure 11.5 – Selecting a language and seeing its spam words

How it works...

Apple made a good effort of converting all the block-based APIs in the iOS foundation to also have an `async await` counterpart. Prior to iOS 15, `URLSession` had the following interface:

```
URLSession
.shared
.dataTask(with: req) { data, response, error in
    //...
}
```

With this block-based API, we were forced to chain the decoding and the handling of the response in another callback function, risking the antipattern usually called *callback hell*.

The iOS 15 version of this API is more readable, as you can see from the following code:

```
let (data, _) = try await URLSession
    .shared
    .data(from: url)
```

With this format, all other operations can be done in sequence without having to nest multiple callbacks.

Finally, as you can also see in the *Integrating SwiftUI and an async function* recipe, the `.task{}` modifier allows us to call an `async` function when the view appears and to update a `@State` property, triggering a re-evaluation of body of the view.

Pulling and refreshing data asynchronously in SwiftUI

Pull-to-refresh is an established touchscreen gesture to refresh a `List` view. First implemented in **Tweetie**, one of the first iOS apps for **Twitter**, it is now part of the **iOS SDK**. Originally supported only in **UIKit**, iOS 15 offers support for it in **SwiftUI**, and as expected, it works well with the `async await` model.

In this recipe, we are going to implement an app that fetches a few random cryptocurrencies and presents them in a `List` view.

Pulling the `List`, the app fetches another sample of the currencies updating the `List`. The network service we will use for this is called **Random Data Generator** (<https://random-data-api.com>). It offers several types of random data, as you can see from its help page:

<https://random-data-api.com/documentation>

The recipe fetches cryptocurrencies, but you can experiment with the same code using another endpoint.

Getting ready

Implement an iOS 15 SwiftUI app called `RefreshableCrypto`.

How to do it...

As typical for an app of this kind, it has two main parts: the service to fetch the data, and a view to present the data. We'll start with the service, and follow with the rendering views.

Let's get started:

1. Create a Coin Decodable struct to manage the response from the server:

```
struct Coin: Decodable, Identifiable {
    let id: Int
    let coinName: String
    let acronym: String
    let logo: String
}
```

2. Add a Service struct, with a JSONDecoder property to decode from *snake case* (`snake_case`) to *camel case* (`camelCase`):

```
struct Service {
    private let decoder: JSONDecoder = {
        let decoder = JSONDecoder()
        decoder.keyDecodingStrategy =
            .convertFromSnakeCase
        return decoder
    }()

    func fetchCoins() async -> [Coin] {
    }
}
```

3. The `fetchCoins()` connects to the network service, fetches the data, and returns an array of `Coin` ordered by the acronym:

```
struct Service {
    //...
    func fetchCoins() async -> [Coin] {
        guard let url = URL(string:
            "https://random-data-
            api.com/api/crypto_coin/
```

```
        random_crypto_coin?size=10")
    else {
        return []
    }
    do {
        let (data, _) = try await URLSession
            .shared
            .data(from: url)

        let list = try decoder.decode([Coin].self,
                                      from: data)
        return list.sorted {
            $0.acronym < $1.acronym
        }
    } catch {
        return []
    }
}
```

4. Let's move to ContentView. Add a List view to present the coins in a CoinView:

```
struct ContentView: View {
    let service = Service()
    @State var coins: [Coin] = []

    var body: some View {
        List(coins) {
            CoinView(coin: $0)
        }
        .listStyle(.plain)
    }
}
```

5. Create a CoinView to present the acronym and the logo of the crypto currency:

```
struct CoinView: View {
    let coin: Coin
    var body: some View {
        HStack {
            Text("\(coin.acronym) : \(coin.coinName)")
            Spacer()
            LogoView(coin: coin)
        }
    }
}
```

6. Add a LogoView, wrapping a SwiftUI AsyncImage, to show the logo of the crypto coin:

```
struct LogoView: View {
    let coin: Coin
    var body: some View {
        AsyncImage(
            url: URL(string: coin.logo),
            content: { image in
                image.resizable()
                    .aspectRatio(contentMode: .fit)
                    .frame(maxWidth: 40, maxHeight: 40)
            },
            placeholder: {
                ProgressView()
            }
        )
    }
}
```

- Finally, let's go back to `ContentView`, where we add the `.refreshable{ }` and `.task{ }` modifiers to the `List` view:

```
var body: some View {
    List(coins) {
        //...
    }
    .listStyle(.plain)
    .refreshable {
        coins = await service.fetchCoins()
    }
    .task {
        coins = await service.fetchCoins()
    }
}
```

When running the app, we can pull the list down to refresh the coins:

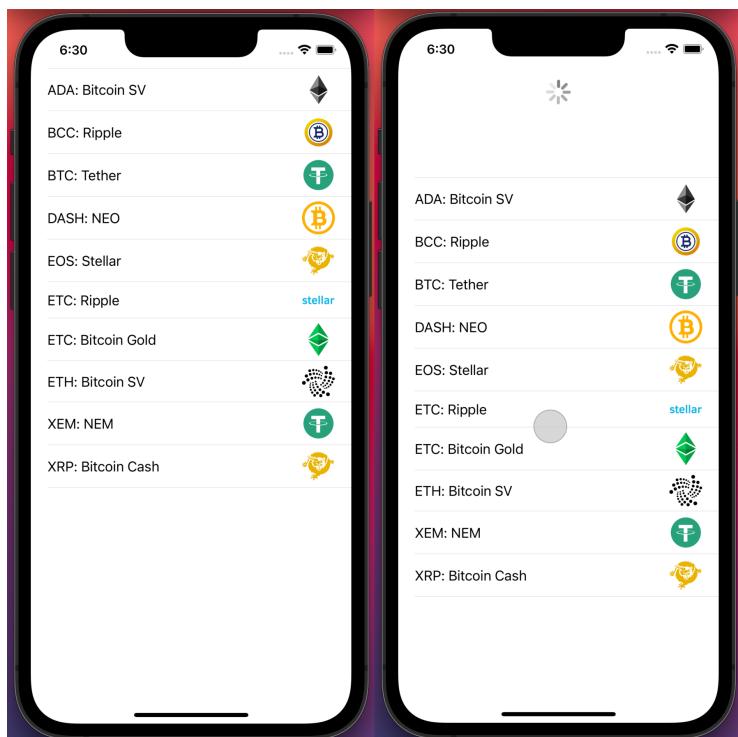


Figure 11.6 – Seeing and refreshing a list of Crypto currencies

How it works...

This is a pretty simple recipe. As in the *Fetching remote data in SwiftUI* recipe, we defined an `async` function, `fetchCoins()`, to do the network call using the new `URLSession data(from: URL)` function.

The `fetchCoins()` is then called inside the `.task{ }` modifier, which is similar to the `.onAppear{ }` modifier, but supports `async` function calls.

Finally, to add a pull-to-refresh control, we append the `.refreshable{ }` modifier that, similar to `.task{ }`, allows us to call `async` functions. When the `.refreshable{ }` modifier is triggered by a pull-to-refresh gesture, the `fetchCoins()` function is called in a different thread, and a spinner is presented on top of the `List` view.

When the `async` function has finished, the spinner disappears and the `coins @State` property is updated in the main thread, triggering a re-evaluation of the body of the `List` with the updated data.

Converting a completion block function to `async await`

It's clear that `async await` is the way that Apple wants us to develop concurrent code in Swift. But what if a framework we have to use (or even our legacy code) still has a completion block-based interface? Swift 5.5 provides a simple way to convert these APIs to `async await` functions.

In this recipe, we'll use an old framework where one of its functions has a completion block API. We'll convert it to an `async await` function and we'll use it in a SwiftUI view.

The package we will use is called **Lorikeet**, implemented by Þorvaldur Rúnarsson (<https://github.com/valdirunars>). Lorikeet is an aesthetic color scheme generator. Starting with one color, it generates a series of other colors that can be used for creating a color theme for an app.

Getting ready

Create an iOS 15 SwiftUI app called `PaletteGenerator`.

Import the SPM package from the following address- <https://github.com/gscalzo/Lorikeet>:

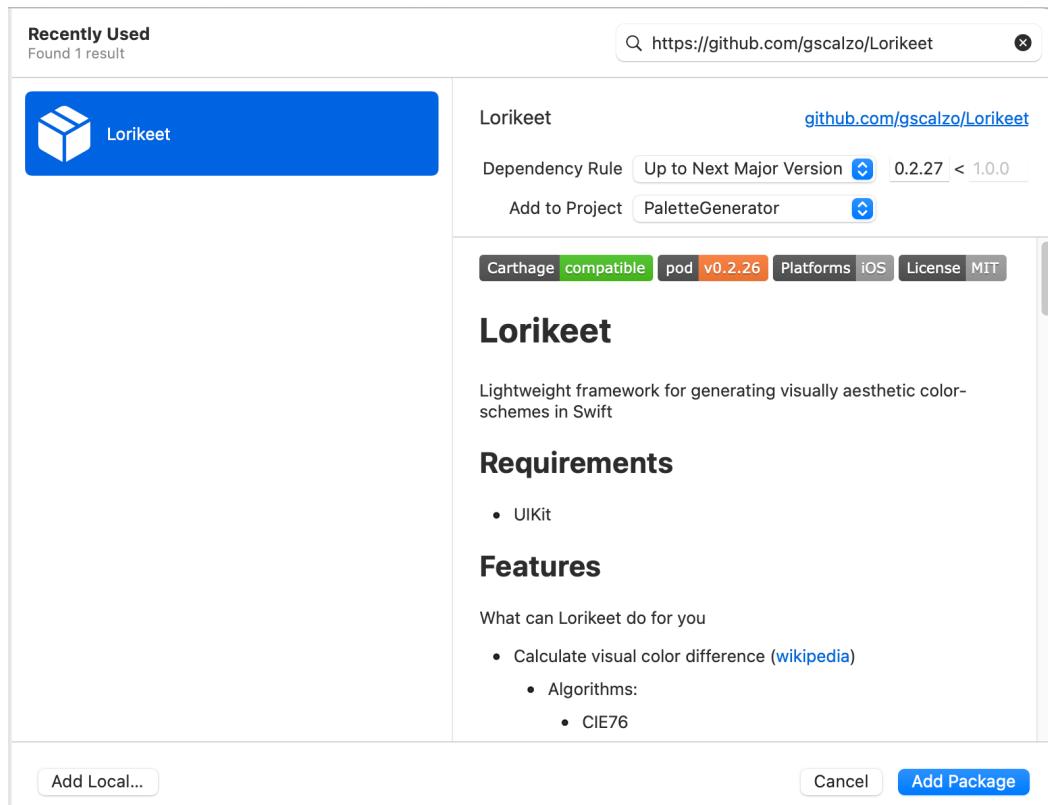


Figure 11.7 – Importing the Lorikeet package

How to do it...

The app shows a color scheme of 10 colors, starting from the default blue. Each color is a cell in a `List` view.

Let's get started:

1. Create a `ContentView` that shows a `List` view with a `Rectangle` for each cell. Each `Rectangle` has a different color:

```
struct ContentView: View {
    @State var colors: [Color] = []
}

var body: some View {
```

```

    ScrollView {
        LazyVStack(spacing: 0) {
            ForEach(colors){ color in
                Rectangle()
                    .foregroundColor(color)
                    .frame(height: 100)
            }
        }
        .edgesIgnoringSafeArea(.vertical)
    }
}

```

2. To use an array of Color in a ForEach block, the Color must conform to Identifiable. Add the conformance with the following code:

```

extension Color: Identifiable {
    public var id: String {
        self.description
    }
}

```

3. Lorikeet extends each UIColor with an lkt property. Since we want to use the SwiftUI Color instead of the UIKit UIColor, add the following code to do the transformation:

```

extension Color {
    var lkt: Lorikeet {
        UIColor(self).lkt
    }
}

```

4. Now, add the following async function to the Lorikeet class:

```

extension Lorikeet {
    func generateColorScheme(numberOfColors: Int,
                            withRange range: HSVRange? = nil,
                            using algorithm: Algorithm = .cie2000) async
        -> [Color] {
}

```

```
        await withCheckedContinuation { continuation in
            generateColorScheme(
                numberOfColors: numberOfColors,
                withRange: range,
                using: algorithm) { colors in
                    continuation
                .resume(returning: colors.map(Color.init))
            }
        }
    }
}
```

5. Finally, add a `.task{ }` modifier to `ScrollView` to generate the colors to show:

```
struct ContentView: View {
    //...
    var body: some View {
        ScrollView {
            //...
        }
        .edgesIgnoringSafeArea(.vertical)
        .task {
            colors = await Color.blue.lkt
            generateColorScheme(
                numberOfColors: 10)
        }
    }
}
```

When running the app, it shows the color scheme generated starting from the default blue color:



Figure 11.8 – The generated color scheme

How it works...

Lorikeet provides a completion block function for generating a color scheme, with the following signature:

We use the `withCheckedContinuation` function to transform it into an `async` function with the following signature:

The `withCheckedContinuation` function suspends the current task, and then it calls the passed callback with a `CheckedContinuation` object.

Inside the callback, we call the completion block-based function, and when it finishes, we resume the execution of the task via the `CheckedContinuation` that `withCheckedContinuation` provided.

With this mechanism, we interface synchronous and asynchronous code.

An important note here is that we must call `continuation.resume()` *exactly once* in the `withCheckedContinuation` block. If we forget to do it, our app would be blocked forever. If we do it twice, the app would crash.

Swift provides another function to deal with APIs that can return an error state. Let's say we have the following function:

```
func oldAPI(completion: (Result<[UIColor], Error>) -> Void)
```

We can use the `withCheckedThrowingContinuation` function to transform it into a function with the following signature:

```
func newAPI() async throws -> [UIColor]
```

The `withCheckedThrowingContinuation` function follows the same pattern as the `withCheckedContinuation`, as you can see from the following code:

```
func newAPI() async throws -> [UIColor] {
    try await withCheckedThrowingContinuation {
        continuation in
        oldAPI { result in
            switch result {
                case .success(let value):
                    continuation.resume(returning: value)
                case .failure(let error):
                    continuation.resume(throwing: error)
            }
        }
    }
}
```

What we have to do is call the old function and depending on the result, either it returns the value by resuming the continuation, or throws an error.

See also

If you want to know more about Lorikeet and the algorithms it uses to calculate the visual difference between colors to create a color scheme, you can refer to this page on Wikipedia:

https://en.wikipedia.org/wiki/Color_difference

Implementing infinite scrolling with `async await`

Infinite scrolling is a technique that loads data when the scrolling reaches the end of the List view. It is particularly useful when the list is backed by a paginated resource.

In this recipe, we'll use a remote API that returns paginated results. Every time the user scrolls to the end of the list, we load the next page, until we have loaded all the data.

For this recipe, we will use a pretty famous service for searching animated gifs called **Giphy** (<https://giphy.com>). Giphy provides a powerful API (<https://developers.giphy.com>), which we'll use to search and fetch a few animated gifs to present in a SwiftUI view.

Getting ready

Create an iOS 15 SwiftUI project called `CoolGifList`.

Since the `AsyncImage` SwiftUI component doesn't support animated gifs, we will use an external package called **NukeUI**. You can add it via SPM to the project with the following address (<https://github.com/kean/NukeUI>):

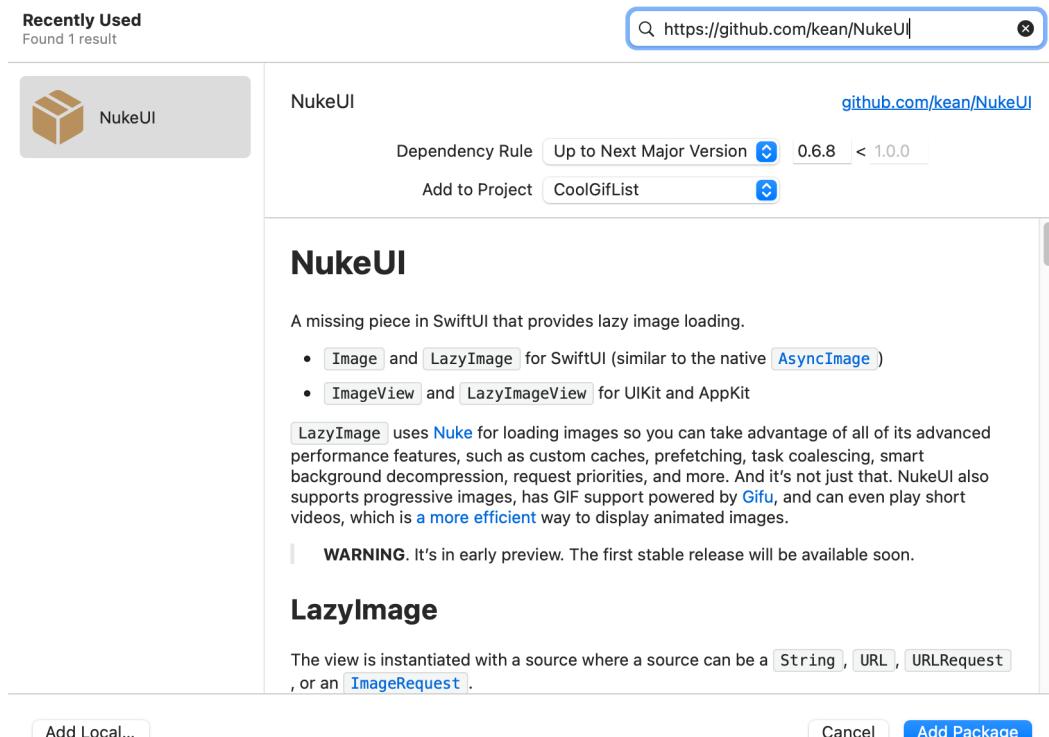
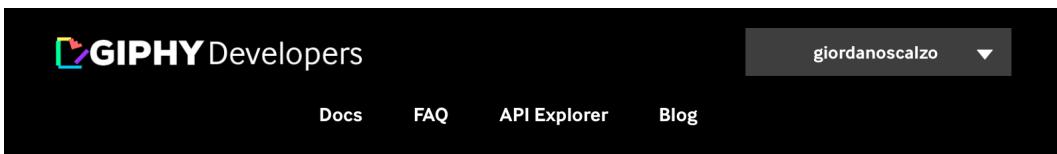


Figure 11.9 – Adding the NukeUI package

To use the Giphy API, we need an API Key.

First of all, create an account on Giphy. After the account is created, log in to <https://developers.giphy.com/dashboard/>, where we can create a new app:



Dashboard

Welcome to your GIPHY Developer Dashboard. Get started by creating an app, where you will be assigned a beta API key. All newly created beta keys are subject to rate limits and are best used in a development environment.

Once you are ready to use your app in production, please verify your GIPHY integration, if needed, and upgrade your key by clicking on 'Upgrade to Production'.

Your Apps

A screenshot of the "Your Apps" section of the Giphy API dashboard. It features a large plus sign icon centered on a light gray background. Below the icon is a message: "It looks like you don't have any apps yet! Click the button to get assigned an API Key and to start creating your first app. Check out our [docs](#) for more help getting started!". At the bottom of the section is a blue button with the text "Create an App".

Figure 11.10 – The Giphy API dashboard

Create a new app, and in the next screen, select the **API Selected** option:

Create A New App 1/2

We'll assign a restricted beta key for your project. Once your app is ready for production, you will need to apply to upgrade your key.

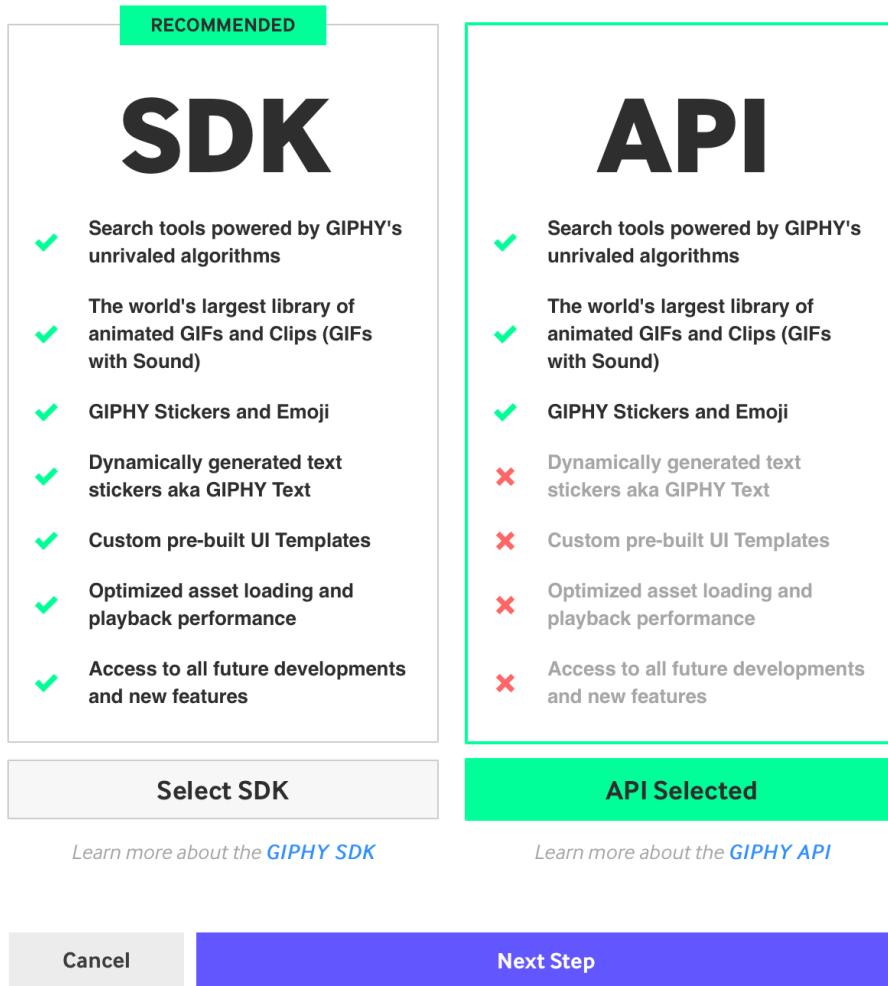


Figure 11.11 – Types of Giphy app

After moving to the next step, create an app called CoolGifApp:

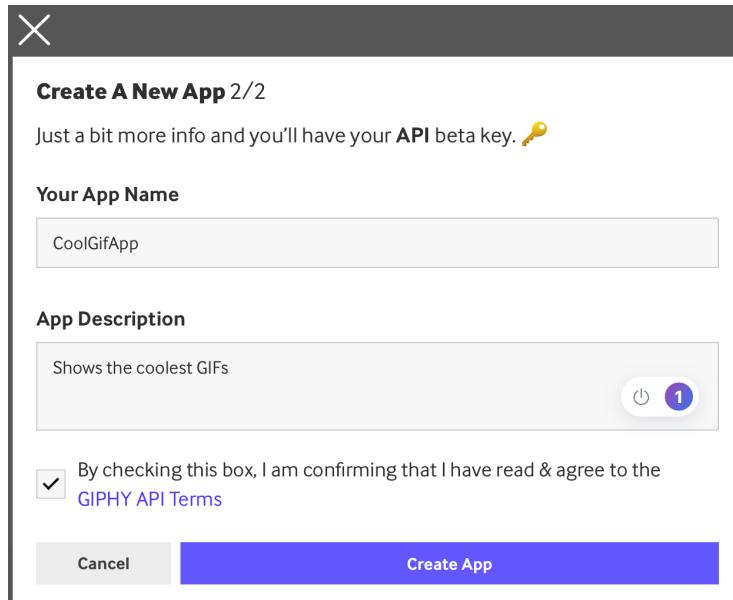


Figure 11.12 – Creating a Giphy app

Now, we have the API key to use in our code to access the Giphy API:

Dashboard

Welcome to your GIPHY Developer Dashboard. Get started by creating an app, where you will be assigned a beta API key. All newly created beta keys are subject to rate limits and are best used in a development environment.

Once you are ready to use your app in production, please verify your GIPHY integration, if needed, and upgrade your key by clicking on 'Upgrade to Production'.

Your Apps

A screenshot of the Giphy developer dashboard under the "Your Apps" section. On the left, there is a card for the app "CoolGifApp" which has an "API" badge. The card shows the "API Key" field with the value "f3Y... [REDACTED] Sqpve". There is an "Edit" link next to the API key field. At the bottom of this card is a blue "Upgrade to Production" button. On the right side of the dashboard, there is a large empty space with a plus sign (+) in the center, and text that says "Ready to build another app already? Click anywhere to get started!". At the bottom of this right section is a blue "Create an App" button.

Figure 11.13 – The Giphy app API key

We are now ready to start building the app.

How to do it...

The app we are building fetches a set of animated gifs from the API and presents them in a `List` view. As usual, there are two parts: a `Service` to interact with the API, and a `View` to present the data.

Your steps should be formatted like so:

1. Create the models to represent the gifs returned by the API:

```
struct Response: Codable {
    let data: [Gif]
}

struct Gif: Identifiable, Codable, Equatable {
    static func == (lhs: Gif, rhs: Gif) -> Bool {
        lhs.id == rhs.id
    }

    let id: String
    let title: String
    var url: String {
        images.downsized.url
    }
    let images: Images
}

struct Images: Codable {
    let downsized: Image
}

struct Image: Codable {
    let url: String
}
```

2. Add a Service struct, with a JSONDecoder property to decode from *snake case* (`snake_case`) to *camel case* (`camelCase`):

```
struct Service {
    private let decoder: JSONDecoder = {
        let decoder = JSONDecoder()
        decoder.keyDecodingStrategy =
            .convertFromSnakeCase
    }
    return decoder
}()

func fetchGifs(page: Int) async -> [Gif] {
}
}
```

3. Add a few properties to the Service struct:

```
struct Service {
    private let apiKey = <INSERT YOUR KEY>
    private let pageSize = 10
    private let query = "cat"
    //...
}
```

4. The `fetchGifs(page:)` connects to the network service, fetches the data for the passed page, and returns an array of Gif:

```
func fetchGifs(page: Int) async -> [Gif] {
    let offset = page * pageSize
    guard let url = URL(string:
        "https://api.giphy.com/v1/
        gifs/search?api_key=\(apiKey)&q=
        \(query)&limit=\(pageSize)&offset=\(offset)")
    else {
        return []
    }
    do {
        let (data, _) = try await URLSession
            .shared
```

```
        .data(from: url)
let response = try
    decoder.decode(Response.self,
                    from: data)
return response.data
} catch {
    print(error)
    return []
}
}
```

5. Let's move to ContentView. Add a List view to present the images and their title:

```
import NukeUI
struct ContentView: View {
    let service = Service()
    @State var gifs: [Gif] = []

    var body: some View {
        List(gifs) { gif in
            VStack {
                LazyImage(source: gif.url)
                    .aspectRatio(contentMode: .fit)
                Text(gif.title)
            }
        }.listStyle(.plain)
    }
}
```

6. Add the `.task{ }` modifier to the List view:

```
var body: some View {
    List(gifs) { gif in
        //...
        }.listStyle(.plain)
        .task {
            gifs = await service.fetchGifs(page: page)
        }
}
```

7. Finally, we add a `page` property and a `.task{ }` modifier to each cell of the list:

```
struct ContentView: View {
    //...
    @State var page = 1

    var body: some View {
        List(gifs) { gif in
            VStack {
                //...
            }
            .task {
                if gif == gifs.last {
                    page += 1
                    gifs += await service
                        .fetchGifs(page: page)
                }
            }
        }
        //...
    }
}
```

When running the app, we can see the animated gifs, loading a page at the time:



Figure 11.14 – The coolest gifs on the internet

How it works...

In this app, the fetching and presenting part is pretty much the same as in the *Fetching remote data in SwiftUI* recipe. I suggest checking the *How it works...* section from that recipe if you want to know more.

The difference here is that we add a `.task{ }` modifier to each cell, but we trigger a call to the service only if the cell is the last cell of the current set of loaded gifs.

When we fetch the last page, any subsequent calls will return an empty array, leaving the `gifs @State` property unaltered.

See also

This pattern was first proposed by Vincent Pradeilles (@v_pradeilles). You can see the original video on his **YouTube** channel at the following link:

<https://www.youtube.com/watch?v=hBTfLbSKZJw>

12

Handling Authentication and Firebase with SwiftUI

Since the creation of the mobile apps market, one of the most important features that most apps utilize is authentication. The normal way in which app creators ensure that authentication is provided is by ensuring that the user creates a new profile in the app they are using. However, this creates some problems regarding the user-friendliness of the app. This is because before the user can use the app, they must do something that could be considered time-consuming, which means they might leave the app without using it.

There are also security concerns here. Because creating a new profile for each app we use is a repetitive process, and we could be tempted to reuse a password that we've already used somewhere else. For example, if a data breach occurs on any of the apps we've used, and we've used the password for that app elsewhere, our other accounts that use that password are vulnerable to being hacked.

To overcome this problem, some big giants of the social app arena, including Facebook, Twitter, and Google, implemented a mechanism to authenticate a user for third-party apps: instead of using a new profile, we can use the one we already have on their respective sites. This is convenient for the user, as well as the app developer, as the only thing the user needs to do is tap on the **Login with Facebook** button to be redirected to the Facebook app and use it to authenticate the third-party app.

This seems to be the perfect solution, doesn't it? Unfortunately, there are concerns about the privacy of the user. Even though the authenticator service won't know what we are doing in the app we are authenticating in, it still knows that we are using that particular app and when we log into it. This may not seem like an important issue at first glance, but it wouldn't be appropriate to hand out this kind of information if, for example, the app is related to health, or it is about sensitive topics such as a person's religion or sexuality.

There is another point to take into consideration here. When we use an app in iOS, we are already being authenticated via our Apple ID, so it would be convenient to use that information in other apps so that we can be authenticated automatically. Always attentive to user privacy, iOS provides the **Sign in with Apple** service, which does exactly this. A third-party developer can use this authentication system in their app easily for other social media login systems.

Since privacy is of paramount importance to Apple's clients, the Sign in with Apple service is implemented in such a way that authentication information never leaves your device, and everything is considered private. Furthermore, since the app usually asks for the email of the user, Sign in with Apple enforces its own privacy regulations. Here, it creates an anonymous email that you can link to your real one; it then sends the anonymous one to the app.

In the first recipe of this chapter, you will learn how to use Sign in with Apple in a SwiftUI app to exploit this nice, powerful capability of iOS. However, social login functionalities aren't going anywhere, so we'll also learn how to integrate the login functions of Google into our SwiftUI apps.

To simplify their integration, we'll use a common platform service called **Firebase**, which allows us to integrate different authentications in a simple and uniform way. One of the most used features of Firebase is its distributed database on the cloud called **Firestore**.

As you will see, we'll spend more time configuring and adjusting the compilation of the app rather than writing code. This could be considered the boring part, but you will use the information you'll learn about in these recipes every time you implement new authentication or a new Firebase-based app. By doing this, you'll see how convenient it is to have all the necessary steps in the same place so that you can return to them for reference.

In this chapter, you'll learn how to handle authentication with Sign in with Apple and the Firebase service, a place where we can store a distributed database.

We are going to cover the following recipes in this chapter:

- Implementing Sign in with Apple in a SwiftUI app
- Integrating Firebase into a SwiftUI project
- Using Firebase to sign in using Google
- Implementing a distributed Notes app with Firebase and SwiftUI

Technical requirements

The code in this chapter is based on Xcode 13 and iOS 15.

You can find the code in the book's GitHub repository under the path <https://github.com/PacktPublishing/SwiftUI-Cookbook-2nd-Edition/tree/main/Chapter12-Handling-authentication-and-Firebase-with-SwiftUI>.

Implementing Sign in with Apple in a SwiftUI app

In this recipe, you'll learn how to use Sign in with Apple in a SwiftUI app. Apple enforces the use of this method for authentication, making it mandatory if an app uses a third-party social login such as Facebook or Google, so it's a useful skill to learn.

Sign in with Apple is the official method that Apple uses for authentication and SwiftUI supports it natively.

We are going to implement a simple app that permits us to log in using our Apple ID and presents our credentials once we are logged in.

Important Note

Sign in with Apple doesn't work reliably with a simulator, so for this recipe, I recommend using a real iOS device.

The app we are going to implement is very basic, but it will give you the foundation for building something more sophisticated. However, there are a couple of points that we must take into consideration:

- First, the framework will only pass the user's credentials the first time they log in. So if they are of importance to us, we must save them in an appropriate manner. For simplicity, we'll save them in `UserDefault`s using the iOS `@AppStorage ("name")` property wrapper. Please don't do this in your production app since `UserDefault`s is not secure. These credentials should be stored in the keychain.
- Second, Apple doesn't provide a Logout API. So the user must go into the **Settings** part of iOS and remove the authentication properties. It means that at startup, our app must check whether the credentials are still valid. We'll cover this as well.

Now, let's start to implement the app.

Getting ready

Create a SwiftUI app called `SignInWithApple`.

How to do it...

As usual, with features that rely on security or privacy, we must configure the capabilities of the app in the Xcode project:

1. First, we must add the capability to the app so that the app has permission to use Sign in with Apple. Select the **Signing & Capabilities** tab in the target configuration and search for `SignInWithApple`:

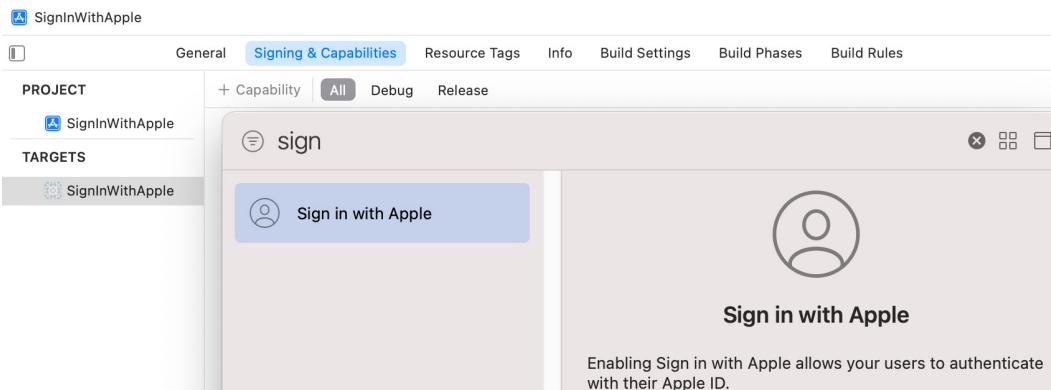


Figure 12.1 – Searching for the Sign in with Apple capability

- After selecting Sign in with Apple, check that there is a new entry in the project, indicating the newly added capability:

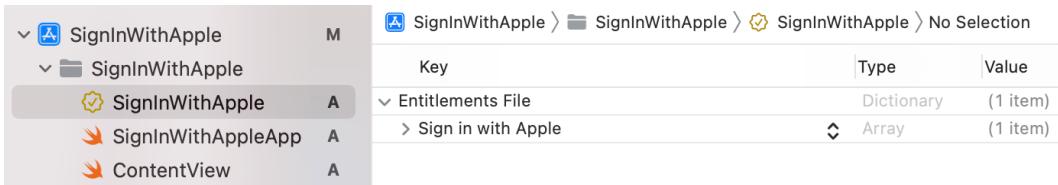


Figure 12.2 – Sign in with Apple added to the project

- Now, let's move on to the code. Here, we are creating the main ContentView, which will show a message when the user is logged in, or the SignInWithApple button when the user starts the app for the first time:

```
import AuthenticationServices

struct ContentView: View {
    @State
    private var userName: String = ""
    @State
    private var userEmail: String = ""

    var body: some View {
        ZStack{
            Color.white
            if userName.isEmpty{
                //...
            } else {
                Text("Welcome\n\(userName), \(userEmail)")
                    .foregroundColor(.black)
                    .font(.headline)
            }
        }
    }
}
```

4. Since we imported the `AuthenticationServices` framework, SwiftUI provides us with a native button called `SignInWithAppleButton`. The button needs two callbacks: one for configuring the request and another for receiving the sign-in result. Add the button and the callbacks in the `ContentView` struct:

```
struct ContentView: View {  
    //...  
    var body: some View {  
        //...  
        if userName.isEmpty{  
            SignInWithAppleButton(.signIn,  
                onRequest: onRequest,  
                onCompletion: onCompletion)  
                .signInWithAppleButtonStyle(.black)  
                .frame(width: 200, height: 50)  
        } else {  
            //...  
        }  
  
        private func onRequest(_ request:  
            ASAAuthorizationAppleIDRequest) {  
        }  
  
        private func onCompletion(_ result:  
            Result<ASAAuthorization, Error>) {  
        }  
    }  
}
```

5. Implement the `onRequest` function, where we ask the framework to return the full name and email of the user:

```
private func onRequest(_ request:  
    ASAAuthorizationAppleIDRequest) {  
    request.requestedScopes = [.fullName, .email]  
}
```

6. The `SignInWithApple` API only returns the credentials the first time we call it, so we must save them somewhere. Add two `@AppStorage` property wrappers to save them locally:

```
struct ContentView: View {  
    @AppStorage("storedName")  
    private var storedName : String = "" {  
        didSet {  
            userName = storedName  
        }  
    }  
    @AppStorage("storedEmail")  
    private var storedEmail : String = "" {  
        didSet {  
            userEmail = storedEmail  
        }  
    }  
    @AppStorage("userID")  
    private var userID : String = ""  
  
    //...  
}
```

7. The `onCompletion` function simply saves the result of the login class in local storage. Add the body of the function with the following code:

```
private func onCompletion(_ result:  
    Result<ASAuthorization,  
    Error>) {  
    switch result {  
        case .success (let authResults):  
            guard let credential = authResults.credential  
                as? ASAuthorizationAppleIDCredential  
            else { return }  
            storedName = credential.fullName?.givenName ?? ""  
            storedEmail = credential.email ?? ""  
            userID = credential.user
```

```
        case .failure (let error):
            print("Authorization failed: " +
                  error.localizedDescription)
    }
}
```

8. Finally, every time the app starts, it must verify whether the user is logged in or not. For this, add a `.task()` modifier to the main view component invoking an `authorize()` function:

```
struct ContentView: View {
    //...
    var body: some View {
        ZStack{
            //...
        }
        .task { await authorize() }
    }
}
```

9. Implement the `authorize()` function, where firstly we fetch the credentials for the user:

```
struct ContentView: View {
    //...
    private func authorize() async {
        guard !userID.isEmpty else {
            userName = ""
            userEmail = ""
            return
        }

        guard
            let credentialState =
                try? Await
                    ASAuthorizationAppleIDProvider()
                    .credentialState(forUserID: userID)
        else {
```

```
        userName = ""
        userEmail = ""
        return
    }
}
```

10. Finish the `authorize()` function, adding code to save the user details if the state is `.authorized` or resetting them otherwise:

```
struct ContentView: View {
    //...
    private func authorize() async {
        //...
        switch credentialState {
        case .authorized:
            userName = storedName
            userEmail = storedEmail
        default:
            userName = ""
            userEmail = ""
        }
    }
}
```

Now, we can run the app and use our **Apple ID** to sign in:

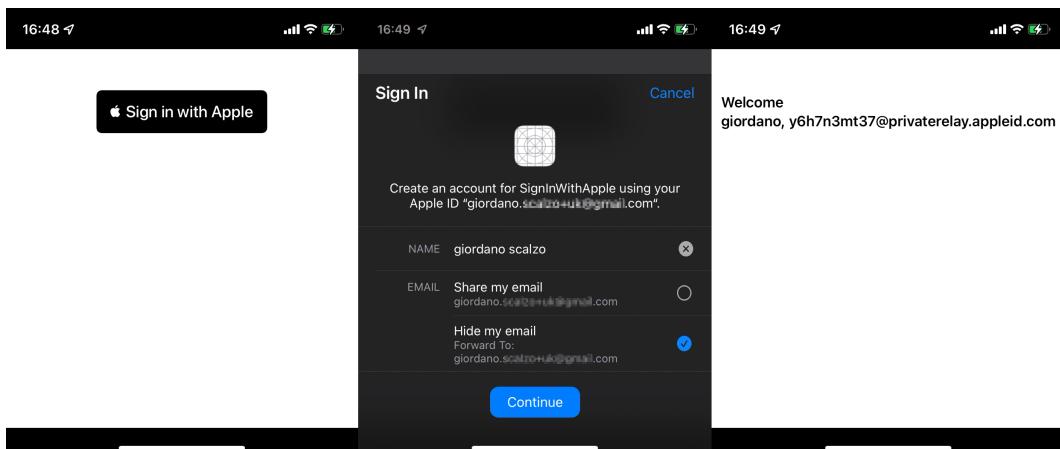


Figure 12.3 – Sign in with Apple in action

How it works...

The native SwiftUI Sign in with Apple implementation is an awaited iOS feature that enforces the Apple policy of using this mechanism for app authentication.

The `SignInWithAppleButton` component works in a SwiftUI way, where we pass two callbacks:

- In the first one, we configure the type of request we are going to make, what scope of credentials we require, and more.
- The second is called when the request is completed, and its result is shown. If it is a success, we can save the credentials somewhere, so that we can reuse them in the future and show them in the UI. By doing this, we can check if the credentials are still valid.

Being a view, `SignInWithAppleButton` doesn't check whether the credentials are still valid when the app starts. So in our recipe, we used the `.CredentialState()` function of `ASAAuthorizationAppleIDProvider()` to check whether the saved credentials are still valid.

The function is asynchronous, so we must call it with the `await` keyword and invoke it from the `.task()` modifier in the main view.

Sign in with Apple doesn't provide a *Sign-Out* API, so the logout process cannot be done inside the app. To log out from our app, the user must disable the credentials provided in the **Settings** section of iOS, in the **Password & Security** section of their profile, as shown in the following screenshot:

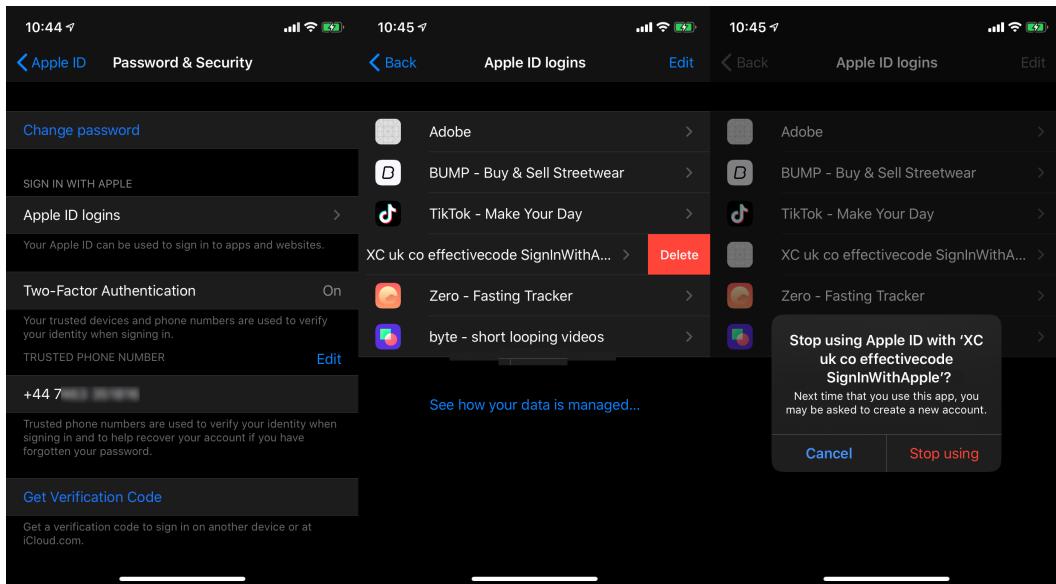


Figure 12.4 – Removing credentials for Apple ID login

Integrating Firebase into a SwiftUI project

Firebase is a mobile development platform that has several products that simplify the implementation of mobile apps. These apps need a backend for persistence, authentication, notifications, and more. A mobile developer can concentrate on implementing only the mobile app without worrying about implementing the services on the cloud that they need to power their app.

Firebase provides a framework so that its services can be used in Swift. Unfortunately, not all of them work smoothly in SwiftUI, so you need to apply some workarounds. Firebase is a pretty sophisticated service, and although it's definitely much easier to use than implementing a backend from scratch, there are a few steps to follow to configure it properly.

Hopefully, Firebase will release a version for SwiftUI soon. We'll start our exploration of Firebase in SwiftUI by integrating its **RemoteConfig** product. RemoteConfig is a service that lets the developer change the behavior or appearance of an app without requiring users to download an update of the app. In this recipe, we are going to implement two sample main screens so that the developer can select which one to present.

Getting ready

Create a new SwiftUI app called `RemoteConfig`.

Firebase supports the Swift Package Manager as a way of distributing its SDK.

Follow these steps to include the correct package:

- Firstly, add to the project a new **Swift Package Manager (SPM)** package. Set the following URL to find the right package: `https://github.com/firebase/firebase-ios-sdk.git`. Since we are going to use the new `async await` interface, we use the `master` branch for fetching the Firebase SDK:

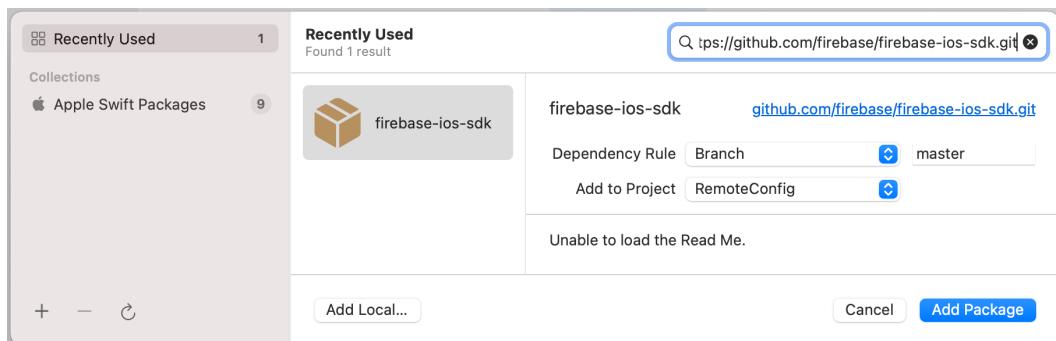


Figure 12.5 – Adding the Firebase package

- In the next step, SPM lists the available Firebase sub-packages. Select `FirebaseRemoteConfig`:

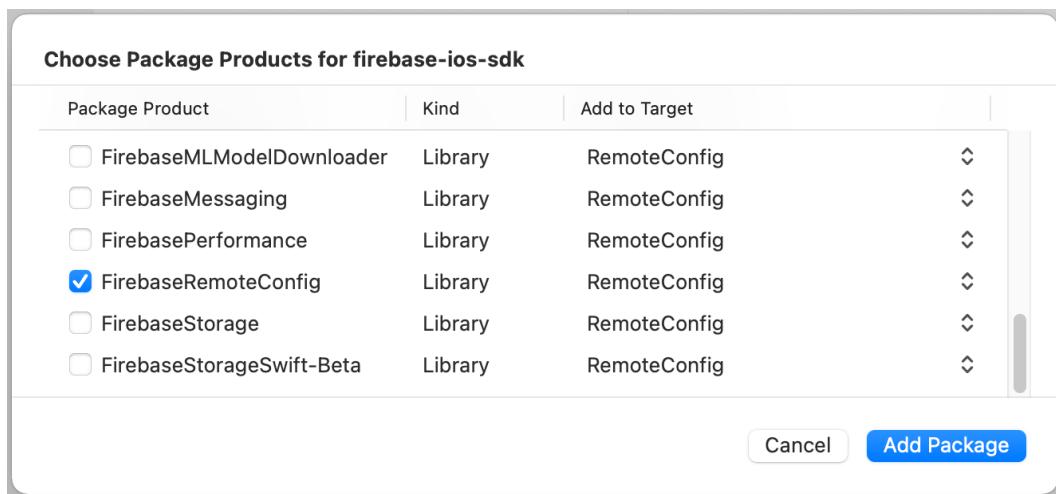


Figure 12.6 – Selecting the `FirebaseRemoteConfig` sub-package

3. After adding the sub-package, our project should look similar to the one in the following figure:

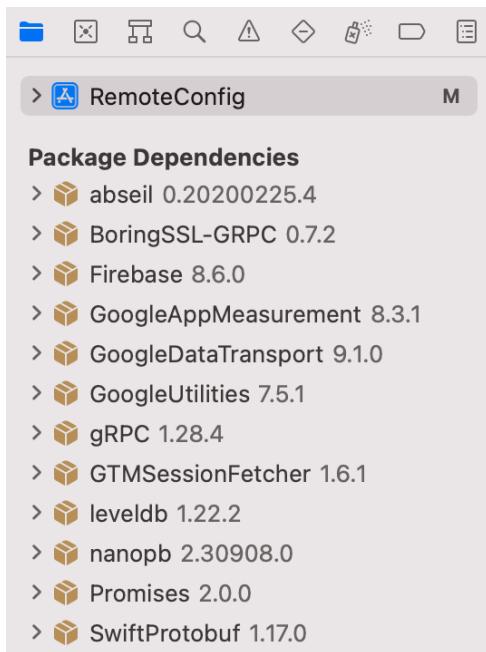


Figure 12.7 – The RemoteConfig project with Firebase packages

How to do it...

In this recipe, we'll encounter more configuration than code. Even though this may appear to be overkill, consider that this is the same amount of configuration for a Hello World application, as well as for a full-fledged social messenger app.

We'll use this recipe as a reference for creating a Firebase-based app. Follow these steps to get started:

1. Let's start by going to the main Firebase website, <https://firebase.google.com/>. After logging into the website with a valid Google account, click on **Get started** to be sent to the main console:

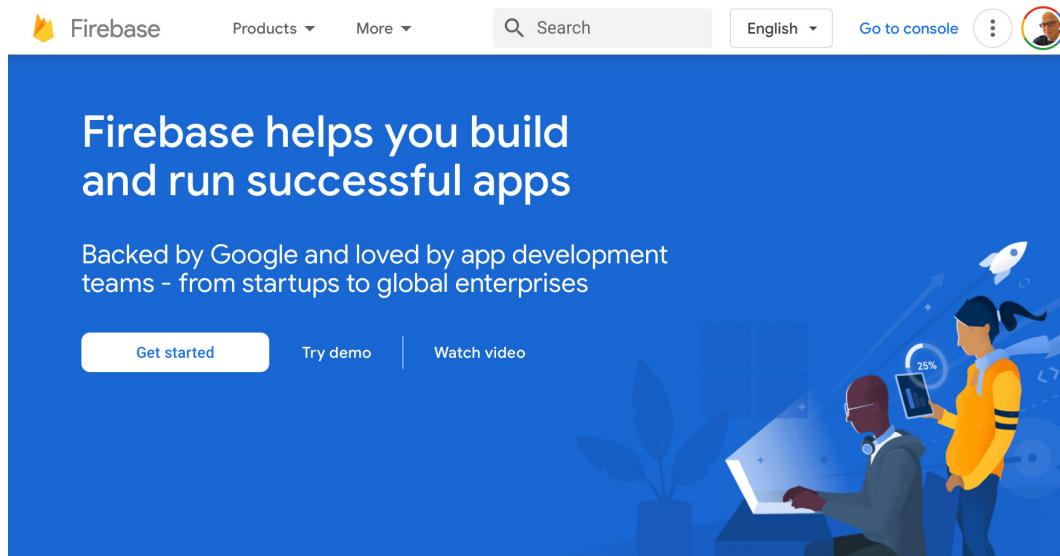


Figure 12.8 – The Firebase website

2. From the console, create a new project by selecting **Add project**:

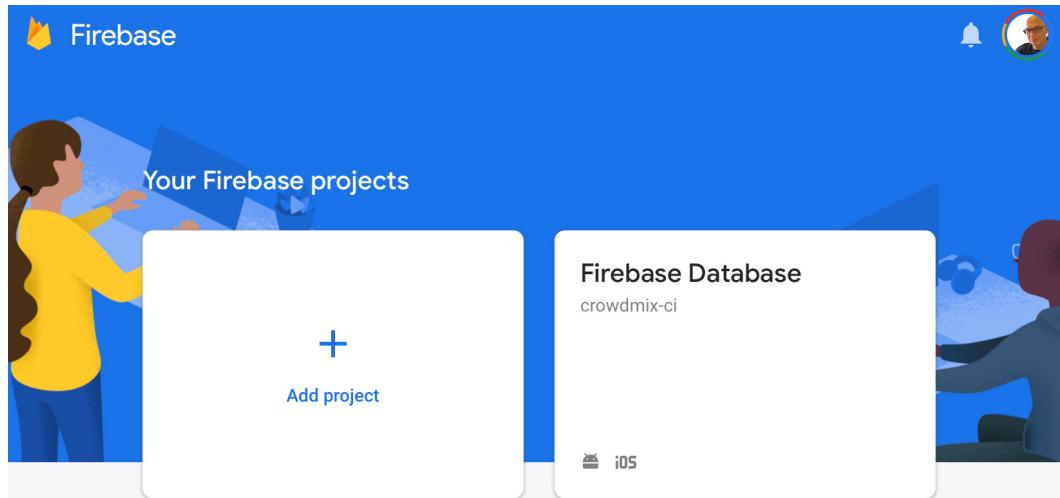


Figure 12.9 – The Firebase console

3. Use a mnemonic name for the project, for example, `FirebaseRemoteConfigApp`:

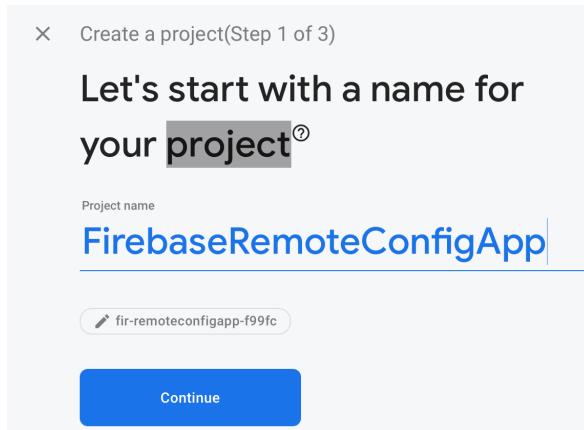


Figure 12.10 – Creating a new Firebase project

4. On the next screen, disable *Google Analytics* since we don't need it for this recipe:

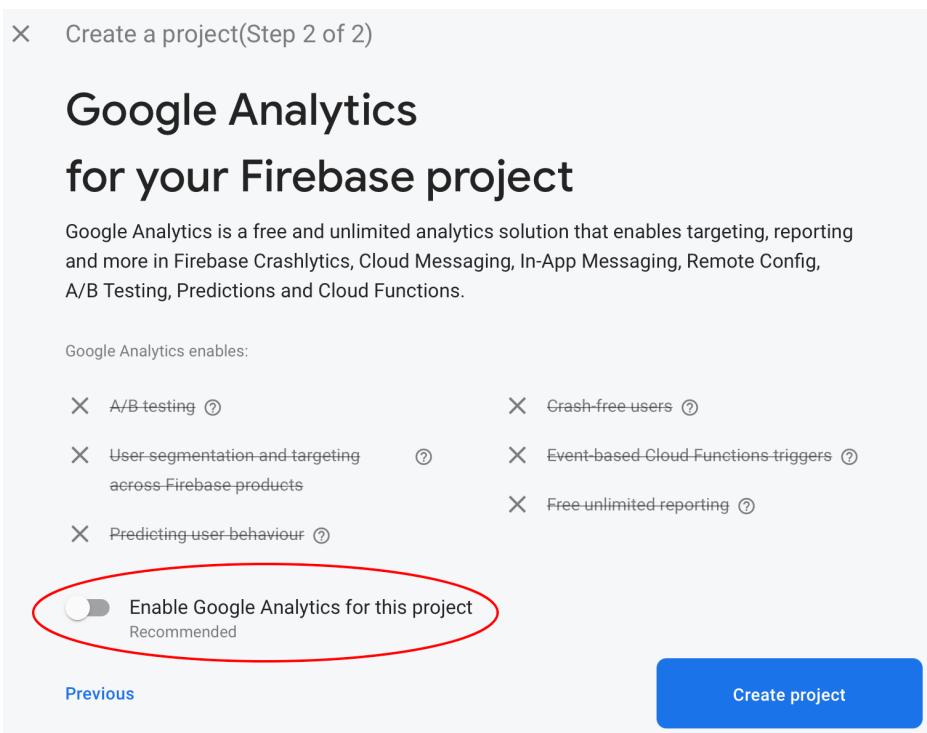


Figure 12.11 – Disabling Google Analytics

- Once you have created the project, it is time to configure Remote Config. Expand the **Engage** section and select the **Remote Config** option. Then, add a parameter called `screenType`, whose default value is `screenA`:

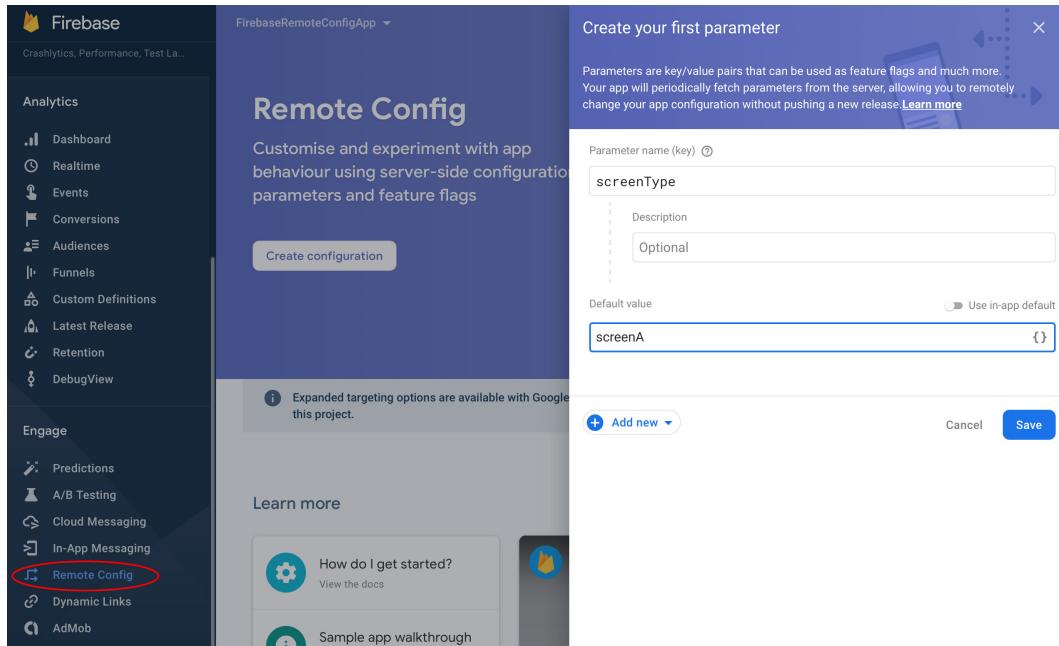


Figure 12.12 – Adding a Remote Config parameter

- After saving the parameter, we can publish it:

The screenshot shows the 'Parameters' section of the Firebase Remote Config interface. At the top, there's a yellow banner with the text 'Unpublished changes' and two buttons: 'Discard all' and 'Publish changes'. Below this, under 'Fetches – Last 24 hours', it says 'No fetch metrics are available for this template.' Under 'Config – Version 0', there's a table with columns: Name, Condition, Value, Fetch %, and Actions. A new row has been added with the name 'screenType', condition 'Default value', value 'screenA', and actions including a pencil icon for editing and three vertical dots for more options. The status of this parameter is 'Draft'.

Figure 12.13 – Remote Config parameter added

7. Publishing these changes makes them immediately available to our customers:

The screenshot shows a confirmation dialog box in the foreground. It contains the text 'Once you publish, changes are **immediately available** to your apps and users.' with two buttons at the bottom: 'Cancel' and a blue 'Publishing changes' button. In the background, the 'Parameters' section of the Remote Config interface is visible, showing the same table as Figure 12.13. A message at the bottom of the table says 'No fetch metrics are available for this template.'

Figure 12.14 – Publishing our parameter

- Once you have finished the **Remote Config** configuration, move back to the **Project Overview** page, and select the **iOS** button, as shown in the following screenshot:



Figure 12.15 – Configuring iOS for Firebase

- To prepare the Firebase configuration file, we must register the app with our **iOS bundle ID**:

A screenshot of the 'Add Firebase to your iOS app' registration form. It has two main sections: '1 Register app' and '2 Download config file'. In the '1 Register app' section, there are fields for 'iOS bundle ID' (containing 'co.uk.effectivecode.RemoteConfig'), 'App nickname (optional)' (containing 'My iOS app'), and 'App Store ID (optional)' (containing '123456789'). A large blue 'Register app' button is at the bottom. In the '2 Download config file' section, there is a smaller blue 'Download config file' button.

Figure 12.16 – Registering the iOS bundle ID

10. This bundle ID generates a .plist file, which must be imported into the app.
To do this, first, download the file:

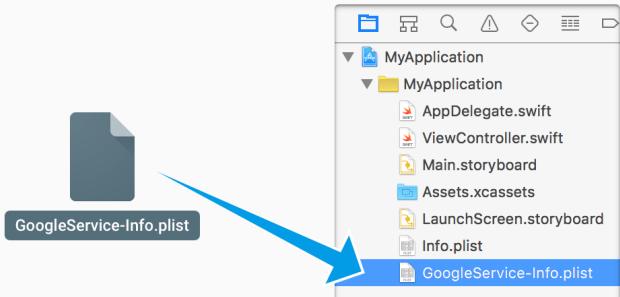
× Add Firebase to your iOS app

1 Register app
iOS bundle ID: co.uk.effectivecode.RemoteConfig

2 Download config file Instructions for Xcode below | [Unity](#) [C++](#)

[!\[\]\(81d4795ac6d7347a9f88d3b33135ca4f_img.jpg\) Download GoogleService-Info.plist](#)

Move the GoogleService-Info.plist file that you just downloaded into the root of your Xcode project and add it to all targets.



Next

3 Add Firebase SDK

Figure 12.17 – Firebase info.plist file

11. Once you've downloaded the file, drag it into the project, taking care to select **Copy items if needed**:

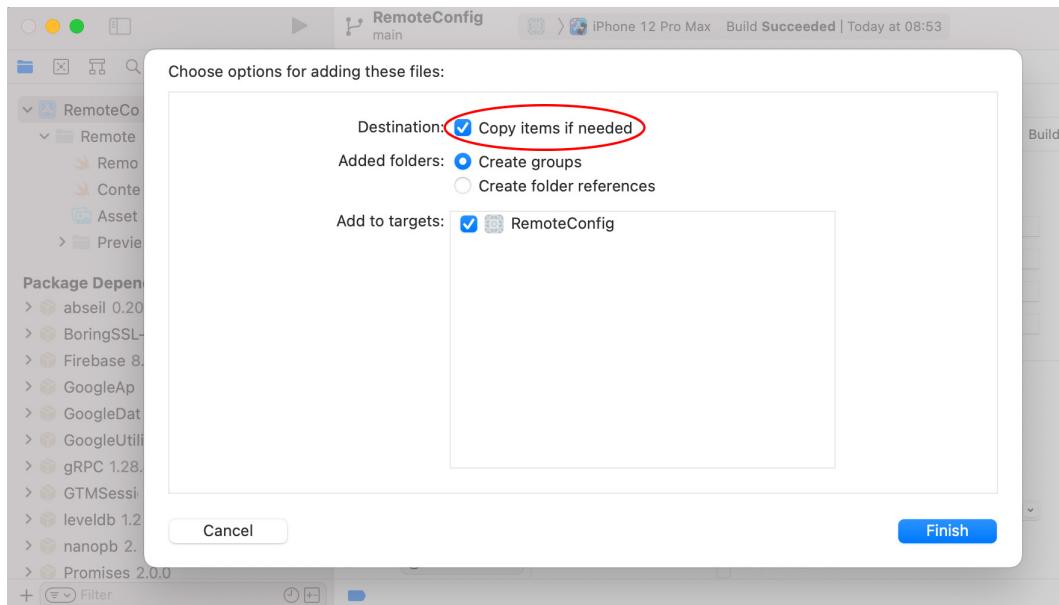


Figure 12.18 – Importing the GoogleService-Info.plist file

12. This is how our project should appear in Xcode:

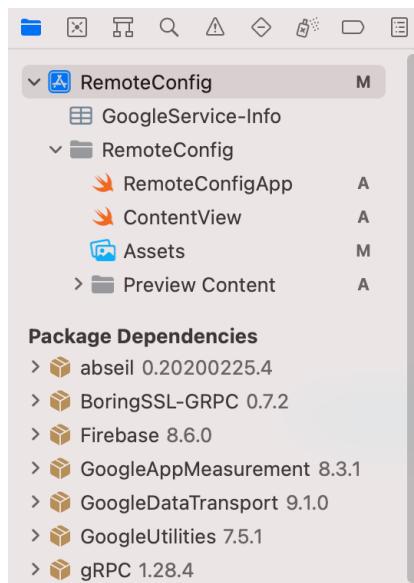


Figure 12.19 – Imported GoogleService-Info.plist file

13. Finally, let's move on to the code. The first thing we must do is configure Firebase in the `RemoteConfigApp` main struct:

```
import Firebase
@main
struct RemoteConfigApp: App {
    init() {
        FirebaseApp.configure()
    }
    var body: some Scene {
        WindowGroup {
            ContentView()
        }
    }
}
```

14. The next step is to implement the `ContentView` that depends on a `RemoteConfig` parameter:

```
struct ContentView: View {
    @StateObject
    private var remoteConfig = RemoteConfig()

    var body: some View {
        VStack {
            Text("This is the Screen")
                .font(.system(size: 50))
            if remoteConfig.activeScreen == "screenA" {
                Text("A")
                    .font(.system(size: 100))
            } else {
                Text("B")
                    .font(.system(size: 100))
            }
        }
    }
}
```

15. The `RemoteConfig` class is a simple wrapper around Firebase's `RemoteConfig` class:

```
import Firebase
class RemoteConfig: ObservableObject {
    @Published
    var activeScreen = "screenA"

    private var remoteConfig =
        Firebase.RemoteConfig.
    remoteConfig()
}
```

16. In the `init()` function, enable *Developer Mode* for Firebase's `RemoteConfig`:

```
init() {
    let settings = RemoteConfigSettings()
    settings.minimumFetchInterval = 0
    remoteConfig.configSettings = settings
}
```

17. Add a `refreshConfig()` `async` function to fetch the parameters from the remote storage, and change the local `@Published` property:

```
func refreshConfig() async {
    guard
        let status = try? await remoteConfig
        .fetch(withExpirationDuration: 1)
    else {
        return
    }
    guard case .success = status else {
        return
    }
    await update()
}
@MainActor func update() {
    activeScreen =
        remoteConfig["screenType"].stringValue
```

```
    ?? "screenA"  
}
```

18. Finally, the `refreshConfig()` function is invoked in the `.task()` modifier of the main view:

```
struct ContentView: View {  
    var body: some View {  
        VStack {  
            //...  
        }  
        .task {  
            await remoteConfig.refreshConfig()  
        }  
    }  
}
```

If you go to the Firebase console to change the value of the parameter and publish these changes, this will be reflected the next time you open the app:

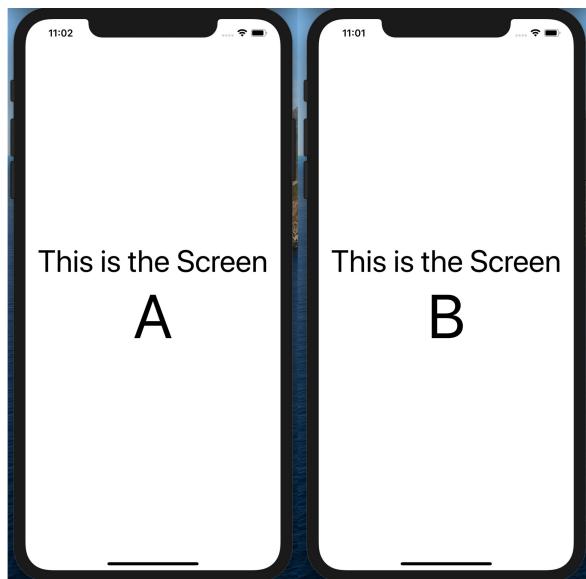


Figure 12.20 – Selecting the View from Remote Config

How it works...

Firebase is a really powerful service that, after doing a bit of tedious configuration, gives your app real superpowers.

In this recipe, you learned how to drive the appearance of our SwiftUI app from a remote configuration, without asking our users to download an update.

We've only just scratched the surface of Remote Config, though. You can also define conditions regarding when a particular parameter should be applied, such as applying a rule to only a particular version of iOS or to a percentage of our customer base. Usually, the changes are not immediate, and you can give a longer duration to your parameters – for example, one day – so that the APIs are not hit too much.

The Firebase SDK interfaces embrace the new `async await` paradigm, so the code is very concise, and we can invoke the `refreshConfig()` function in the `.task()` modifier of the main view. This function changes the `@Published activeScreen` property. Any change to it triggers a new rendering of the view. iOS requires that any rendering must be done in the main thread, but since `refreshConfig()` is an `async` function, this cannot be guaranteed.

To be sure that every call to the function `update()` happens in the main thread, we decorate it with the property wrapper `@MainActor`. It guarantees that the code in the `update()` function will be executed in the main thread, even if the function caller is executed in another thread.

There's more...

In our implementation, we only check the parameters while the app is being started up, but what about extending this and doing this more often during the life cycle of the app? Using a timer should do the trick – try it out!

Also, I encourage you to explore the conditional rules in Firebase's Remote Config, in order to learn how and when to apply them to your real app.

Using Firebase to sign in using Google

A social login is a method of authentication where the authentication is delegated to a trustworthy social networking service outside our app. A common social networking service that offers this kind of opportunity is Google.

In this recipe, you'll learn how to integrate a social login with Google using Firebase.

Getting ready

First, create a SwiftUI app called `GoogleLogin` with Xcode. After creating it, configure the project following these steps:

1. Add the Firebase SPM package:

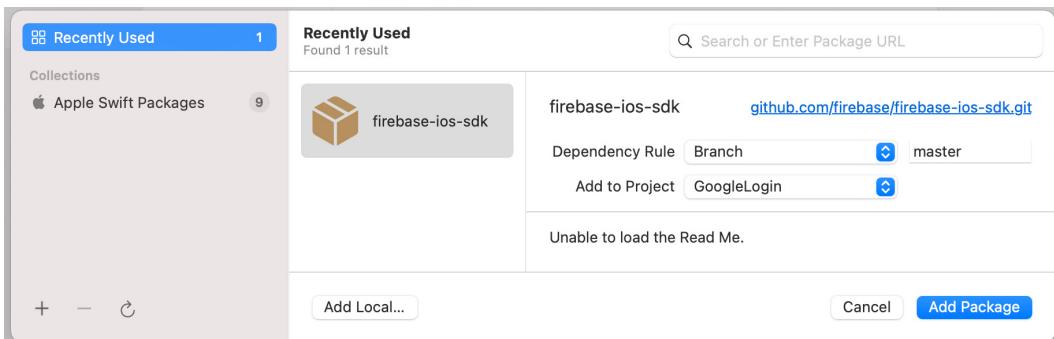


Figure 12.21 – Adding the Firebase main package

2. Then select the `FirebaseAuth` sub-package:

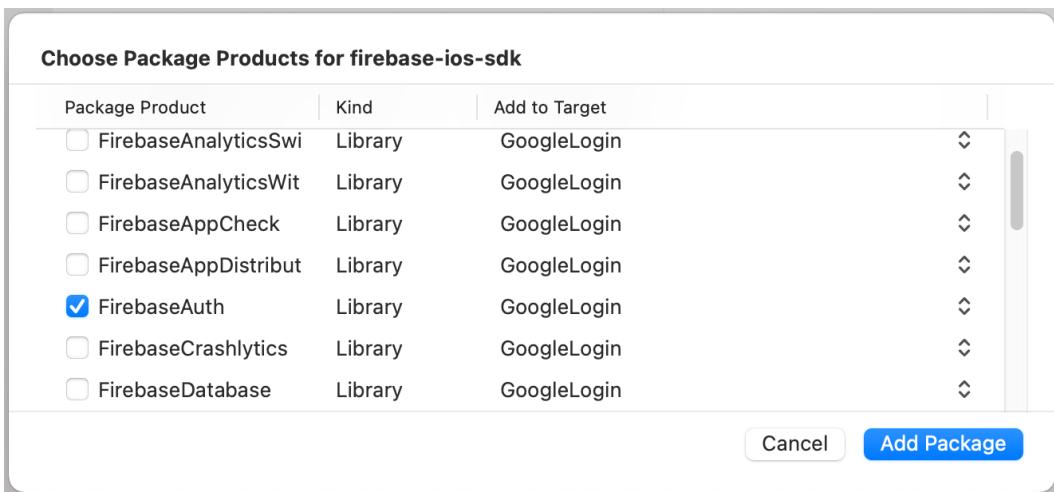


Figure 12.22 – Adding the FirebaseAuth sub-package

3. Add the GoogleSignIn SPM package, setting the URL to <https://github.com/google/GoogleSignIn-iOS>:

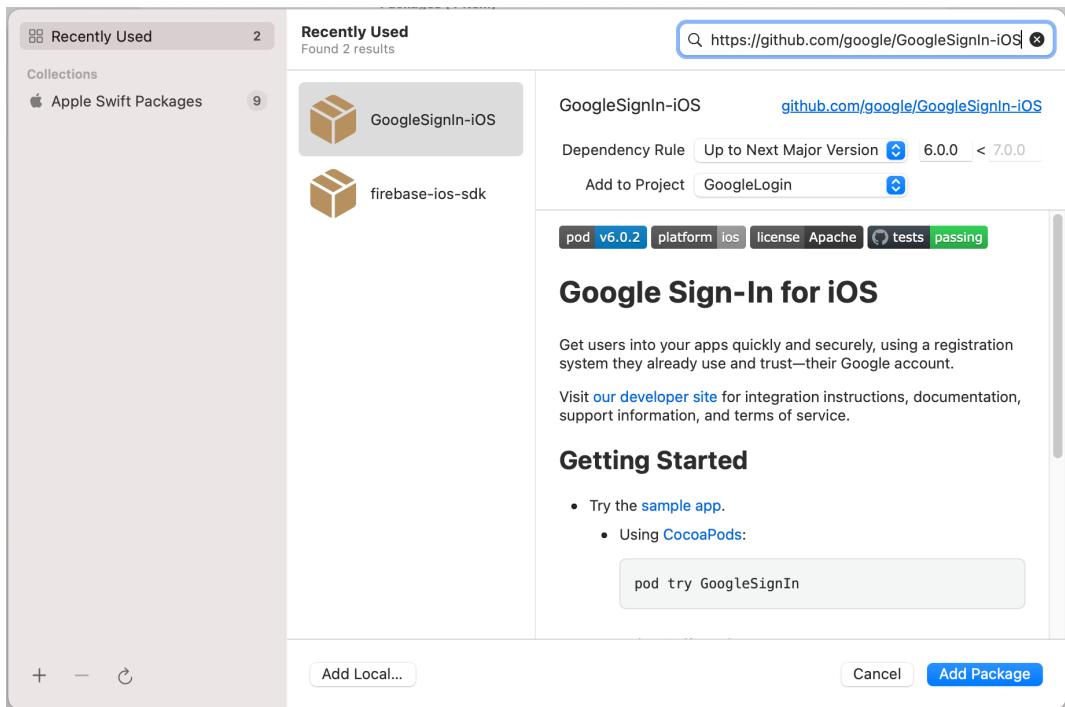


Figure 12.23 – Adding the GoogleSignIn SPM package

After configuring the project, create a `FirebaseGoogleSignInApp` in Firebase following the steps in the recipe *Integrating Firebase into a SwiftUI project*.

How to do it...

Firebase provides a library for using Google login. However, it is based on UIKit and sometimes, there are some hiccups since it expects a view controller to present the Google login action sheet. However, wrapping the provided component in a `UIViewRepresentable` allows us to implement it in a SwiftUI project.

Before you start coding, you must go to Firebase to configure the authentication providers for this app. Follow these steps to do so:

1. The first step is to select the **Authentication** section of the app in Firebase:

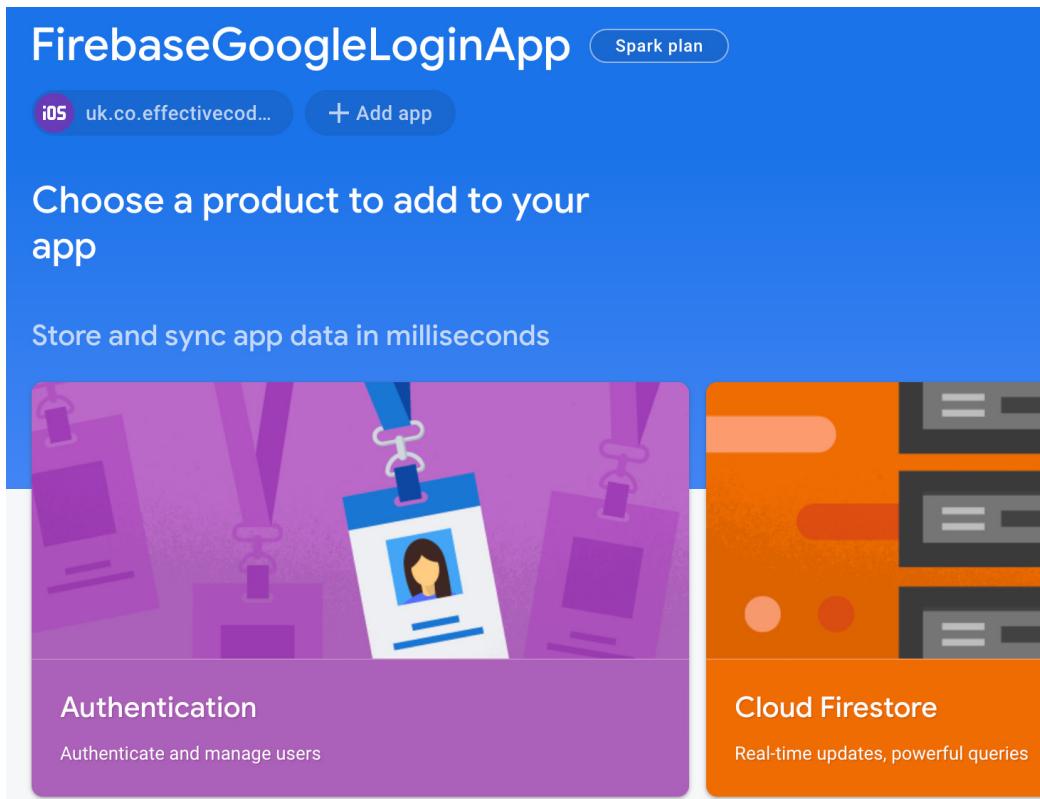


Figure 12.24 – Firebase Authentication section

2. In the **Authentication** section, click on the **Set up sign-in method** option:

A screenshot of the Firebase Authentication page. It shows a table with columns: Identifier, Providers, Created, Signed In, and User UID. A search bar at the top left says "Search by email address, phone number or user UID". An "Add user" button and a three-dot menu icon are on the right. Below the table, there's a large circular icon with a badge and the text "Authenticate and manage users from a variety of providers without server-side code". It includes links to "Learn more" and "View the docs". A blue "Set up sign-in method" button is at the bottom.

Figure 12.25 – Firebase project authentication page

3. From the list of the providers, **Enable** Google as the authentication provider, and **Save** this change:

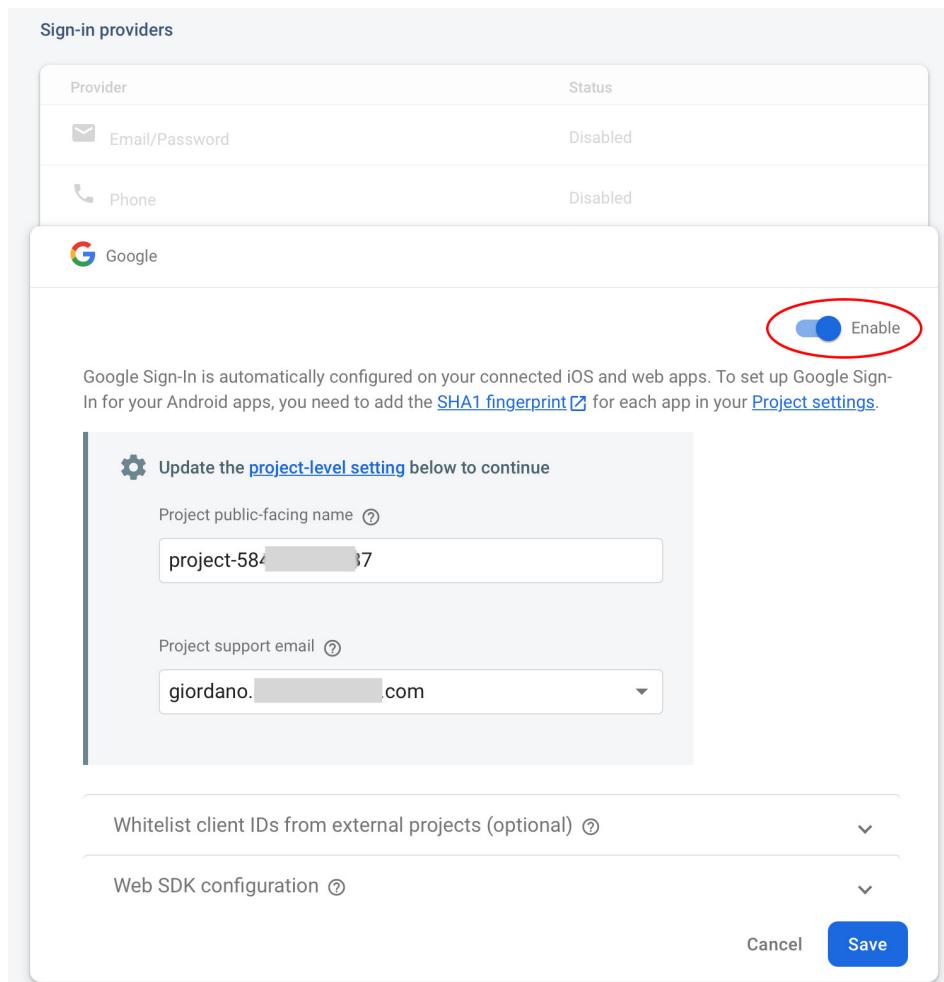


Figure 12.26 – Google sign-in provider

4. After adding the Google provider, you must go to the iOS configuration, where you must add the bundle ID of your app to create the `GoogleService-Info.plist` configuration file:

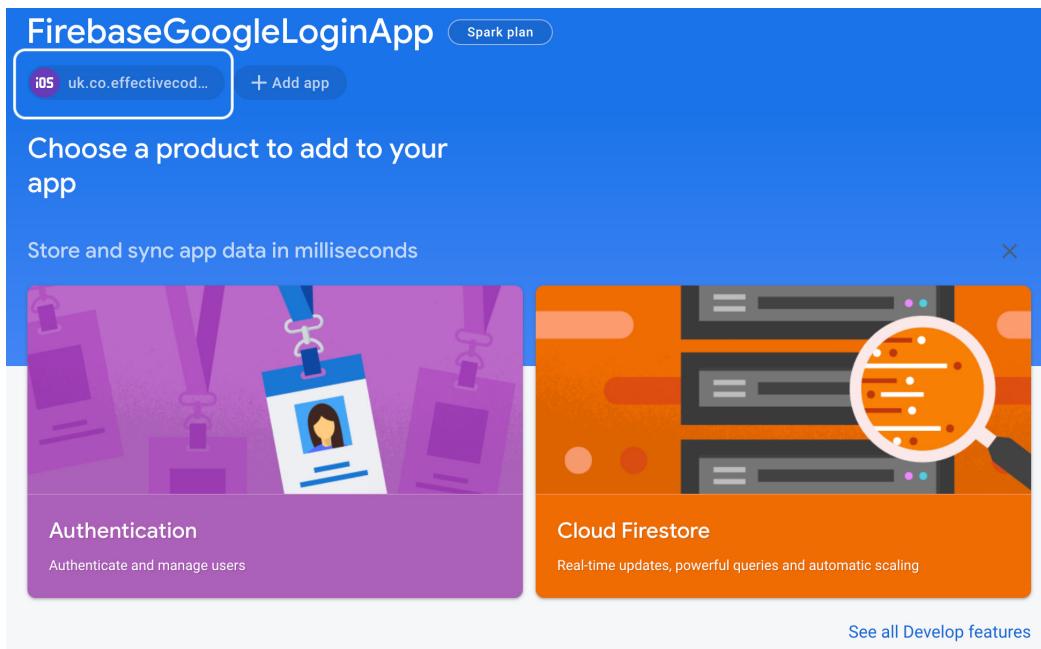


Figure 12.27 – Firebase project's iOS configuration

- From there, a new GoogleService-Info.plist file is generated. Download it and add it to the project:

The image contains two side-by-side screenshots. The left screenshot is from the Firebase console under the 'Add Firebase to your iOS app' section. It shows a 'Download config file' button and instructions to move the file into the Xcode project root. A blue arrow points from this screen to the right screenshot, which shows the 'GoogleService-Info.plist' file being dropped into the Xcode project's file hierarchy. The right screenshot shows the Xcode interface with the project structure expanded, showing files like 'AppDelegate.swift', 'ViewController.swift', 'Main.storyboard', 'Assets.xcassets', 'LaunchScreen.storyboard', and 'Info.plist'.

Figure 12.28 – Downloading the latest GoogleService-Info.plist file

6. In the `GoogleService-Info.plist` file, there is a value for the `REVERSED_CLIENT_ID` field that must be added to the project's configuration. Open `GoogleService-Info.plist` and copy the value to the clipboard.

Select the app from the **TARGETS** section, select the **Info** tab, and expand the **URL Types** section. Paste the copied value into the **URL Schemes** box:

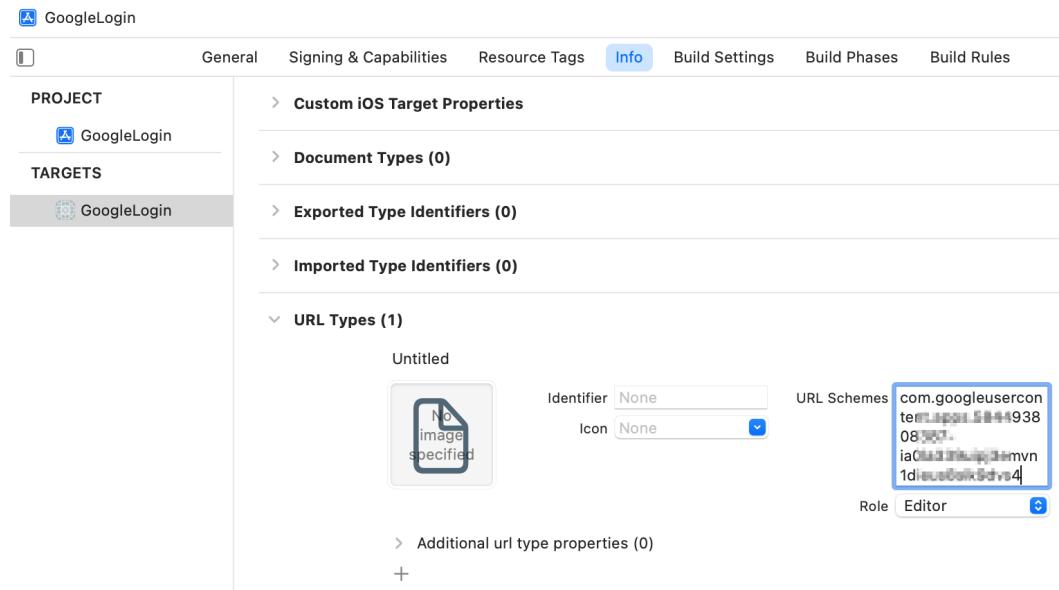


Figure 12.29 – Configuring URL Schemes

7. Move to the code now. Firstly, configure Firebase in the `GoogleLoginApp` main struct:

```
import SwiftUI
import Firebase

@main
struct GoogleLoginApp: App {
    init() {
        FirebaseApp.configure()
    }
    var body: some Scene {
        WindowGroup {
            ContentView()
        }
    }
}
```

```
    }
}
```

8. In the ContentView struct, create a wrapper around the **Google Sign In** button:

```
import Firebase
import GoogleSignIn
struct GoogleLogin: UIViewRepresentable {
    @Binding
    var signedIn: Bool
    @Binding
    var username: String
    @Binding
    var email: String
    func makeUIView(context: Context) -> UIView {
        Task {
            let (user, error) = await
                GIDSignIn.sharedInstance
                    .restorePreviousSignIn()
            await context.coordinator
                .sign(user: user,WithError: error)
        }
        let button = GIDSignInButton()
        button.addTarget(context.coordinator,
                        action: #selector(Coordinator
                            .action(sender:)),
                        for: .touchUpInside)
        return button
    }
    func updateUIView(_ uiView: UIView, context:
        Context) {
    }
}
```

9. Unfortunately, the Google Sign In button doesn't support the `async await` interfaces yet, but it provides a completion callback mechanism. Add the following extension to fill the gap and implement the two functions we need in `async await` way:

```
extension GIDSignIn {
    func restorePreviousSignIn() async ->
        (GIDGoogleUser?, Error?) {
        await withCheckedContinuation { continuation in
            GIDSignIn.sharedInstance
                .restorePreviousSignIn { user, error in
                    continuation.resume(with: .success((user,
                        error)))
                }
        }
    }

    func signIn(with configuration: GIDConfiguration,
               presenting viewController: UIViewController)
        async -> (GIDGoogleUser?, Error?) {
        await withCheckedContinuation { continuation in
            GIDSignIn.sharedInstance
                .signIn(with: configuration,
                        presenting: viewController)
            { user, error in
                continuation.resume(with: .success((user,
                    error)))
            }
        }
    }
}
```

10. Add the Coordinator class to the `GoogleLogin` button view:

```
struct GoogleLogin: UIViewRepresentable {
    @MainActor
    class Coordinator: NSObject {
        private let parent: GoogleLogin
```

```

        init(_ parent: GoogleLogin) {
            self.parent = parent
        }
        @objc func action(sender: UIControl) {
            guard let clientID = FirebaseApp
                .app()?.options.clientID,
            let rootViewController = UIApplication
                .currentRootViewController else {
                return
            }
            let configuration =
                GIDConfiguration(clientID:
                    clientID)
            Task {
                let (user, error) = await GIDSignIn
                    .sharedInstance.signIn(with: configuration,
                        presenting: rootViewController)
                await self.sign(user: user, withError: error)
            }
        }
    }
}

```

11. In the same Coordinator class, add a `sign()` function to complete the operation on Firebase and set the properties of the logged-in user in the view:

```

class Coordinator: NSObject {
//...
func sign(user: GIDGoogleUser?, withError error:
    Error?) async {
    guard let authentication =
        user?.authentication else { return }
    let credential = GoogleAuthProvider
        .credential(withIDToken:
            authentication.idToken!, accessToken:
            authentication.accessToken)
}

```

```
        guard let authResult = try? await Auth.auth()
                                .signIn(with: credential)
                else {
                    return
                }
                parent.signedIn = true
                if let username = authResult.user.displayName {
                {
                    parent.username = username
                }
                if let email = authResult.user.email {
                    parent.email = email
                }
            }
        }
```

12. Add a `makeCoordinator()` function to the `GoogleLogin` struct:

```
struct GoogleLogin: UIViewRepresentable {
    //...
    func makeCoordinator() -> Coordinator {
        Coordinator(self)
    }
    //...
}
```

13. As mentioned in the introduction of this recipe, the Google Sign In button requires a `ViewController` to be presented onto, and SwiftUI doesn't directly expose the current view controller. To overcome this, add the following extension to `UIApplication`:

```
extension UIApplication {
    static var currentRootViewController:
        UIViewController? {
            UIApplication.shared.connectedScenes
                .filter({$0.activationState ==
                    .foregroundActive})
                .map({$0 as? UIWindowScene})
```

```

        .compactMap({$0})
        .first?.windows
        .filter({$0.isKeyWindow})
        .first?
        .rootViewController
    }
}

```

14. We finished with the GoogleLogin code. Now create the GoogleLogout button:

```

struct GoogleLogout: UIViewRepresentable {
    @Binding
    var signedIn: Bool

    func makeUIView(context: Context) -> UIView {
        let button = UIButton(frame: .zero)
        button.setTitle("Logout", for: .normal)
        button.backgroundColor = UIColor.red
        button.addTarget(context.coordinator,
                        action:
                        #selector(Coordinator.onLogout),
                        for: .touchUpInside)
        return button
    }

    func updateUIView(_ uiView: UIView, context:
        Context) {
    }
}

```

15. Add the Coordinator to handle the button action:

```

struct GoogleLogout: UIViewRepresentable {
    //...
    func makeCoordinator() -> Coordinator {
        Coordinator(self)
    }
    class Coordinator: NSObject {

```

```
private let parent: GoogleLogout

init(_ parent: GoogleLogout) {
    self.parent = parent
}
@objc
func onLogout(button: UIButton) {
    GIDSignIn.sharedInstance.signOut()
    GIDSignIn.sharedInstance.disconnect()
    try? Auth.auth().signOut()
    parent.signedIn = false
}
}
```

16. Complete the app creating the `ContentView` struct, presenting either the `GoogleLogin` or `GoogleLogout` button depending on the user state:

```
struct ContentView: View {
    @State private var signedIn = false
    @State private var username: String = ""
    @State private var email: String = ""
    var body: some View {
        ZStack {
            Color.white
            if signedIn {
                VStack(spacing: 4) {
                    Text("Username: \(username)")
                        .foregroundColor(.black)
                    Text("Email: \(email)")
                        .foregroundColor(.black)
                    GoogleLogout(signedIn: $signedIn)
                        .frame(width: 200, height: 30,
                               alignment: .center)
                }
            } else {
                GoogleLogin(signedIn: $signedIn,
```

```
username: $username,  
email: $email)  
.frame(width: 200, height: 30,  
alignment: .center)  
}  
.edgesIgnoringSafeArea(.all)  
}  
}
```

Finally, we can view the full scenario in the app:

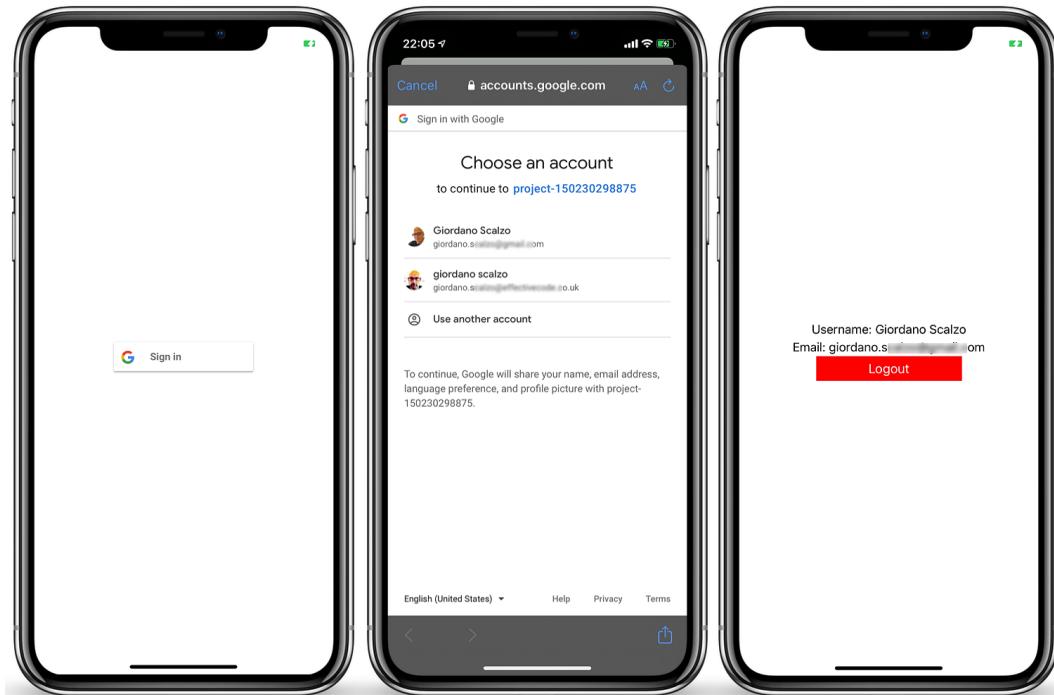


Figure 12.30 – Signing in with Google

How it works...

Another long recipe with quite a few configuration steps, but in the end, we created some components that can be useful in other apps too.

The authentication process is a sequence of two operations:

- Authenticate with Google
 - Authorize the credentials in Firebase

While the Firebase SDK fully embraces the `async await` pattern, that isn't the case for the Google Sign In framework yet. We filled that gap by writing an extension with two functions using the `withCheckedContinuation` function, which is the Swift-provided mechanism to transform a completion callback interface to an `async await` one.

With this extension, it is easier to write the code for the sequence of calls for doing the authentication and the authorization. Since the function where we make the two calls in sequence isn't marked as `async`, we must wrap the calls in a `Task { }` block, to allow the operations to terminate.

Following is the self-documenting code for the button tap:

```
@objc func action(sender: UIControl) {
    //...
    Task {
        let (user, error) = await GIDSignIn.sharedInstance
            .signIn(with: configuration,
                     presenting: rootViewController)
        await self.sign(user: user, withError: error)
    }
}
```

We use the same pattern when we create the Google login button, to fetch the current user if the user is already logged in:

```
func makeUIView(context: Context) -> UIView {
    Task {
        let (user, error) = await GIDSignIn.sharedInstance
            .restorePreviousSignIn()
        await context.coordinator.sign(user: user,
           WithError: error)
    }
    //...
}
```

One last thing to note – unfortunately, the Google Sign In framework isn't SwiftUI-friendly. It requires a `ViewController` to present the Google login page. In this recipe, we used a workaround to detect the current root `ViewController`:

```
extension UIApplication {  
    static var currentRootViewController: UIViewController? {  
        //...  
    }  
}
```

However, this solution isn't a reliable and future-proof way of providing a `ViewController` to present the Google Login, since Apple can change how the view hierarchy of an iOS app is created.

Until Firebase and Google release a pure SwiftUI solution, the safest implementation would be to have a `UIViewController` that encapsulates that login part so that we are sure that the `UIViewController` hierarchy is in place.

Implementing a distributed Notes app with Firebase and SwiftUI

One of the strongest features of Firebase is its distributed database capabilities. Since its first release, the possibility of having a distributed database in the cloud gave mobile developers a simple way of handling secure persistent storage in the cloud.

Firebase offers two types of databases:

- **Realtime Database**, which is the original one
- **Cloud Firestore**, which is a new and more powerful implementation

For this recipe, we are going to use Cloud Firestore. It not only allows apps to save data in the repository, but it also sends events when it is updated by another client, permitting your app to react to these changes in a seamless way. This asynchronous feature works very well with SwiftUI.

In this recipe, we are going to implement a simplified version of the default **Notes** app. In this app, we can save our notes in a Firestore collection, without being concerned with explicitly saving the notes or handling offline mode, since this is managed automatically by the Firebase SDK.

Getting ready

First, create a SwiftUI app called `FirebaseNotes` with Xcode.

After creating it, configure the project by following these steps:

1. Add the Firebase SPM package:

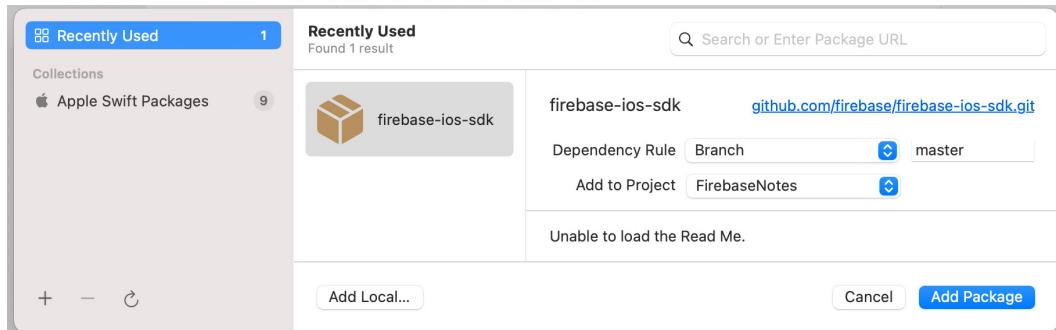


Figure 12.31 – Adding the main Firebase package

2. Then select the `FirebaseFirestore` and `FirebaseFirestoreSwift-Beta` sub-packages:

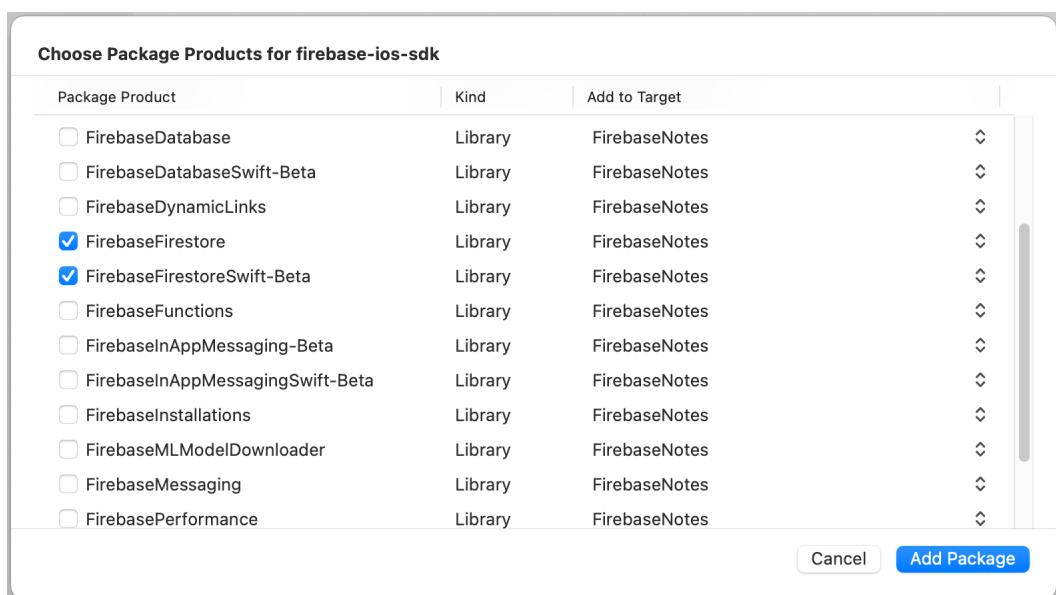


Figure 12.32 – Adding the Firebase sub-packages

After configuring the project, create a `FirebaseNotesApp` in Firebase by following the steps in the recipe *Integrating Firebase into a SwiftUI project*.

How to do it...

The app we are going to implement will have two main parts: the UI and the repository manager. The repository manager will be implemented around the Firestore SDK so that it can be managed by SwiftUI.

As usual, we must configure the project in Firebase based on our needs. Follow these steps to get started:

1. Go to the `FirebaseNotesApp` project in Firebase to create a database. Select **Firestore Database** from the left menu and click on **Create database**:

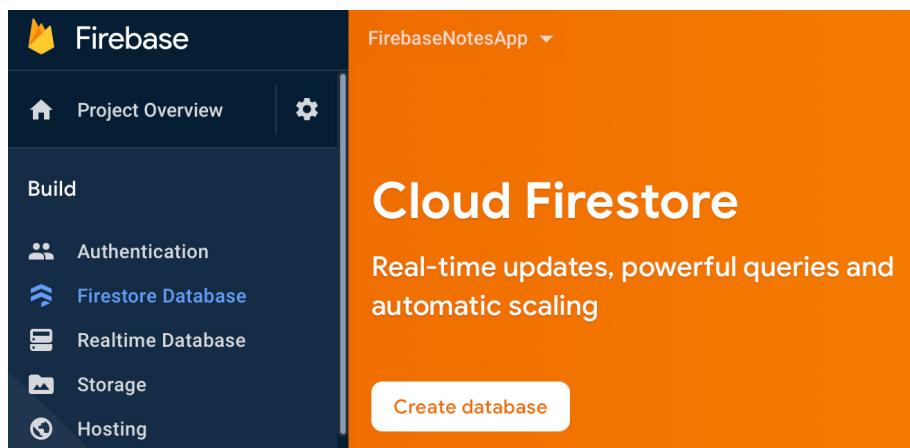


Figure 12.33 – Creating a database

2. Once you've created a new database, select the level of security you require. Exploring Firebase's security rules is beyond the scope of this book, so select the **Start in test mode** option, which is good enough for our goals:

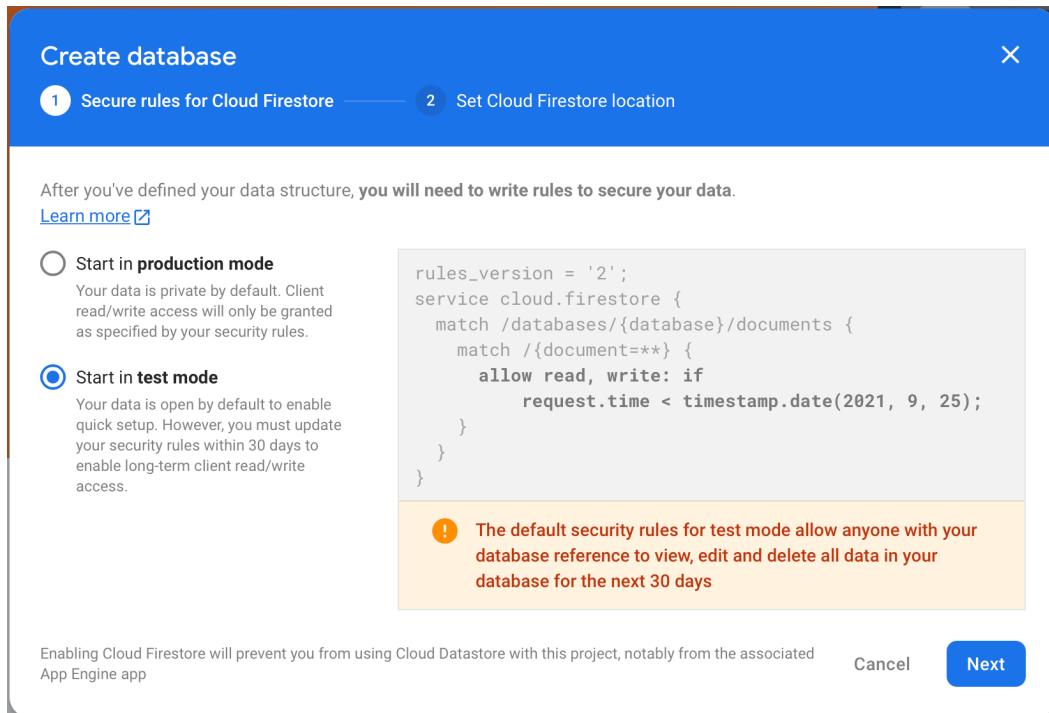


Figure 12.34 – Database security rules

3. Now, we must select the region where we're creating the database. Select the one closest to your area:

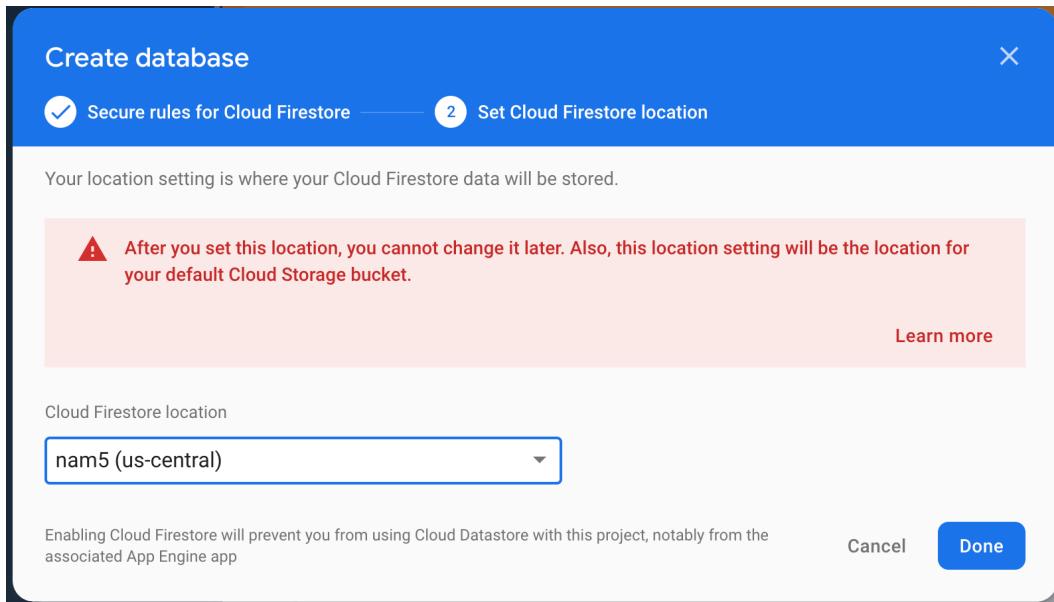


Figure 12.35 – Cloud Firestore location

4. By doing this, we have created the database. Verify this in the Firebase console:

The screenshot shows the 'Database' overview in the Firebase console. The top navigation bar has tabs for 'Data', 'Rules', 'Indexes', and 'Usage', with 'Data' being the active tab. Below the tabs, there is a table with one row. The first column contains a house icon and a plus sign, and the second column contains the database name 'fir-notesapp-4534b'. At the bottom of the table is a blue button labeled '+ Start collection'.

Figure 12.36 – Database overview on Firebase

5. Move to the code now. Firstly, configure Firebase in the `FirebaseNotesApp` main view:

```
import SwiftUI
import Firebase
@main
struct FirebaseNotesApp: App {
    init() {
        FirebaseApp.configure()
    }
    var body: some Scene {
        WindowGroup {
            ContentView()
        }
    }
}
```

6. The Note we are going to implement is a simple `struct` with a `title`, a `body`, and its creation date visible. Since we'll show a list of notes in a `List` View, make it `Identifiable`. Add the following code in a new file in the project:

```
struct Note: Identifiable, Codable {
    let id: String
    let title: String
    let date: Date
    let body: String
}
```

7. Create a `NotesRepository` struct that can add a new note and that fetches the list of the created notes:

```
struct NotesRepository {
    func newNote(title: String,
                date: Date,
                body: String) async -> [Note] {
    }

    func fetchNotes() async -> [Note] {
```

```
    }  
}
```

8. Add a reference to a collection called "notes" in the Firestore instance:

```
struct NotesRepository {  
    private let dbCollection = Firestore.firestore()  
        .collection("notes")  
    //...  
}
```

9. Implement the `fetchNotes()` function:

```
func fetchNotes() async -> [Note] {  
    guard let snapshot = try? await dbCollection  
        .getDocuments() else {  
        return []  
    }  
  
    let notes: [Note] = snapshot.documents  
        .compactMap { document in  
            try? document.data(as: Note.self)  
        }  
    return notes.sorted {  
        $0.date < $1.date  
    }  
}
```

10. The `newNote()` function appends a new document to the collection and then refreshes the list of the notes:

```
func newNote(title: String,  
            date: Date,  
            body: String) async -> [Note] {  
    let note = Note(id: UUID().uuidString,  
                    title: title,  
                    date: date,  
                    body: body)  
    _ = try? dbCollection.addDocument(from: note)
```

```
        return await fetchNotes()
    }
```

11. Now that we have finished the repository, let's concentrate our efforts on visualizing the notes. We are going to implement three views: ContentView to show the notes, NewNote to create a new one, and ShowNote to visualize it. Let's start with ContentView, where we'll add the properties to watch and a convenient DateFormatter instance:

```
struct ContentView: View {
    static let taskDateFormat: DateFormatter = {
        let formatter = DateFormatter()
        formatter.dateStyle = .long
        return formatter
    }()
    private var repository: NotesRepository =
        NotesRepository()
    @State
    var isNewNotePresented = false
    @State
    var notes: [Note] = []
    //...
}
```

12. In the body, add a List to present the notes. If the user clicks on the row, they will be redirected to a new view that shows the selected note:

```
var body: some View {
    NavigationView {
        List(notes) { note in
            NavigationLink(destination:
                ShowNote(note: note)) {
                VStack(alignment: .leading) {
                    Text(note.title)
                        .font(.headline)
                        .fontWeight(.bold)
                    Text("\\"(note.date,
                        formatter: Self.taskDateFormat)")
                        .font(.subheadline)
                }
            }
        }
    }
}
```

```
        }
    }
}
}
```

13. To create a new note, add a "plus" button to the navigation bar that toggles the `isNewNotePresented` `@State` variable:

```
var body: some View {
    NavigationView {
        List(notes) { note in
            //...
        }
        .navigationBarTitle("FireNotes", displayMode:
            .inline)
        .navigationBarItems(trailing:
            Button {
                isNewNotePresented.toggle()
            } label: {
                Image(systemName: "plus")
                    .font(.headline)
            }
        )
    }
}
```

14. The `NewNote` view is presented using the `.sheet()` modifier:

```
var body: some View {
    NavigationView {
        //...
        .navigationBarItems(trailing:
            //...
        )
        .sheet(isPresented: $isNewNotePresented) {
            NewNote(isNewNotePresented:
                $isNewNotePresented,
```

```
        notes: $notes,
        repository: repository)
    }
}
```

15. Finally, add a `.task()` modifier to fetch the already created notes:

```
var body: some View {
    NavigationView {
        //...
        .sheet(isPresented: $isNewNotePresented) {
            //...
        }
        .task {
            notes = await repository.fetchNotes()
        }
    }
}
```

16. Now, let's implement a view for adding a new note. It must be possible to add a title, a body, and save it. Follow the same pattern we followed for `ContentView`, presenting the components in a `NavigationView` with a checkmark bar button item to indicate that we are happy with our changes. We only make the `title` mandatory, so the `done` button will only be enabled after we fill in the `title` component. For the body of the note, we are going to use the `TextEditor` component, which supports multi-line text, just like the `UITextView` class does in UIKit. Start by defining the interface of the `NewNote` view:

```
struct NewNote: View {
    @State
    private var title: String = ""
    @State
    private var bodyText: String = ""
    @Binding
    var isNewNotePresented: Bool
    @Binding
    var notes: [Note]
```

```
var repository: NotesRepository  
//...  
}
```

17. In body, present a `VStack` with a `title` and `body` that's encapsulated in a `NavigationView`:

```
var body: some View {  
    NavigationView {  
        VStack(spacing: 12) {  
            TextField("Title", text: $title)  
                .padding(4)  
                .border(.gray)  
            TextEditor(text: $bodyText)  
                .border(.gray)  
        }  
    }  
}
```

18. Now, add some padding and a title to the View:

```
var body: some View {  
    NavigationView {  
        VStack(spacing: 12) {  
            //...  
        }  
        .padding(32)  
        .navigationBarTitle("New Note", displayMode:  
            .inline)  
    }  
}
```

19. Finally, add the *done* button in the form of a checkmark:

```
var body: some View {
    NavigationView {
        //...
        .navigationBarTitle("New Note", displayMode:
            .inline)
        .navigationBarItems(trailing:
            Button {
                Task {
                    notes = await repository
                        .newNote(title: title,
                                date: Date(),
                                body:
                                bodyText)
                }
                isNewNotePresented.toggle()
            } label: {
                Image(systemName:
                    "checkmark")
                    .font(.headline)
            }.disabled(title.isEmpty)
        )
    }
}
```

Note that the enabled/disabled state is determined by the state of the `TextField` component of `title`.

20. Finally, move to implement the `ShowNote` view, which is a simplified version of the `NewNote` view. Create it by adding the following code:

```
struct ShowNote: View {
    let note: Note
    var body: some View {
        VStack(spacing: 12) {
            Text(note.title)
                .font(.headline)
                .fontWeight(.bold)
```

```
        ReadonlyTextEditor(text: note.body)
            .border(.gray)
    }
    .padding(32)
}
}
```

21. When viewing a note, its content shouldn't be editable. Since the `TextEditor` component doesn't support read-only mode, we must implement a simple `UIViewRepresentable` around a read-only `UITextView`:

```
struct ReadonlyTextEditor: UIViewRepresentable {
    var text: String
    func makeUIView(context: Context) -> UITextView {
        let view = UITextView()
        view.isScrollEnabled = true
        view.setEditable = false
        view.isUserInteractionEnabled = true
        return view
    }
    func updateUIView(_ uiView: UITextView,
                      context: Context) {
        uiView.text = text
    }
}
```

Done! The app is now complete. We can try it out by adding recipes to it, as shown in the following screenshot:

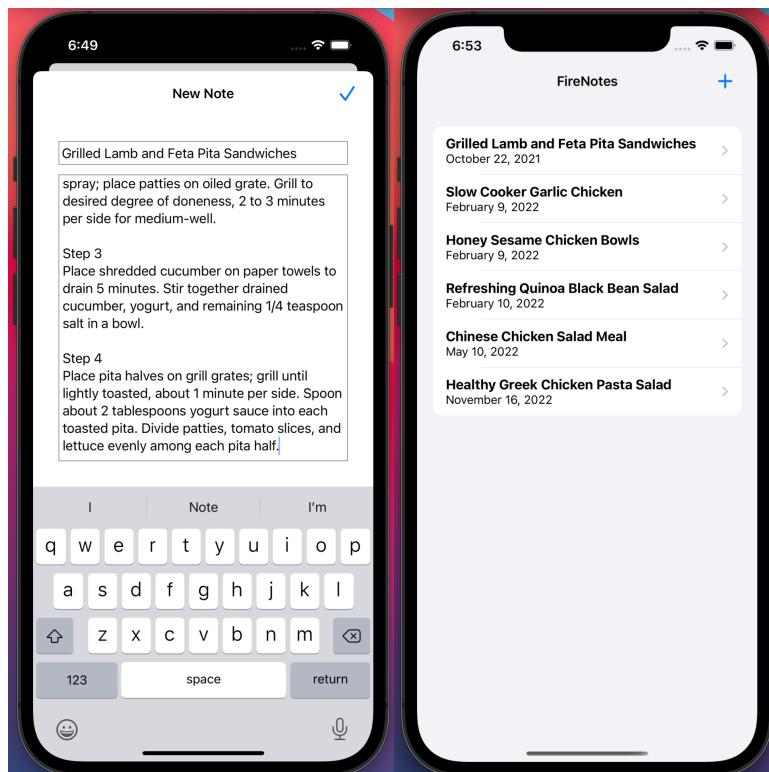


Figure 12.37 – Notes app in the cloud

How it works...

The first thing to notice is the unbalanced amount of code needed to implement this recipe; the code we needed to manage the database is probably about one-third of the code needed for the UI!

Also, note that the bridging between the Firebase library and SwiftUI is pretty natural and that it works the way it should. Firestore is a so-called document store, where the data is saved in an informal way in the form of a collection of dictionaries.

If you have minimal experience with relational databases, then you should know that the first thing you should do is model the data and define a schema for the database. If, in the future, our data needs an extension, we need to define a new schema, provide a migration script, and so on. In the case of Firestore, after creating a project in Firebase, you are ready to start creating documents, updating them, and so on.

We didn't use them here, but the Firestore instance also provides sophisticated functions for performing queries, such as for returning all the documents that contain a particular word, or those that have been created on a particular day.

Another interesting feature is the notification mechanism, where the app is notified when the repository changes – maybe because another device has changed it or because the repository has been changed directly from the Firebase console.

Firebase provides a web console for viewing and editing a database, providing a simple but powerful tool that we can use to administer our databases.

The following screenshot shows what the database that we created in this recipe looks like:

The screenshot shows the Firebase Firestore web console interface. At the top, there's a navigation bar with 'fir-notesapp-4534b' and tabs for 'Data', 'Rules', 'Indexes', and 'Usage'. Below the navigation bar, the 'Database' section is selected, and it shows a 'Cloud Firestore' dropdown. The main area displays a hierarchical view of a 'notes' collection. On the left, under 'fir-notesapp-4534b / notes', there are several document IDs: 1H5B8ohPasp909YF8duq, 49YHVq08f6jPhjTbYXim, 81hz4BqchO31DZLCsjHJ, SyHAV7ToksaJUiKbvGDr, mM9ZKSV75J4dL0jUfh5Z, p2pIuv2MLC71gnmEWIzm, and zo1vqY2UKhnRdShlTKoG. The document 'zo1vqY2UKhnRdShlTKoG' is currently selected, and its details are shown on the right. The document contains fields: 'body' (with a detailed recipe for Grilled Lamb and Feta Pita Sandwiches), 'date' (set to '9 April 2020 at 14:25:26 UTC+1'), and 'title' ('Grilled Lamb and Feta Pita Sandwiches'). There are also buttons for 'Start collection' and 'Add field'.

Figure 12.38 – Firestore's web console

The Firestore SDK provides asynchronous functions that fully embrace the `async await` model, hence the integration with SwiftUI is very natural.

The `newNote()` function must return the refreshed notes after adding the new one as a document to the "notes" collection. Being both `addDocument()` and `fetchNotes()` `async`, we can create a sequence of the operations by just writing them in two sequential lines.

There's more...

As we have seen, using Firestore is really easy and fits well with the SwiftUI model. From here, we can add more features.

An obvious one would be to add the possibility of editing a note. Firestore has a powerful mechanism that notifies the code that uses it about when the repository has changed.

At the moment, we are only refreshing the notes at startup or when we add or delete a new note.

I encourage you to explore the notification capabilities of Firestore and add a reload when someone else changes the repository. You can test this by launching the app in two different simulators.

I'm pretty sure you'll be amazed by the simplicity of adding these features, thus making our basic notes app really close to Apple's official Notes app.

13

Handling Core Data in SwiftUI

Core Data is one of the most essential Apple frameworks in the iOS and macOS ecosystem. Core Data provides persistence, which means it can save data outside the app's memory, and the data you save can be retrieved after you restart your app. Given its importance, it's not a surprise that Apple has implemented some extensions for Core Data to make it work nicely with SwiftUI.

In Core Data language, a stored object is an instance of `NSManagedObject`, which conforms to the `ObservableObject` protocol so that it can be observed directly by a SwiftUI's view.

Also, `NSManagedObjectContext` is injected into the environment of the View's hierarchy so that SwiftUI's View can access it to read and change its managed objects.

A very common feature of Core Data is that you can fetch objects from its repository. For this purpose, SwiftUI provides the `@FetchRequest` property wrapper, which can be used in a view to load data.

When you create a new project in Xcode, you can check the **Use Core Data** checkbox to make Xcode generate the code needed to inject the Core Data stack into your code. In the first recipe of this chapter, we'll learn how to do this manually to understand where Core Data fits into an iOS app architecture. In the remaining recipes, we'll learn how to perform the basic persistence operations; that is, creating, reading, and deleting persistent objects in Core Data and SwiftUI.

In this chapter, we are going to learn how to integrate the default Apple way of performing persistence with Core Data in SwiftUI.

We'll explore the basics of Core Data in SwiftUI by covering the following recipes:

- Integrating Core Data with SwiftUI
- Showing Core Data objects with `@FetchRequest`
- Adding Core Data objects from a SwiftUI view
- Filtering Core Data requests using a predicate
- Deleting Core Data objects from a SwiftUI view
- Presenting data in a sectioned list with `@SectionedFetchRequest`

Technical requirements

The code in this chapter is based on Xcode 13. The minimum iOS version required is iOS 15.

You can find the code for this chapter in this book's GitHub repository at <https://github.com/PacktPublishing/SwiftUI-Cookbook-2nd-Edition/tree/main/Chapter13-Handling-Core-Data-in-SwiftUI>.

Integrating Core Data with SwiftUI

Over the years, you may have found different ways of using Core Data in your apps from an architectural point of view. For example, Apple, pre-iOS14, provided Xcode templates that created the Core Data containers in `AppDelegate`. Other developers prefer to wrap Core Data inside manager classes, abstracting Core Data entirely, while encapsulating the whole Core Data Stack and managed objects in a module, so that it's easy to move to another solution, such as **Realm**, if needed.

SwiftUI's integration, however, points firmly in one direction: create the container when the app starts, inject it into `Environment`, and then use it to fetch data or make changes.

When building a new app with Xcode, you can check the **Use Core Data** checkbox so that Xcode creates a template that injects the Core Data stack in the most efficient way possible.

Although the template provided by Xcode is quite complete and powerful, it is unnecessarily complicated. In this recipe, we'll introduce Core Data manually in a SwiftUI project, along with a minimum set of functionalities, so that you can build more complex interactions with Core Data without being overwhelmed by its complex initial code.

Getting ready

Let's create a SwiftUI app called `SwiftUICoreDataStack`, ensuring that we leave the **Use Core Data** option unchecked:

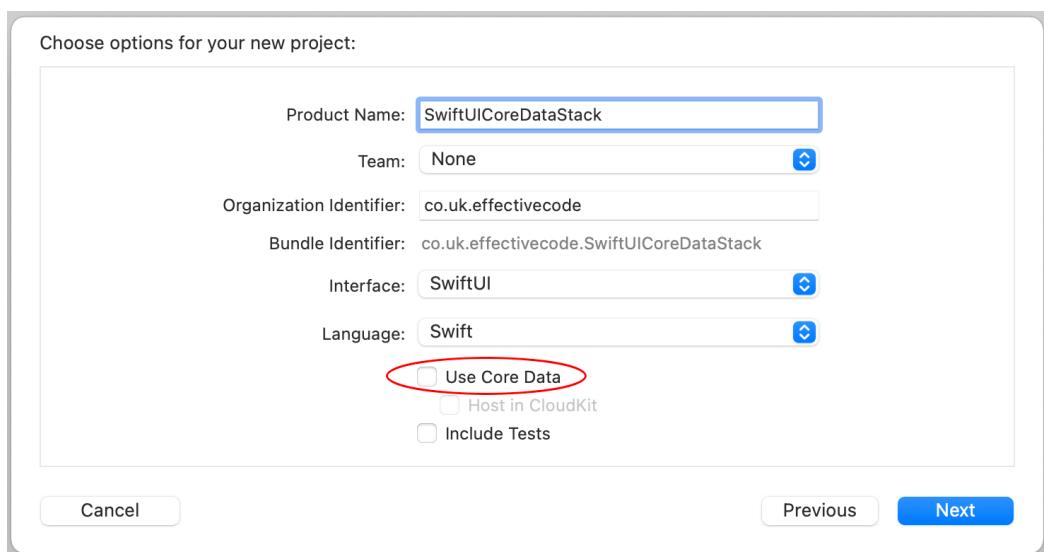


Figure 13.1 – A SwiftUI app without Core Data

How to do it...

In this recipe, we are going to wrap the Core Data stack into a class called `CoreDataStack` (what a surprise!) and instantiate it in the `@main` struct, which will handle its life cycle.

Let's get started:

1. First, add a **Core Data** model called `ContactsModel`. We should add some entities to the model, but for the moment, let's leave it empty:

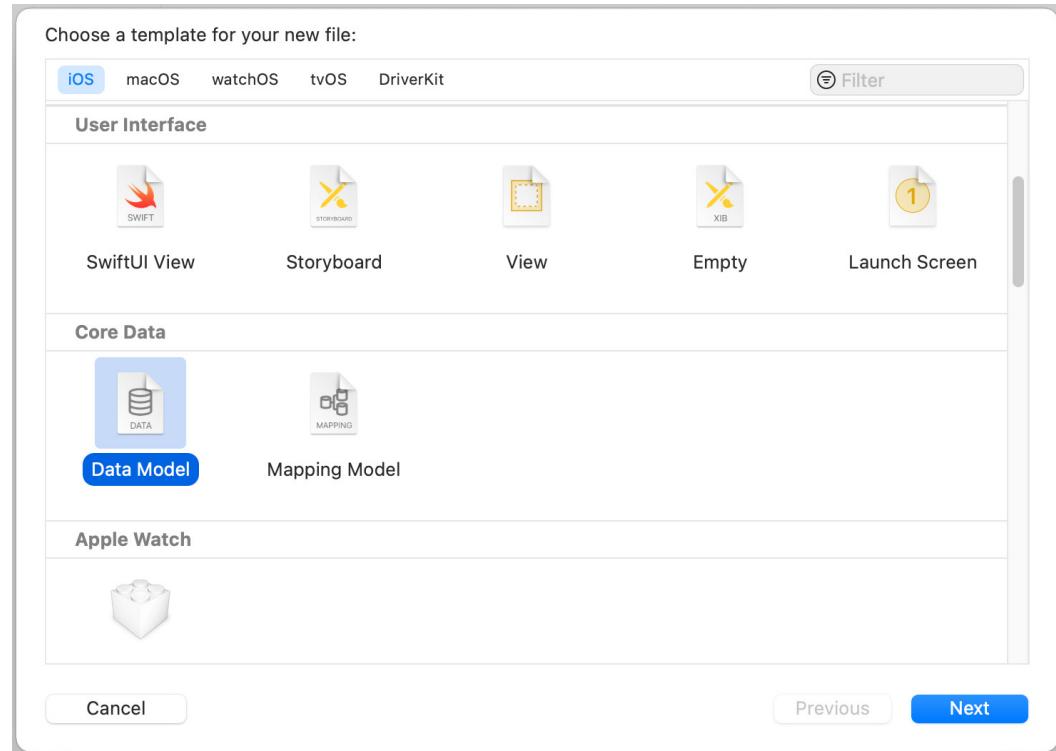


Figure 13.2 – Adding a Core Data model

2. Now, implement a class that will wrap the Core Data framework. This class will be called `CoreDataStack`. The name of the model is passed as a parameter. This class creates and holds the container:

```
import CoreData

class CoreDataStack {
    private let persistentContainer:
        NSPersistentContainer
    var managedObjectContext: NSManagedObjectContext {
        persistentContainer.viewContext
    }
}
```

```
init(modelName: String) {
    persistentContainer = {
        let container = NSPersistentContainer(
            name: modelName)
        container
            .loadPersistentStores { description,
                error in
                if let error = error {
                    print(error)
                }
            }
        return container
    }()
}
```

3. Add another function to this class that saves the changed objects:

```
class CoreDataStack {
    //...
    func save () {
        guard managedObjectContext
            .hasChanges else { return }
        do {
            try managedObjectContext.save()
        } catch {
            print(error)
        }
    }
}
```

4. Create a unique instance of CoreDataStack in the @main struct by injecting the managedObjectContext instance into Environment:

```
@main
struct SwiftUICoreDataStackApp: App {
    private let coreDataStack = CoreDataStack(
```

```
        modelName: "ContactsModel")  
  
    var body: some Scene {  
        WindowGroup {  
            ContentView()  
                .environment(\.managedObjectContext,  
                           coreDataStack.managedObjectContext)  
        }  
    }  
}
```

5. iOS provides a mechanism to detect when an app goes into the background via an Environment variable called `scenePhase`. We will add a listener function to detect when the variable changes. When this happens, we will save the new or modified Core Data objects. Add the following code to do this:

```
@main  
struct SwiftUICoreDataStackApp: App {  
    @Environment(\.scenePhase) var scenePhase  
  
    var body: some Scene {  
        WindowGroup {  
            //...  
        }  
        .onChange(of: scenePhase) { _ in  
            coreDataStack.save()  
        }  
    }  
}
```

6. Finally, access `managedObjectContext` from the `ContentView` view by adding the following code:

```
struct ContentView: View {  
    @Environment(\.managedObjectContext)  
    var managedObjectContext  
  
    var body: some View {
```

```
Text ("\\(managedObjectContext)")  
}  
}
```

By running the app, we can see that `managedObjectContext` is valid:

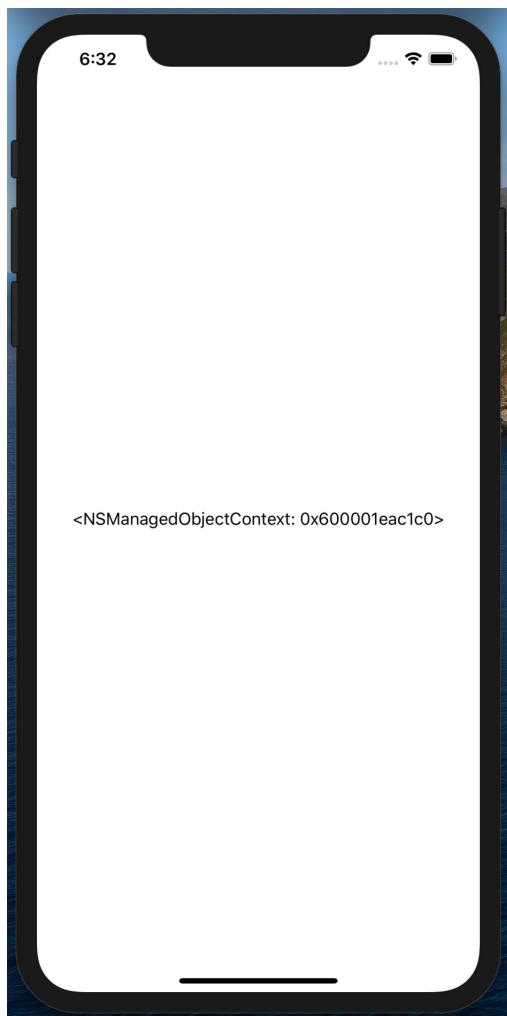


Figure 13.3 – The shared instance of `NSManagedObjectContext` in SwiftUI

How it works...

In this recipe, we learned how to create a container for a Core Data database. In a SwiftUI architecture, Apple gives us a strong indication of how to use a Core Data stack in the app; that is, by passing `NSManagedObjectContext` into `Environment`. This allows the predefined Core Data property wrapper, such as `@FetchRequest`, to work out of the box by accessing the storage without passing it during the fetch. We will look at this in more detail in the *Showing Core Data objects with `@FetchRequest`* recipe.

Note that we could have added some code to persist `managedObjectContext` when it is added or changed, but this would not be very efficient. Our `CoreDataStack` implementation holds the change in memory and detects when the app changes phases; for example, going to the background.

This can be achieved by subscribing to the changes of the `Environment` property called `scenePhase`. The value of `scenePhase` changes when the app goes into the background or comes back to the foreground, triggering a `managedObjectContext` check and persisting it if it has any modifications.

There's more...

To understand the code shown in this recipe, I encourage you to create another SwiftUI Core Data project. In this new project, add some support for Core Data while creating the project so that Xcode will use its template. By doing this, you can compare that code with our `CoreDataStack` implementation to see their similarities and differences.

Showing Core Data objects with `@FetchRequest`

The most critical feature of persistent storage is its fetching capability. We could prebuild a Core Data database and bundle it with our app, which would just read and present the data. An example of this kind of app could be a catalog for a clothes shop, which contains the clothes for the current season. When the new fashion season arrives, a new app with a new database is created and released.

Given the importance of having this skill, Apple has added a powerful property wrapper to make fetching data from a repository almost trivial. In this recipe, we'll create a simple contact list visualizer in SwiftUI. The objects in the repository will be added the first time we run the app, and `ContentView` will present the contacts in a list view.

Getting ready

Let's create a SwiftUI app called `FetchContact`.

How to do it...

In this recipe, we will add a bunch of hardcoded contacts to the storage at app startup, ensuring that we only do this the first time the app launches.

In the `ContentView` struct, we are going to use the `@FetchRequest` property wrapper to automatically populate a list of contacts to be fetched from the Core Data storage.

Let's get started:

1. Add a `CoreDataStack` class to hold `managedObjectContext` and save the new or modified Core Data objects that still reside in memory (if there are any):

```
import CoreData

class CoreDataStack {
    private let persistentContainer:
        NSPersistentContainer
    var managedObjectContext: NSManagedObjectContext {
        persistentContainer.viewContext
    }

    init(modelName: String) {
        persistentContainer = {
            let container = NSPersistentContainer(
                name: modelName)
            container
                .loadPersistentStores { description,
                    error in
                    if let error = error {
                        print(error)
                    }
                }
            return container
        }()
    }
}
```

```
    }

    func save () {
        guard managedObjectContext.hasChanges else {
            return }
        do {
            try managedObjectContext.save()
        } catch {
            print(error)
        }
    }
}
```

2. In the @main struct, inject the managedObjectContext instance into Environment and save it when the phase changes:

```
@main
struct FetchContactsApp: App {
    private let coreDataStack = CoreDataStack(
        modelName: "ContactsModel")
    @Environment(\.scenePhase) var scenePhase

    var body: some Scene {
        WindowGroup {
            ContentView()
                .environment(\.managedObjectContext,
                            coreDataStack.
                    managedObjectContext)
                .onChange(of: scenePhase) { _ in
                    coreDataStack.save()
                }
        }
    }
}
```

3. Add a Core Data model called `ContactsModel`. Open the model file and do the following:
 - a. Add a new entity called `Contact` (1).
 - b. Add three string attributes called `firstName`, `lastName`, and `phoneNumber` (2).
 - c. Check that **Codegen** is set to **Class Definition** (3).

The following screenshot shows the steps you must follow to update the model:

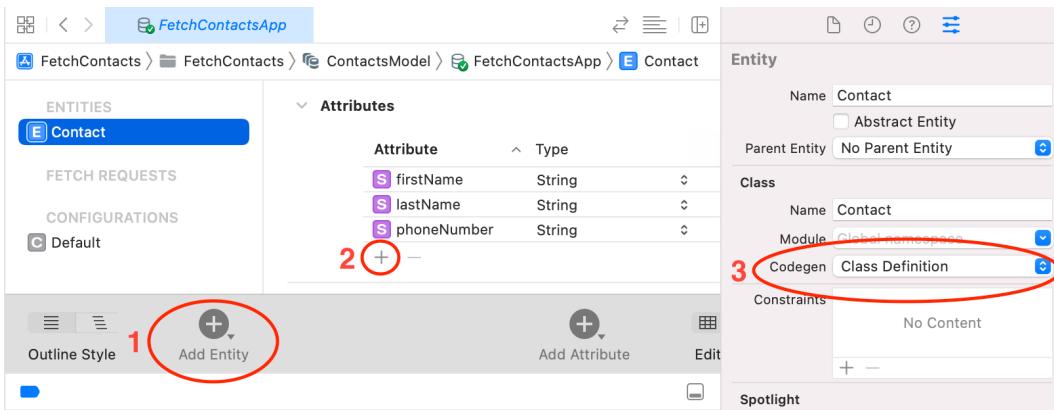


Figure 13.4 – Creating the Core Data model

4. Setting **Codegen** to **Class Definition** means that Xcode creates a convenience class that will manage the entities; in our case, it creates a `Contact` class. Add an `insertContact()` function to the `CoreDataStack` class:

```
class CoreDataStack {
    //...
    func insertContact(firstName: String,
                       lastName: String,
                       phoneNumber: String) {
        let contact = Contact(context:
            managedObjectContext)
        contact.firstName = firstName
        contact.lastName = lastName
        contact.phoneNumber = phoneNumber
    }
}
```

5. Now, let's create a function that will insert a bunch of contacts, but only if this is the first time we're running the app; otherwise, the repository will be filled with duplicates. To check that this is the first time we're running the app, we need to set a flag in `UserDefault`s. For this purpose, we've created a `Contacts.swift` file, which you can find in this book's GitHub repository at <https://github.com/PacktPublishing/SwiftUI-Cookbook-2nd-Edition-/blob/main/Chapter13-Handling-Core-Data-in-SwiftUI/02-Showing-Core-Data-objects-with-%40FetchRequest/FetchContacts/FetchContacts.swift>. It contains the following code:

```
func addContacts(to coreDataStack: CoreDataStack) {  
    guard UserDefaults  
        .standard.bool(forKey: "alreadyRun") ==  
    false else {  
        return  
    }  
    UserDefaults.standard.set(true, forKey:  
        "alreadyRun")  
  
    [ ("Daenerys", lastName: "Targaryen",  
        "02079460803"),  
        ("Bran", lastName: "Stark", "02079460071"),  
        //... the rest is in the GitHub repo  
        ("Sansa", lastName: "Stark", "02890180764")]  
        .forEach { (firstName, lastName, phoneNumber)  
            in  
                coreDataStack.insertContact(firstName:  
                    firstName,  
                    lastName: lastName,  
                    phoneNumber: phoneNumber)  
        }  
}
```

6. In the @main struct, add an .onAppear modifier that calls this function:

```
@main
struct FetchContactsApp: App {
    var body: some Scene {
        WindowGroup {
            ContentView()
            //...
            .onChange(of: scenePhase) { _ in
                coreDataStack.save()
            }
            .onAppear {
                addContacts(to: coreDataStack)
            }
        }
    }
}
```

7. Now, let's move on to the code for visualizing the contacts. Create a ContactView struct:

```
struct ContactView: View {
    let contact: Contact
    var body: some View {
        HStack {
            Text(contact.firstName ?? "-")
            Text(contact.lastName ?? "-")
            Spacer()
            Text(contact.phoneNumber ?? "-")
        }
    }
}
```

8. In the `ContentView` struct, add the list of contacts via the `@FetchRequest` property wrapper:

```
struct ContentView: View {  
    @FetchRequest(  
        sortDescriptors: [  
            NSSortDescriptor(keyPath:  
                \Contact.lastName,  
                ascending: true),  
            NSSortDescriptor(keyPath:  
                \Contact.firstName,  
                ascending: true),  
        ]  
    )  
    var contacts: FetchedResults<Contact>  
}
```

9. Finally, `body` will simply render the list of contacts. Add the following code:

```
struct ContentView: View {  
    //...  
    var body: some View {  
        List(contacts, id: \.self) {  
            ContactView(contact: $0)  
        }  
        .listStyle(.plain)  
    }  
}
```

Upon running the app, we'll see our list of contacts:

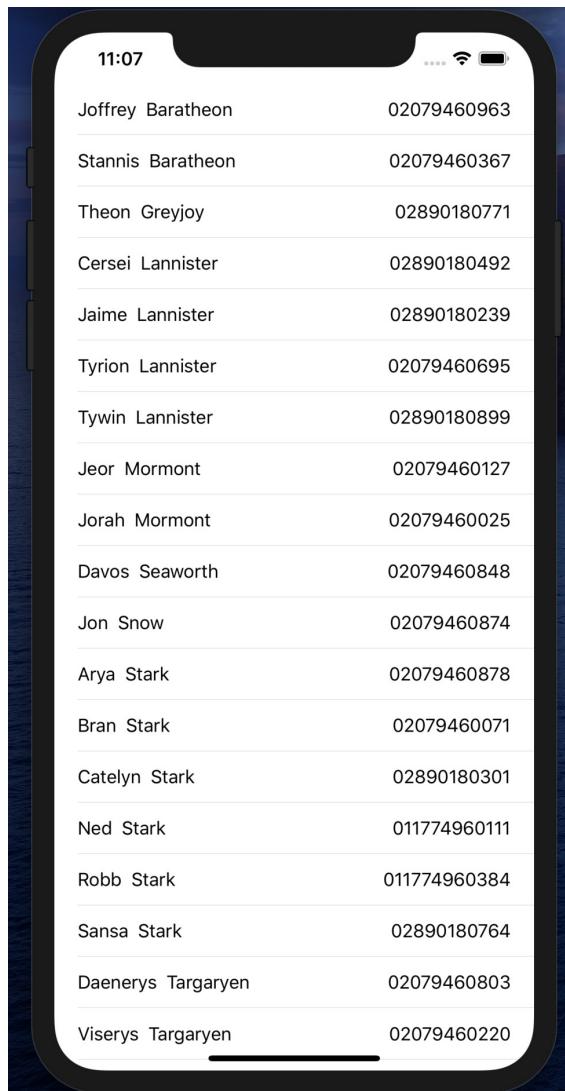


Figure 13.5 – List of contacts from Core Data

How it works...

The basic Core Data Stack is the same as what's shown in the *Integrating Core Data with SwiftUI* recipe. You can refer to it for any explanations.

Specifically for fetching Core Data objects, all the magic of our SwiftUI Core Data integration resides in the `@FetchRequest` property wrapper, where we set our `sortDescriptors`, and they magically populate the `contacts` property.

`@FetchRequest` infers the type of the entity from the type of the property so that we don't need to provide the type of the object to fetch.

`@FetchRequest` also accepts an optional `Predicate` as a parameter so that we can filter the fetched data before it's extracted from the repository.

`@FetchRequest` retrieves an object from `managedObjectContext` and expects it in `@Environment(\.managedObjectContext)`. As you may recall, in the `@main` struct, we are injecting `managedObjectContext` into `@Environment(\.managedObjectContext)` so that our code will work without needing any further configuration.

Adding Core Data objects from a SwiftUI view

A store without any data in it is useless. In this recipe, we will learn how easy it is to implement a function that will add data to Core Data in SwiftUI.

In this recipe, we are going to implement a simple Contact app where we can add storable contact profiles to a Core Data database.

Getting ready

Create a SwiftUI app called `AddContacts`.

Before we start this recipe, complete *Steps 1 to 4* of the *Showing Core Data objects with @FetchRequest* recipe. Then, you can complete this recipe.

How to do it...

Besides adding profiles, we must present our list of already saved contacts. For this, we are going to reuse some of the code from the *Showing Core Data objects with @FetchRequest* recipe.

To add a profile, we are going to implement a simple modal view with three text fields: two for the full name and one for the phone number. Let's get started:

1. Let's look at the code for visualizing the contacts. Create a `ContactView` struct:

```
struct ContactView: View {  
    let contact: Contact  
    var body: some View {  
        HStack {  
            Text(contact.firstName ?? "-")  
            Text(contact.lastName ?? "-")  
            Spacer()  
            Text(contact.phoneNumber ?? "-")  
        }  
    }  
}
```

2. In the `ContentView` struct, add the list of contacts via the `@FetchRequest` property wrapper:

```
struct ContentView: View {  
    @FetchRequest(  
        sortDescriptors: [  
            NSSortDescriptor(keyPath:  
                \Contact.lastName,  
                ascending: true),  
            NSSortDescriptor(keyPath:  
                \Contact.firstName,  
                ascending: true),  
        ]  
    )  
    var contacts: FetchedResults<Contact>  
}
```

3. The body function simply iterates on this contacts list and presents each profile. The list is embedded in NavigationView. Add the following code to achieve this:

```
struct ContentView: View {  
    //...  
    var body: some View {  
        NavigationView {  
            List(contacts, id: \.self) {  
                ContactView(contact: $0)  
            }  
            .listStyle(.plain)  
            .navigationBarTitle("Contacts",  
                displayMode: .inline)  
        }  
    }  
}
```

4. To present the modal view so that we can add a contact, add a button to the navigation bar that toggles the @State property called `isAddContactPresented`:

```
struct ContentView: View {  
    //...  
    @State  
    private var isAddContactPresented = false  
  
    var body: some View {  
        NavigationView {  
            //...  
            .navigationBarTitle("Contacts",  
                displayMode: .inline)  
            .navigationBarItems(trailing:  
                Button {  
                    isAddContactPresented.toggle()  
                } label: {  
                    Image(systemName: "plus")  
                    .font(.headline)  
                }  
            )  
        }  
    }  
}
```

```
        })
    }
}
```

5. If the `isAddContactPresented` property is `true`, we will present our View so that we can add the necessary contact details. Add a `.sheet` modifier to open the new page:

```
struct ContentView: View {
//...
@EnvironmentObject
var coreDataStack: CoreDataStack

var body: some View {
    NavigationView {
        //...
        .navigationBarItems(trailing:
            //...
        ))
    .sheet(isPresented:
        $isAddContactPresented) {
            AddNewContact(isAddContactPresented:
                $isAddContactPresented)
        .environmentObject(coreDataStack)
    }
}
}
```

6. The `AddNewContact` View's `struct` has three text fields for the profile details. These are for the first name, last name, and phone number:

```
struct AddNewContact: View {
@EnvironmentObject
var coreDataStack: CoreDataStack

@Binding
```

```
    var isAddContactPresented: Bool  
    @State  
    var firstName = ""  
    @State  
    var lastName = ""  
    @State  
    var phoneNumber = ""  
  
}
```

7. The body function renders the text fields in a vertical stack, embedded in NavigationView. Add the following code:

```
struct AddNewContact: View {  
    //...  
    var body: some View {  
        NavigationView {  
            VStack(spacing: 16) {  
                TextField("First Name", text:  
                    $firstName)  
                TextField("Last Name", text:  
                    $lastName)  
                TextField("Phone Number", text:  
                    $phoneNumber)  
                    .keyboardType(.phonePad)  
                Spacer()  
            }  
            .padding(16)  
            .navigationTitle("Add A New Contact")  
        }  
    }  
}
```

8. Inside NavigationView, add a button for when we have finished filling in the contact details:

```
var body: some View {
    NavigationView {
        //...
        .navigationTitle("Add A New Contact")
        .navigationBarItems(trailing:
            Button(action: saveContact) {
                Image(systemName: "checkmark")
                    .font(.headline)
            }
            .disabled(isDisabled)
        )
    }
}
```

9. This button is only enabled if all the fields have been filled in:

```
private var isDisabled: Bool {
    firstName.isEmpty ||
    lastName.isEmpty ||
    phoneNumber.isEmpty
}
```

10. When this button is clicked, we will insert a new contact into the database and save it. Achieve this by adding the following code:

```
private func saveContact() {
    coreDataStack.insertContact(firstName: firstName,
                                lastName: lastName,
                                phoneNumber: phoneNumber)
    isAddContactPresented.toggle()
}
```

Now, we can run the app, add contacts, and see them visualized in the list:

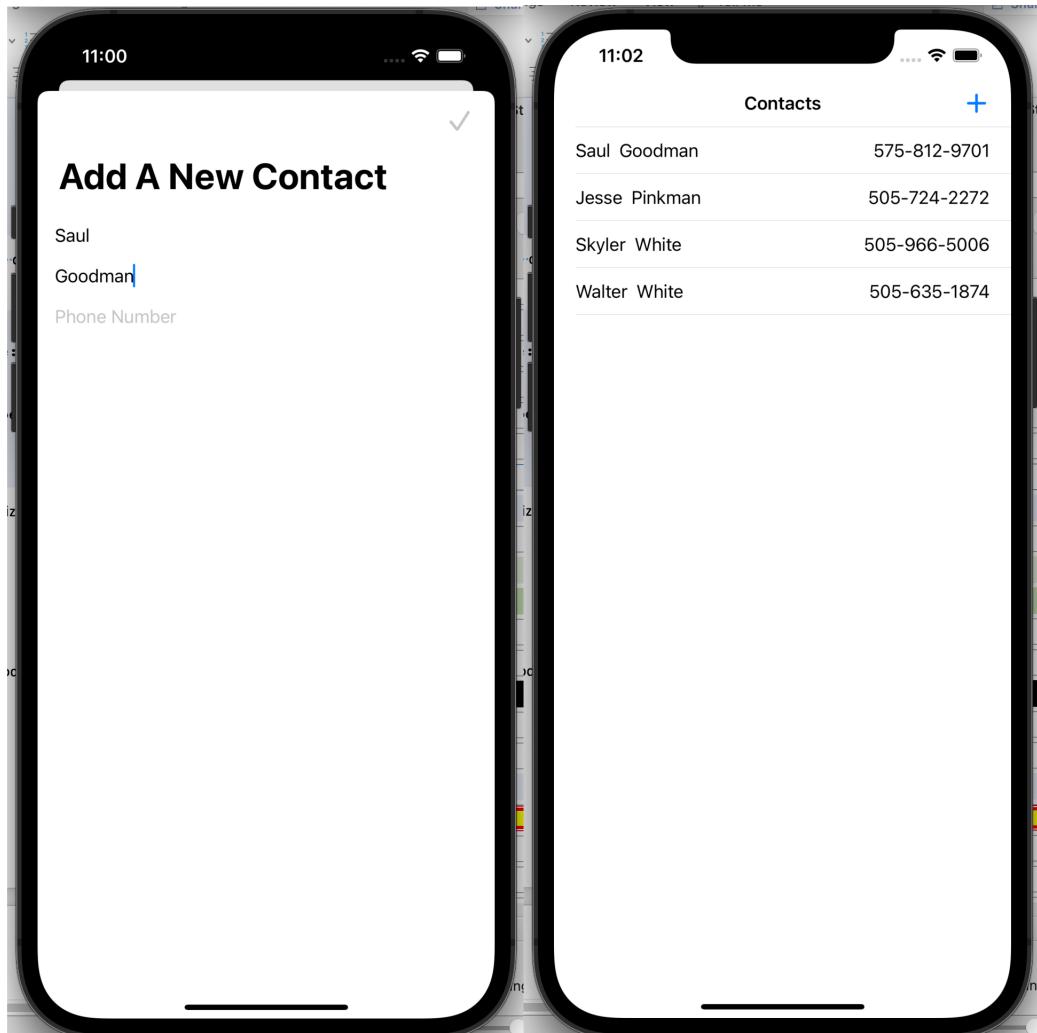


Figure 13.6 – Adding a contact to the list

How it works...

This recipe is built on the foundation of two other recipes: *Integrating Core Data with SwiftUI*, which introduced the `CoreDataStack` class and injected it into `@Environment`, and *Showing Core Data objects with @FetchRequest*, where we learned how to fetch objects from Core Data storage. Please refer to them for more in-depth explanations.

In this recipe, we injected the `coreDataStack` instance and used it in the `AddNewContact` View to save the object.

In general, since saving is a costly operation, it is better to check if any changes have been made before saving the context. This is what we do in the `save()` function of the `CoreDataStack` class:

```
func save () {
    guard managedObjectContext.hasChanges else { return }
    do {
        try managedObjectContext.save()
    } catch {
        print(error)
    }
}
```

Again, for simplicity, we didn't add any error recovery functions, so if an error occurs while saving the contact, the app prints an error in the console. I strongly recommend enriching this app, such as by adding something to warn the user that the contact wasn't saved in case of an error, and maybe they should try again.

Filtering Core Data requests using a predicate

An essential characteristic of Core Data is the possibility of filtering the results of a `@FetchRequest` so that only the objects that match a filter are retrieved from the repository and transformed into actual objects.

A predicate is a condition that the Core Data objects must satisfy to be fetched; for example, the name must be shorter than 5 characters, or the age of a person should be greater than 18. The conditions in a predicate can also be composite; for example, *fetch all the data where the name is equal to "Lewis" and the age is greater than 18*.

Even though the property wrapper accepts `NSPredicate`, which is a filter for Core Data, the problem is that this cannot be dynamic, which means it must be created at the beginning. It cannot change during the life cycle of the view as a result of a search text field, for example.

In this recipe, we'll learn how to create a dynamic filter for a contact list, where the user can restrict the list of visualized contacts with a search text field. When we set something in that search field, we are filtering the contacts whose surnames start with the value of the search field.

Getting ready

Create a SwiftUI app called `FilterContacts`.

Before we start this recipe, complete *Steps 1 to 7* of the *Showing Core Data objects with @FetchRequest* recipe. Then, you can complete this recipe.

How to do it...

To show our contacts, we are going to reuse part of the code provided in the *Showing Core Data objects with @FetchRequest* recipe. I suggest that you go back to it if you want to gain a better insight.

To filter the contacts, we must enrich the `ContentView` struct with a searchable text field and create a `FilteredContacts` view to pass the value of that text field.

Let's get started:

1. Create a `FilteredContacts` view whose `init()` function accepts a `filter` string as parameter, and it build a `fetchRequest` to fetch the contacts from the Core Data storage:

```
struct FilteredContacts: View {  
    let fetchRequest: FetchRequest<Contact>  
  
    init(filter: String) {  
        let predicate: NSPredicate? = filter.isEmpty ?  
            nil :  
            NSPredicate(format: "lastName BEGINSWITH %@",  
                       filter)  
        fetchRequest = FetchRequest<Contact>(  
            sortDescriptors: [  
                NSSortDescriptor(keyPath:  
                    \Contact.lastName,  
                    ascending: true),  
                NSSortDescriptor(keyPath:  
                    \Contact.firstName,  
                    ascending: true)],  
            predicate: predicate)  
    }  
}
```

2. The body of the `FilteredContacts` view simply presents the contacts in a `List` view. Add the following code:

```
var body: some View {
    List(fetchRequest.wrappedValue, id: \.self) {
        ContactView(contact: $0)
    }
    .listStyle(.plain)
}
```

3. Embed the view we created in *Steps 1 and 2* in `NavigationView` in the body of the `ContentView` struct:

```
struct ContentView: View {
    @State
    private var searchText : String = ""

    var body: some View {
        NavigationView {
            FilteredContacts(filter: searchText)
                .navigationBarTitle("Contacts",
                    displayMode:.inline)
        }
        .searchable(text: $searchText)
    }
}
```

Upon running the app, we will see that changing the text in `SearchBar` changes the list of visualized contacts accordingly:

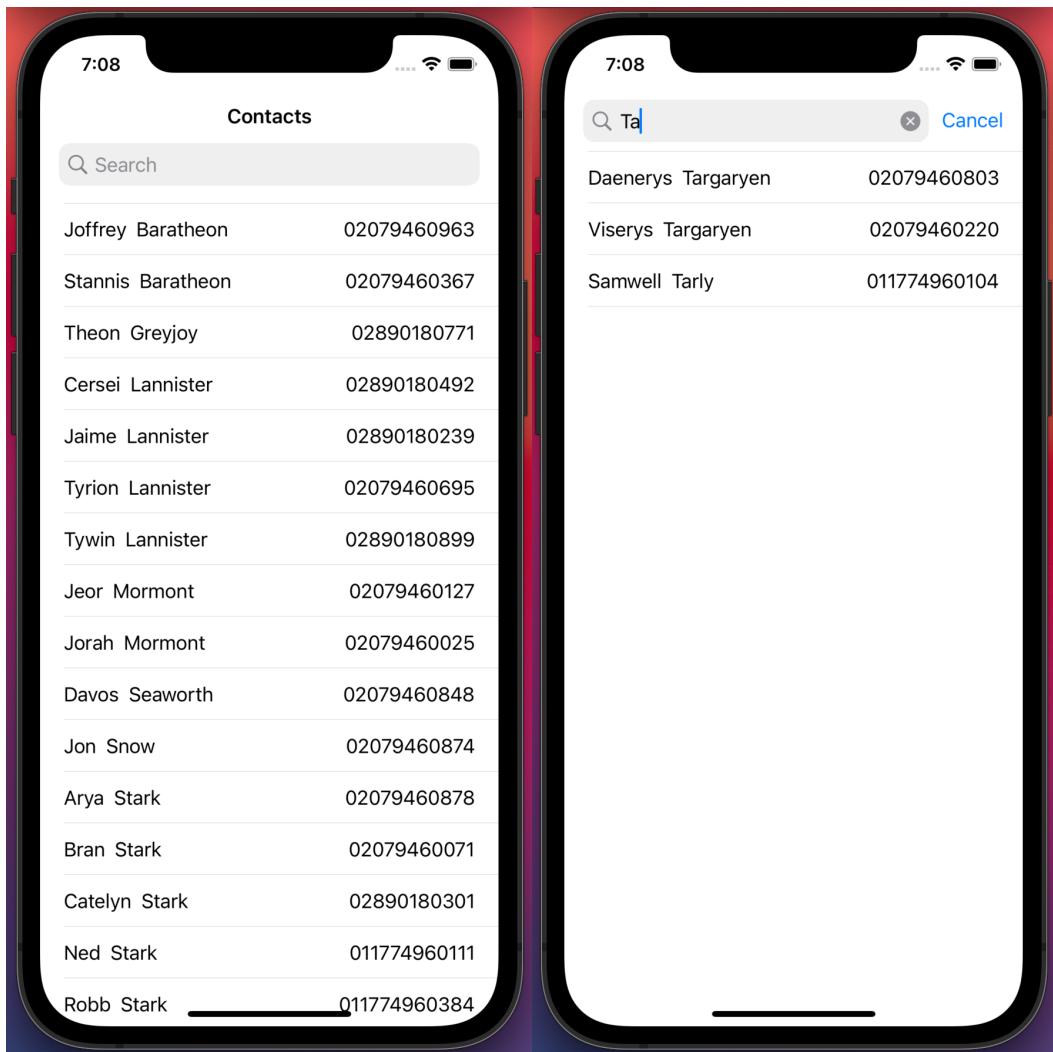


Figure 13.7 – Core Data dynamic filtering

How it works...

Taking inspiration from the *Showing Core Data objects with @FetchRequest recipe*, it would have been nice to write a definition such as the following:

```
@FetchRequest(  
    sortDescriptors: [  
        NSSortDescriptor(keyPath: \Contact.lastName,  
                         ascending: true),  
        NSSortDescriptor(keyPath: \Contact.firstName,  
                         ascending: true),  
    ],  
    predicate: NSPredicate(format: "lastName BEGINSWITH  
        %@", filter)  
)  
  
var contacts: FetchedResults<Contact>
```

Unfortunately, this is not possible (at least, not yet) since the filter is a `@State` variable that can change during the life cycle of the view.

To overcome this limitation, we created a dedicated component, `FilteredContacts`, to present the filtered contacts, injecting the text that acts as a filter.

In the `FilteredContacts` component, we created a `FetchRequest` with the text we passed as a parameter into `init()` as a filter for the `FetchRequest` class.

The `FetchRequest` component, via `managedObjectContext`, retrieves the objects and stores them internally. We can reach them via the `.wrapped()` function.

Since these objects are returned as a collection, we can iterate them as follows:

```
List(fetchRequest.wrappedValue, id: \.self) { contact in  
    //...  
}
```

This is a solution that follows the SwiftUI philosophy: split the logic into smaller components that can be represented as a `View`.

Besides the actual problem this solves, I think that, in this recipe, we learned how to solve this problem in the SwiftUI way. This is a process that can be used every time we face a similar problem in SwiftUI code.

The last thing to note is the use of the `.searchable()` modifier: when it is added to `NavigationView`, it adds a `SearchBar` component to the top of the view contained inside `NavigationView`. In just one line of code, you can implement a very elegant searchable view.

Deleting Core Data objects from a SwiftUI view

How can you delete objects from a Core Data repository? Removing objects is almost as important as adding them. In this recipe, we'll learn how to integrate the Core Data delete options into a SwiftUI app.

Getting ready

Create a SwiftUI app called `DeleteContacts`.

Before we start this recipe, complete *Steps 1 to 7* of the *Showing Core Data objects with @FetchRequest* recipe. Then, you can complete this recipe.

How to do it...

We are going to reuse part of the code provided in the *Showing Core Data objects with @FetchRequest* recipe. Please refer to that recipe if you want to find out more.

Let's get started:

1. In the `ContentView` struct, add the list of contacts via the `@FetchRequest` property wrapper:

```
struct ContentView: View {  
    @FetchRequest(  
        sortDescriptors: [  
            NSSortDescriptor(keyPath:  
                \Contact.lastName,  
                ascending: true),  
            NSSortDescriptor(keyPath:  
                \Contact.firstName,  
                ascending: true),  
        ]  
    )  
    var contacts: FetchedResults<Contact>  
}
```

- Finally, the body function will simply render the list of contacts. Add the following code:

```
struct ContentView: View {  
    //...  
    var body: some View {  
        List(contacts, id: \.self) {  
            ContactView(contact: $0)  
        }  
        .listStyle(.plain)  
    }  
}
```

- So far, this is the code we've been using to render a list of objects from Core Data. Now, let's add the `.onDelete()` modifier, which will call the `deleteContact()` function:

```
var body: some View {  
    //...  
    ForEach(contacts, id: \.self) {  
        ContactView(contact: $0)  
    }  
    .onDelete(perform: deleteContact)  
    //...  
}
```

- Add the `deleteContact()` function, which simply fetches the `Contact` object from the list and removes it from the `managedObjectContext` instance:

```
private func deleteContact(at offsets: IndexSet) {  
    guard let index = offsets.first else {  
        return  
    }  
  
    managedObjectContext.delete(contacts[index])  
}
```

When you run the app, the cells can be swiped. Upon swiping a cell to the left, the standard red **Delete** button will be shown so that you can remove that cell:

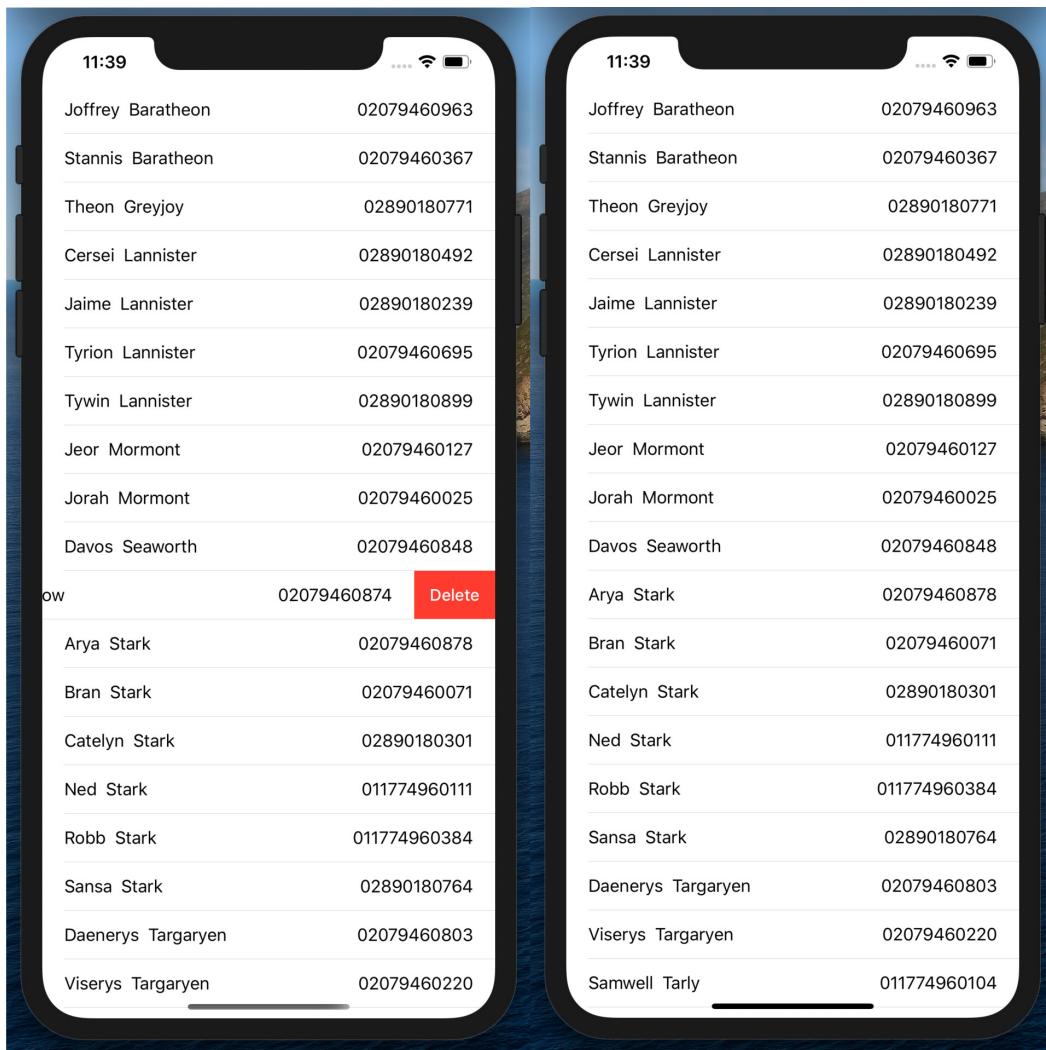


Figure 13.8 – Deleting a Core Data object in SwiftUI

How it works...

If you want more information regarding how to create objects and how to render them in the View, please read the *Showing Core Data objects with @FetchRequest* recipe in this chapter. Here, we are only looking at the *delete* feature.

As you have seen, the way you can integrate a `delete()` function into Core Data from SwiftUI is just a matter of calling the function with the right object.

Here, we added a `.onDelete()` modifier to the `ForEach` component inside the `List` view. From there, we called a function to delete the selected object.

The `.onDelete()` modifier is called with the `index` property of the row to remove as a parameter. However, to delete an object in Core Data, we must fetch the exact `Contact` object from the Core Data storage. Luckily, we have the list of these objects in the `contacts` property, which means we can refer to the object to delete using that property and its `index`.

Bear in mind that a deletion operation is irreversible. It would be safer to add a dialog box here to ask for confirmation from the user before removing the object from the Core Data storage.

Presenting data in a sectioned list with `@SectionedFetchRequest`

Sometimes, a planned `List` View isn't enough to present a set of values. Depending on the type of data to be presented, you may need to show them in sections. Think, for example, of a settings list where the settings are aggregates for types of settings; or a list of contacts, where the contacts are aggregated into sections for the surname initial.

SwiftUI provides a mechanism for fetching objects from Core Data storage that's already been aggregated into sections, allowing it to easily be presented in a sectioned `List` View.

In this recipe, we'll implement a list of contacts where the contacts are grouped into sections, depending on the initial of their surname.

Getting ready

Create a SwiftUI app called `SectionedContacts`.

Before we start this recipe, complete *Steps 1 to 7* of the *Showing Core Data objects with `@FetchRequest`* recipe. Then, you can complete this recipe.

How to do it...

We are going to reuse part of the code provided in the *Showing Core Data objects with `@FetchRequest`* recipe. Please refer to that recipe if you want to find out more.

Using the `@SectionedFetchRequest` property wrapper, we will create a `List` View with contacts separated into sections, depending on the initial of the surname.

The steps are as follows:

1. Add a function to an extension of the `Contact` class to return the initial of the surname:

```
extension Contact {  
    @objc var lastNameInitial: String {  
        get {  
            if let initial = lastName?.prefix(1) {  
                return String(initial)  
            } else {  
                return ""  
            }  
        }  
    }  
}
```

2. Add a computed property to an extension of the `Contact` class to return the surname initial:

```
extension Contact {  
    @objc var lastNameInitial: String {  
        get {  
            String(lastName?.prefix(1) ?? "")  
        }  
    }  
}
```

3. In the `ContentView` component, add the `@SectionedFetchRequest` property wrapper to fetch the contacts:

```
struct ContentView: View {  
    @SectionedFetchRequest<String, Contact>(  
        sectionIdentifier: \.lastNameInitial,  
        sortDescriptors: [  
            NSSortDescriptor(keyPath:  
                \Contact.lastName,
```

```
        ascending: true),
        NSSortDescriptor(keyPath:
            \Contact.firstName,
            ascending: true),
    ],
    animation: .default
)
var sectionedContacts
}
```

4. Finally, in the body function of the component, present Contacts grouped into sections:

```
var body: some View {
    NavigationView {
        VStack {
            List(sectionedContacts) { section in
                Section(header:
                    Text("Section for
                        '\\" + (section.id) + "'"))
                ForEach(section) {
                    ContactView(contact: $0)
                }
            }
        }
    }
    .navigationTitle("Contacts")
}
}
```

By running the app, we can see the contacts grouped as per the surname initial in a sectioned List view:

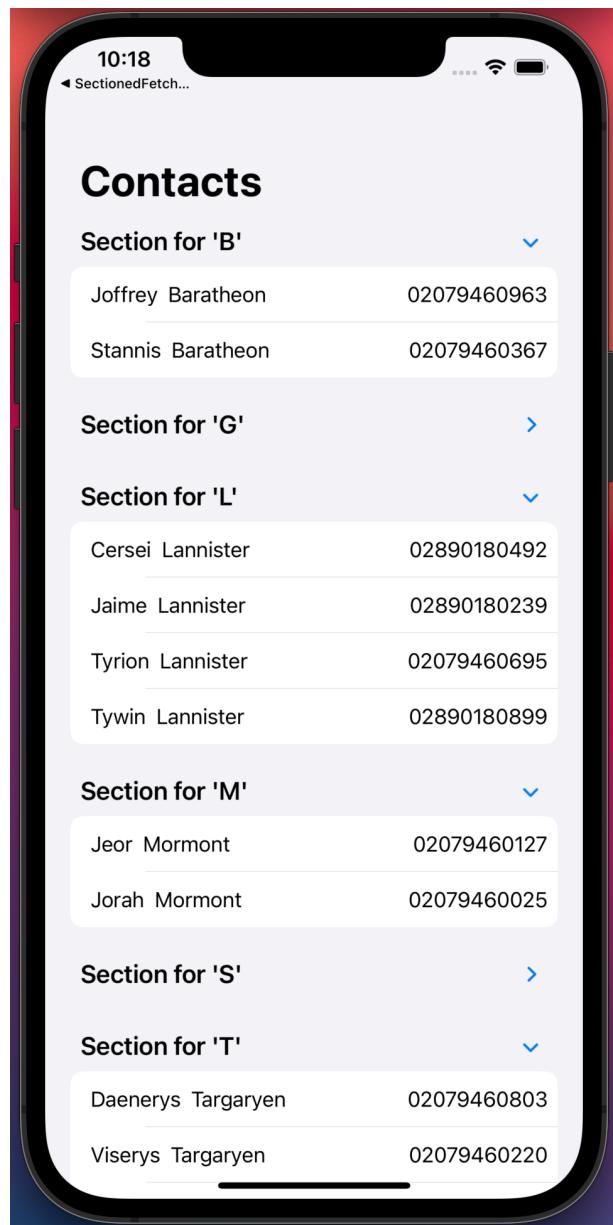


Figure 13.9 – Sectioned contacts List view

How it works...

If you want more information regarding how to create objects and how to render them in the View, please read the *Showing Core Data objects with @FetchRequest* recipe in this chapter. Here, we are only looking at the sectioned fetching feature.

@SectionedFetchRequest is the property wrapper that does all the magic. You can see that its signature is similar to the one of the @FetchRequest property wrapper, which retrieves objects from Core Data storage in a flat sequence.

One additional parameter, `sectionIdentifier`, allows this property wrapper to group the list of contacts by applying `keyPath`, which is passed and identifies the object's section.

Since `lastNameIdentifier` isn't part of the model, we added it as a computed property to an extension of the `Contact` class:

```
extension Contact {  
    @objc var lastNameInitial: String {  
        get {  
            String(lastName?.prefix(1) ?? "")  
        }  
    }  
}
```

Since that property must be accessed by Core Data, which uses the **ObjectiveC** runtime to do so, we have to decorate its definition with `@objc`. ObjectiveC was the primary language that was used to build iOS before Swift. Some system frameworks still rely on ObjectiveC system libraries and conventions, such as Core Data.

14

Creating Cross-Platform Apps with SwiftUI

SwiftUI makes it easy to take some or all the code written for one Apple platform and use it to create an app for another platform in the Apple ecosystem. For example, in this chapter, we create an iOS app and then reuse some of the components to create a macOS and a watchOS app.

When using Cross-Platform development in SwiftUI, we share common resources between each platform while creating other resources that are platform-specific. For example, models may be shared across platforms, but certain images and SwiftUI views would be made platform-specific. Creating platform-specific views allows us to follow platform-specific best-practice design guidelines and improve the user experience provided by our apps.

This chapter covers some of the Cross-Platform functionalities of SwiftUI with the following recipes:

- Creating an iOS app in SwiftUI
- Creating the macOS version of the iOS app
- Creating the watchOS version of the iOS app

Technical requirements

The code in this chapter is based on Xcode 13.

You can find the code in the book's GitHub repository under the path <https://github.com/PacktPublishing/SwiftUI-Cookbook-2nd-Edition/tree/main/Chapter14-Cross-Platform-SwiftUI>.

Creating an iOS app in SwiftUI

Before going Cross-Platform with your development, we will first create an iOS app. We will be starting with the app created during our work on the *Using mock data for previews* recipe in *Chapter 4, Viewing while Building with SwiftUI Preview*. The version created for this recipe will be made more modular to allow for code reuse across platforms, and the resources will not be stored in the preview section of the app.

The app will display a list of insects where you can click on any insect in the list to see more details about it. The data regarding the insects will be read from a JSON file and made available to our views using an `@Environment` variable.

Getting ready

Let's create a new single-view iOS app in SwiftUI named `Cross-Platform`.

How to do it...

We will set up the model and data source used in this recipe, and then proceed to create the views and subviews needed to display the list of insects and the details of each one. The steps are as follows:

1. To stay organized, let's create a group to store our models:
 - a. Right-click on the `Cross-Platform` folder in the navigation pane.
 - b. Select **New Group**.
 - c. Name the new folder `Models`.
2. Create a folder for the resource files used in the project:
 - a. Right-click on the `Cross-Platform` folder in the navigation pane.
 - b. Select **New Group**.
 - c. Name the new folder `Resources`.
3. Drag and drop the `insectData.json` file from the book's `Resources` folder from GitHub into this project's `Resources` folder (<https://github.com/PacktPublishing/SwiftUI-Cookbook-2nd-Edition/tree/main/Resources/Chapter14/Recipe01>).
4. Click on `insectData.json` to view its content. From the data, we notice every insect has an ID, an image name, a habitat, and a description. So, let's create a model to match the JSON content.
5. Select the `Models` folder from the project navigation pane:
 - a. Press *Command* (⌘) + *N*.
 - b. Select **Swift File**.
 - c. Click **Next**.
 - d. Click on the **Save As** field and enter the text `Insect`.
 - e. Click **Finish**.

6. Create the Insect struct:

```
struct Insect : Decodable, Identifiable, Hashable{  
    var id: Int  
    var imageName:String  
    var name:String  
    var habitat:String  
    var description:String  
}
```

7. Below the struct declaration, create an instance of the struct that will be used for previews:

```
let testInsect = Insect(id: 1, imageName: "grasshopper",  
name: "grass", habitat: "pond", description: "long  
description here")
```

8. With the Models folder still selected, create the insectData model. We'll use the model to read the JSON file and make it available to the app:

- a. Press *Command* (⌘) + *N*.

- b. Select **Swift File**.

- c. Click **Next**.

- d. Click on the **Save As** field and enter the text **InsectData**.

- e. Click **Finish**.

9. Create an Observable object to store the data from the **insectData.json** file as an array of insects:

```
final class InsectData: ObservableObject {  
    @Published var insects = testInsects  
}
```

10. The code does not build yet because we have not yet defined the value of the **testInsects** variable. Let's read the file and decode its contents into the **Insect** struct. Place this code below the class definition in the previous step:

```
var testInsects : [Insect] {  
    guard let url = Bundle.main.url(forResource:  
    "insectData", withExtension: "json"),  
    let data = try? Data(contentsOf: url)
```

```
else{
    return []
}
let decoder = JSONDecoder()
let array = try?decoder.decode([Insect].self,
from: data)
return array ?? [testInsect]
```

11. The rest of the files we'll create should be placed within the Cross-Platform folder. First, let's create a SwiftUI view named `InsectCellView`:
 - a. Press `Command (⌘)+ N`.
 - b. Select **SwiftUI View**.
 - c. Click **Next**.
 - d. Click on the **Save As** field and enter the text `InsectCellView`.
 - e. Click **Finish**.
12. The `InsectCellView` will contain the design for a row in our `insect` list:

```
struct InsectCellView: View {
    var insect:Insect
    var body: some View {
        HStack{
            Image(insect.imageName)
                .resizable()
                .aspectRatio(contentMode: .fit)
                .clipShape(Rectangle())
                .frame(width:100, height: 80)

            VStack(alignment: .leading){
                Text(insect.name).font(.title)
                Text(insect.habitat)
            }.padding(.vertical)
        }
    }
}
```

13. To preview the design, pass the `testInsect` variable to the `InsectCellView` function call in `InsectCellView_Previews`:

```
struct InsectCellView_Previews: PreviewProvider {  
    static var previews: some View {  
        InsectCellView(insect: testInsect)  
    }  
}
```

The `InsectCellView` preview should look as follows:



Figure 14.1 – InsectCellView preview

14. We will also show the details regarding a particular insect in its own view.

Let's create a SwiftUI view called `InsectDetailView` (see *Step 11* of this recipe for details on creating a new SwiftUI view). The `InsectDetailView` should display the attributes of the insect:

```
struct InsectDetailView: View {  
    var insect: Insect  
    var body: some View {  
        VStack{  
            Text(insect.name)  
                .font(.largeTitle)  
            Image(insect.imageName)  
                .resizable()  
                .aspectRatio(contentMode: .fit)  
            Text("Habitat")  
                .font(.title)  
            Text(insect.habitat)  
            Text("Description")  
                .font(.title)  
                .padding()  
            Text(insect.description)  
        }  
    }  
}
```

15. Preview the design by passing the `testInsect` variable to the `InsectDetailView()` function call in the `previews` section:

```
struct InsectDetailView_Previews: PreviewProvider {  
    static var previews: some View {  
        InsectDetailView(insect: testInsect)  
    }  
}
```

The `InsectDetailView` preview should look as follows:

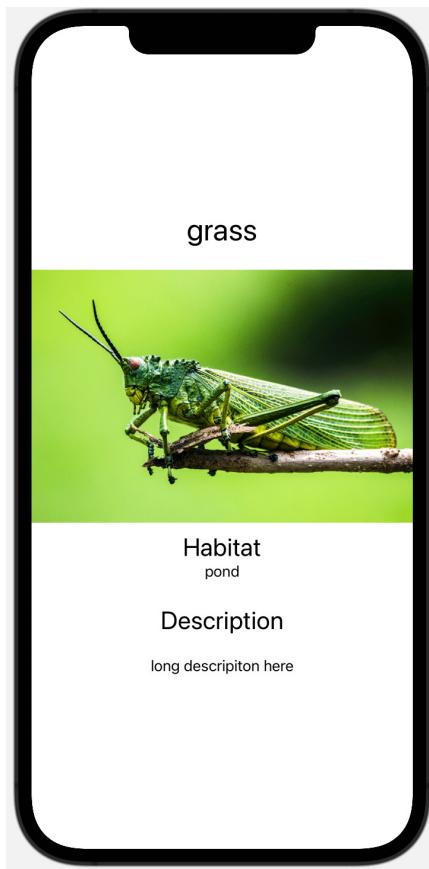


Figure 14.2 – `InsectDetailView` preview

16. Now create a SwiftUI view called `InsectListView`.
17. Above the `body` variable of our `InsectListView` SwiftUI view, declare and initialize our `@EnvironmentObject` that contains the insect data as an array of the `Insect` struct.
18. Within the `InsectListView` body, implement a `List` view that iterates over the data and displays it using the `InsectCellView` SwiftUI views:

```
List{  
    ForEach(insectData.insects){insect in  
        NavigationLink(  
            destination: InsectDetailView(insect: insect)){  
            InsectCellView(insect: insect)  
        }  
    }  
}
```

```
        }
    }
}.navigationBarTitle("Insects", displayMode: .inline)
```

19. To preview the design, add the `.environmentObject()` modifier to the `InsectListView()` function call in our canvas preview code:

```
struct InsectListView_Previews: PreviewProvider {
    static var previews: some View {
        InsectListView().environmentObject(InsectData())
    }
}
```

The `InsectListView` preview should look as follows:

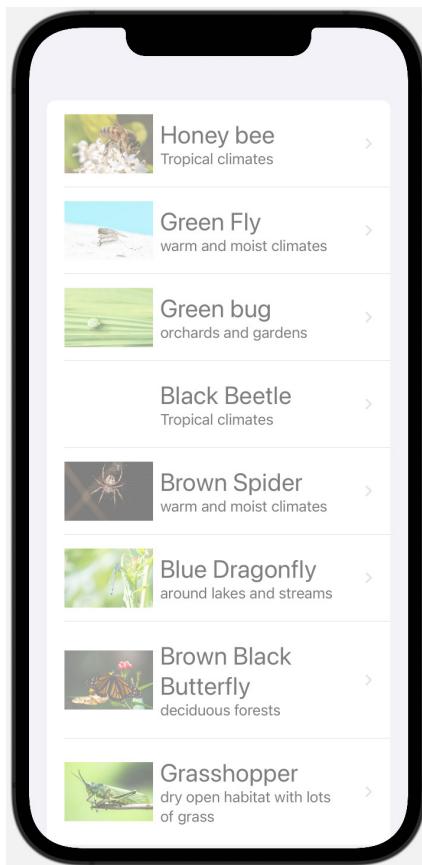


Figure 14.3 – InsectListView preview

20. So far, we have not provided a path for the user to view our list when running the app. The `ContentView.swift` file is the entry point for the app. Let's set it up. Add the `@Environment` variable above the `body` variable:

```
@EnvironmentObject var insectData: InsectData
```

21. Add a `NavigationView` and `InsectListView` to the `body` variable:

```
var body: some View {
    NavigationView{
        InsectListView()
    }
}
```

22. Add the `.environmentObject()` modifier to the `ContentView_Previews` to fetch our data:

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

The `ContentView` preview should look as follows:

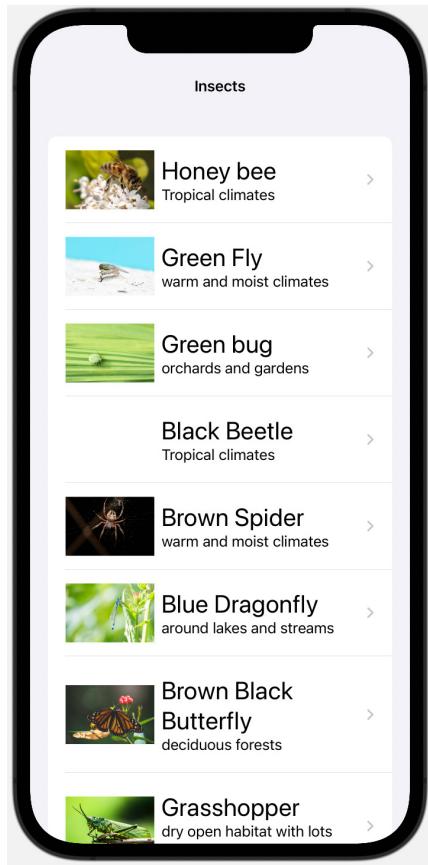


Figure 14.4 – ContentView preview

23. Now, you should be able to run the preview in the Xcode canvas to view the list of insects and insect details. However, running the program in an iPhone simulator presents a blank screen. Let's solve this by passing our `.environmentObject()` modifier to our scene function in the `Cross_PlatformApp.swift` file:

```
WindowGroup {  
    ContentView()  
        .environmentObject(InsectData())  
}
```

Congrats! You can now view the list of insects from the device previews or within a simulated device.

How it works...

Our goal in this recipe was to read insect data from a JSON file and display the content in our SwiftUI app. To get started, we observed the raw JSON data and created a struct that holds the data – that is, the `Insect` struct.

The `Insect` struct implements the `Decodable` and `Identifiable` protocols. The `Decodable` protocol allows us to decode data from our `insectData.json` file and use it to create our `Insect` struct. The `Identifiable` protocol allows us to uniquely identify each insect in an array and use the `ForEach` loop without an `id` parameter.

The `InsectData.swift` file is the most important component of this recipe. It is used to read data from the `insectData.json` file and store it in an `ObservableObject` called `InsectData`. The `Observable` object contains the `@Published` variable called `insects`. `Observable` objects allow us to share data between multiple views.

We used a modular app design since our goal is to create a Cross-Platform application. Therefore, each view handles a single task. For example, the `InsectCellView` SwiftUI view takes an `insect` parameter and displays its content horizontally in an `HStack`.

The `InsectListView` SwiftUI view iterates through our insect data and displays the content in a list where each row is an `InsectCellView` SwiftUI view.

When a cell in the `InsectListView` is clicked, the app opens an `InsectDetailView` that contains the full information we have regarding the selected insect.

Creating the macOS version of the iOS app

Our app's iOS version showed a list of insects in one view and details regarding the selected insect in a separate view because of the limited amount of space available on a phone screen. However, a laptop screen has a larger amount of screen space. Therefore, we can display the list of insects on the left side of the screen and the details regarding the selected insect on the right side.

Getting ready

Download the chapter materials from GitHub:

<https://github.com/PacktPublishing/SwiftUI-Cookbook-2nd-Edition-/tree/main/Chapter14-Cross-Platform-SwiftUI/02>Create-the-MacOS-version>

Open the StartingPoint folder and double-click on Cross-Platform.xcodeproj to open the app built in the *Creating an iOS app in SwiftUI* recipe of this chapter. We will be continuing from where we left off in the previous recipe.

How to do it...

We'll create the macOS version of the app by reusing some components from the iOS version and creating custom components for the macOS version. The steps are as follows:

1. Create a macOS Target... in Xcode:

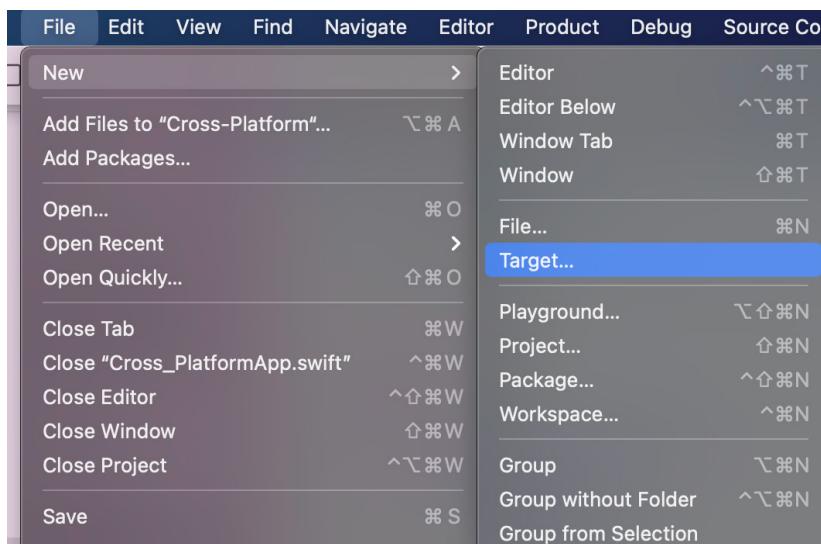


Figure 14.5 – Creating a new target

2. Choose the **macOS** template, scroll down, select **App**, then click **Next**:

Choose a template for your new target:

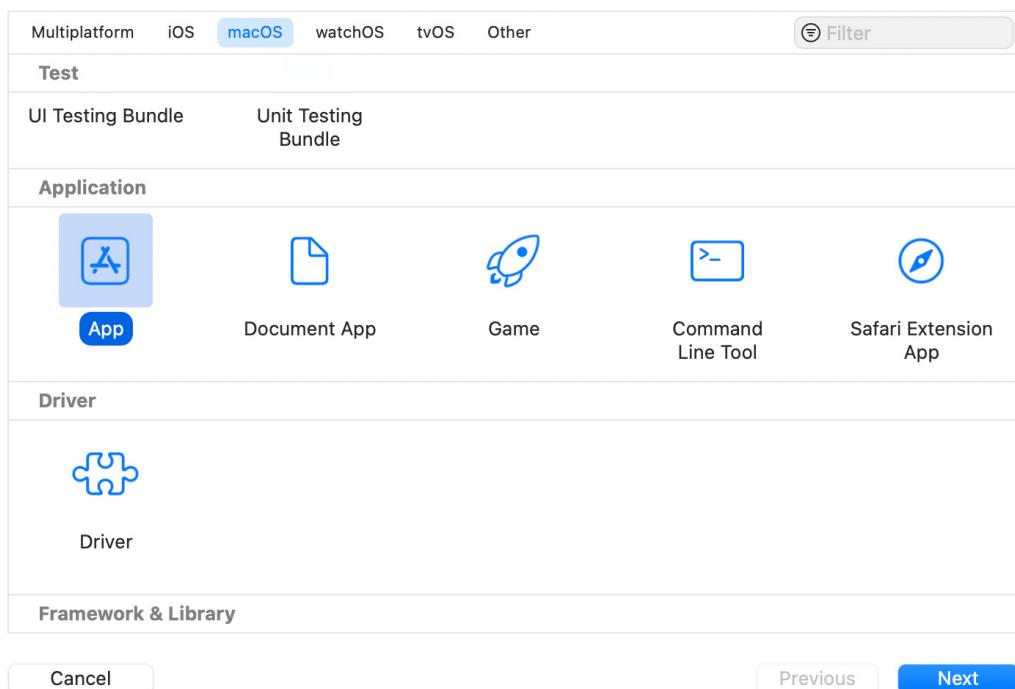


Figure 14.6 – Selecting macOS App

3. In the next screen, enter the product name, **macOS-Cross-Platform**.
4. To be able to preview and run the macOS applications, let's set the Xcode active scheme to **macOS-Cross-Platform**:

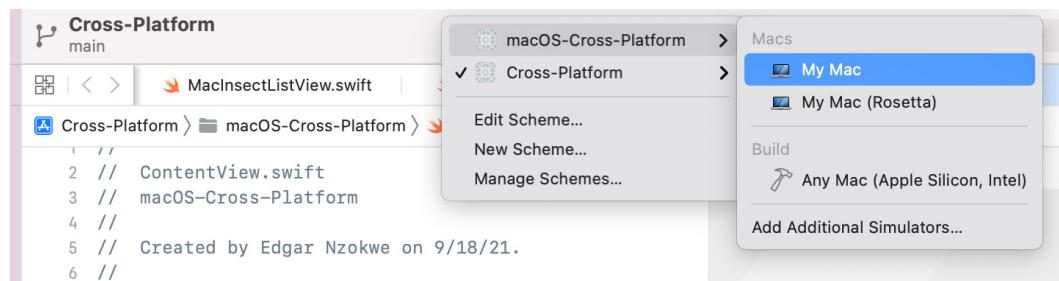


Figure 14.7 – Changing Xcode scheme to macOS-Cross-Platform

5. Select the ContentView.swift file in the macOS-Cross-Platform folder. Run the preview. It should look as follows:

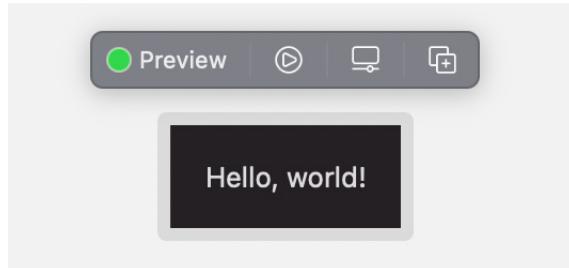


Figure 12.8 – macOS empty ContentView preview

6. For the next step, make sure the Navigator and Inspector panes are open. We will be using both panes to make certain files and resources available across targets:



Figure 12.9 – Navigator and Inspector panes in Xcode

7. Let's share files between multiple platforms. Select the Insect.swift, InsectData.swift, insectData.json, InsectDetailView.swift, and Assets.xcassets files located in the Models folder in our Cross-Platform app. To select multiple files, hold the *Command* key and click on the files listed.
8. In the Inspector pane, check the **macOS-Cross-Platform** checkbox:

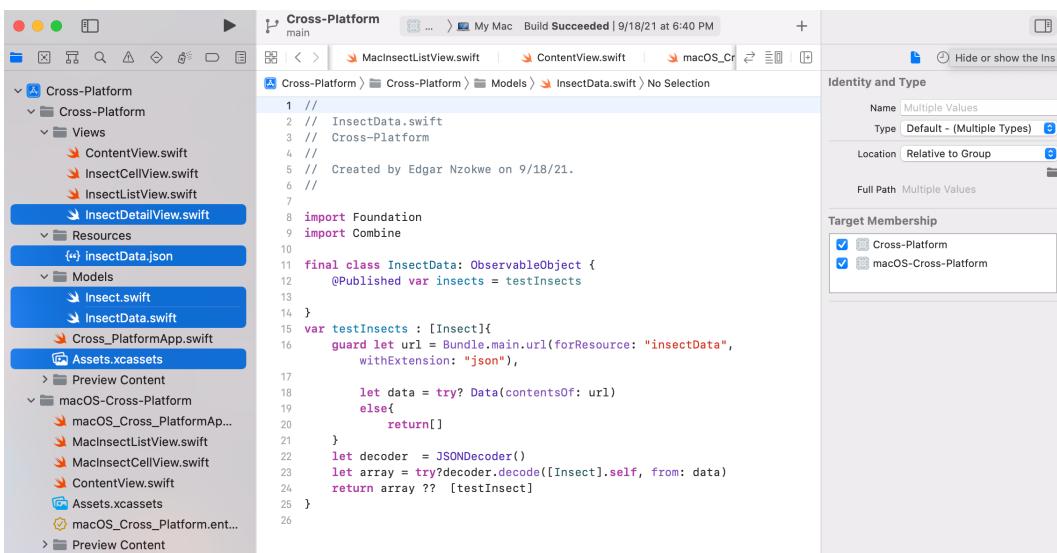


Figure 14.10 – Making iOS files accessible in the macOS target

9. Create a new MacInsectCellView SwiftUI view in the macOS-Cross-Platform folder. Make sure the right platform target is selected (**macOS-Cross-Platform**):

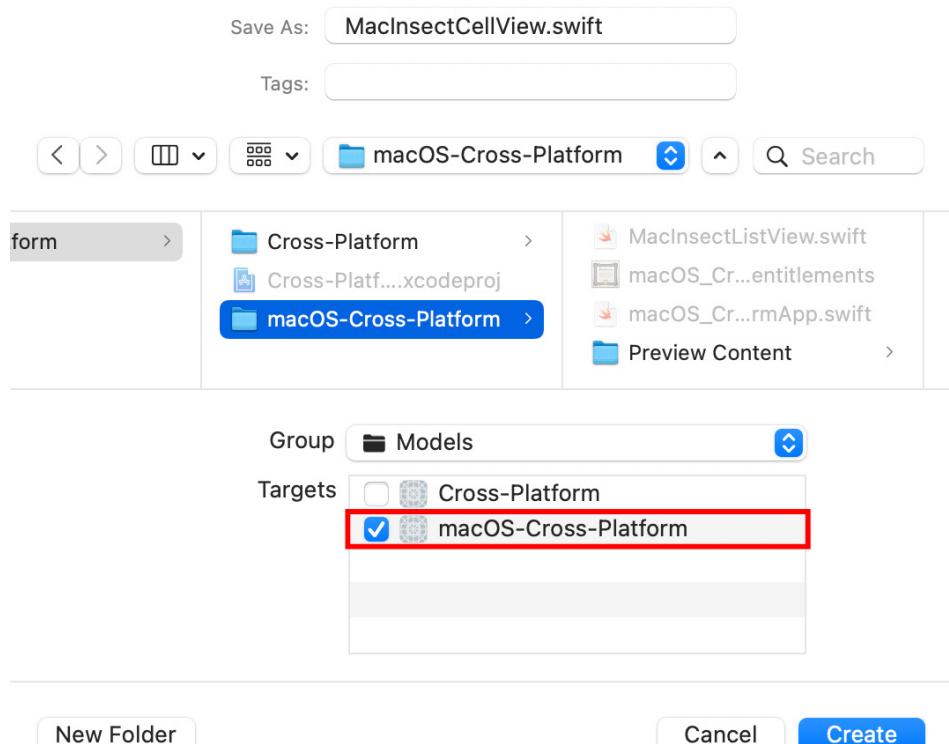


Figure 14.11 – Creating a new macOS view and selecting the right target

10. The content of MacInsectCellView is similar to that of InsectCellView in the iOS Cross-Platform app. The only difference is the size of the frame. The file content should look as follows:

```
struct MacInsectCellView: View {  
    var insect: Insect  
  
    var body: some View {  
        HStack{  
            Image(insect.imageName)  
                .resizable()  
                .aspectRatio(contentMode: .fit)
```

```
        .clipShape(Rectangle())
        .frame(width:160, height: 100)

    }

    VStack(alignment: .leading) {
        Text(insect.name).font(.subheadline)
        Text(insect.habitat)
    }.padding(.vertical)
}
}
```

11. Pass the `testInsect` variable in the preview to view the design within the Xcode canvas:

```
struct MacInsectCellView_Previews: PreviewProvider {
    static var previews: some View {
        MacInsectCellView(insect: testInsect)
    }
}
```

The preview should look as follows:



Figure 14.12 – MacInsectCellView preview

12. Create `MacInsectListView` to show a list of items. Make sure the **macOS-Cross-Platform** target is selected.

13. Add the `@EnvironmentObject` and `@Binding` variables above the `body` variable of the `MacInsectListView` struct:

```
@State var selectedInsect: Insect?  
@EnvironmentObject var userData: InsectData
```

14. Add a `List` view to the `body`. The `List` view should display the contents of our `insectData` array obtained from our environment variable. Add a `.tag()` modifier to `MacInsectCellView` to use it to identify the selected cell. Also add the `.listStyle()` modifier that presents the list using `SidebarListStyle()`:

```
var body: some View {  
    List(selection: $selectedInsect){  
        ForEach(userData.insects){ insect in  
            MacInsectCellView(insect:  
                insect).tag(insect)  
        }  
    }.listStyle(SidebarListStyle())  
}
```

15. Let's preview the design by passing in a constant binding and adding the `.environmentObject()` modifier to our `MacInsectListView()` function call in the preview:

```
struct MacInsectListView_Previews: PreviewProvider {  
    static var previews: some View {  
        MacInsectListView(selectedInsect:  
            .constant(testInsect))  
            .environmentObject(InsectData())  
    }  
}
```

The preview should look as follows:

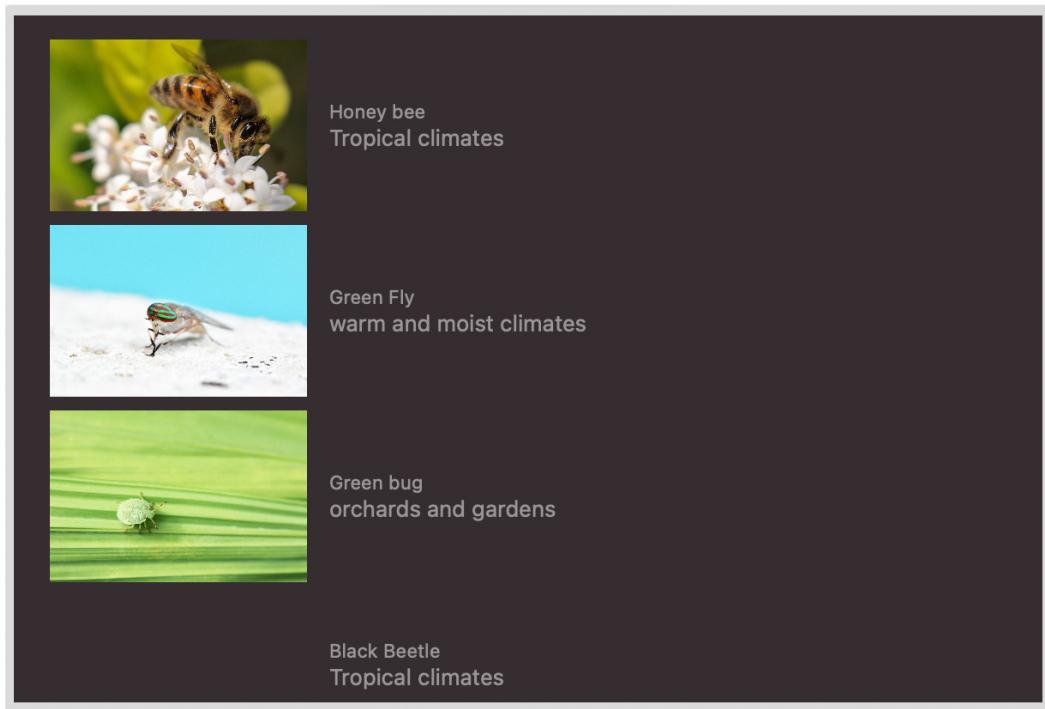


Figure 14.13 – MacInsectListView preview

16. At this point, the code doesn't build because the selection parameter of the `List` view requires the item being selected to be hashable. Let's make our `Insect` struct implement the `Hashable` protocol. Open the `Insect.swift` file in the `Cross-Platform` folder and add the `Hashable` protocol:

```
struct Insect : Decodable, Identifiable, Hashable{  
    var id: Int  
    var imageName:String  
    var name:String  
    var habitat:String  
    var description:String  
}
```

17. Open the `ContentView.swift` file located in the `macOS-Cross-Platform` folder. Add a `@State` variable to hold the selected insect from our insect list and an `@EnvironmentObject` variable to hold the insect data we'll be displaying:

```
 @State var selectedInsect: Insect?  
 @EnvironmentObject var userData: InsectData
```

18. Within the body, add a `NavigationView` and `VStack` that display the list of insects. Also, set the size of the view by adding a `.frame()` modifier at the end of the `VStack`:

```
 NavigationView{  
     VStack{  
         MacInsectListView(selectedInsect:  
             $selectedInsect)  
     }.frame(minWidth: 250, maxWidth: 400)  
 }
```

19. Below the `VStack` code, let's add an `if` statement to display the insect details within a `ScrollView` view. A `ScrollView` view is used because the content may be larger than the amount of screen space available since users control window sizes:

```
 if let selectedInsect = selectedInsect {  
     ScrollView{  
         InsectDetailView(insect: selectedInsect)  
     }  
 } else {  
     EmptyView()  
 }
```

20. To view the design in the canvas preview, add an `.environmentObject` modifier to the content preview:

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

The resulting ContentView preview should look as follows:

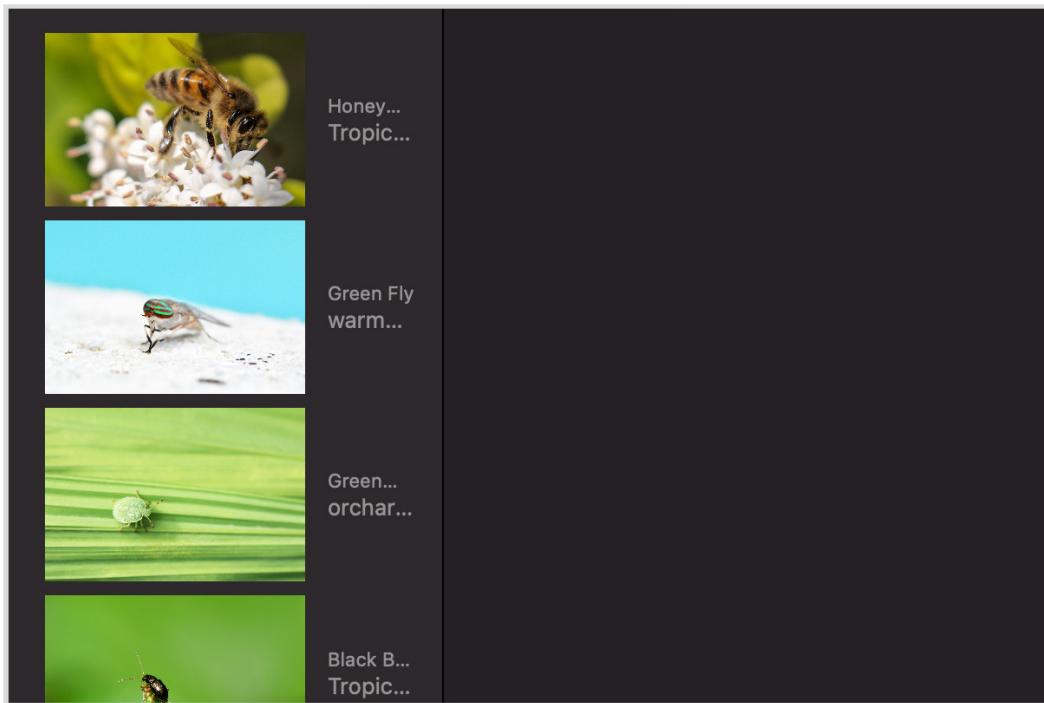


Figure 14.14 – ContentView preview at the end of the recipe

21. Finally, let's add a `.environmentObject()` modifier to the `ContentView()` function located in the `macOS_Cross_PlatformApp.swift` file in our `macOS-Cross-Platform` folder:

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

Adding the `.environmentObject()` modifier to `macOS_Cross_PlatformApp.swift` makes our data globally available to our app.

How it works...

When using SwiftUI code written for one platform across iOS platforms, we need to add targets for the new platforms in question. For example, in this recipe, we are building a macOS app from an existing iOS app. We, therefore, added the macOS app target to our Xcode project.

To run apps for a particular platform, we must first change Xcode's active scheme. So, for example, if we try running the code from the **macOS-Cross-Platform** section without changing the scheme first, we'll get an error. The error occurs because Xcode requires the scheme to match the type of code being run.

To reuse certain iOS files in macOS, we selected those files from the Navigator pane and checked the macOS target within the Inspector pane. Taking such steps allows us to reuse the code without copying it over. Any changes and updates to the code can thus be done in a single place.

To implement selection within our `MacInsectListView`, we made the `Insect` struct `Hashable` and added a `.tag()` modifier to our `MacInsectCellView`. Without the `.tag()` modifier, the app would still build and run, but we would be unable to select an item from the list to view its details.

Lastly, the `.listStyle(SidebarListStyle())` modifier in `MacInsectListView` is only available in macOS and causes a list to be displayed on the side, so that content can be displayed to the right of the list.

Creating the watchOS version of the iOS app

So far, we've created an iOS app and a macOS version of that app. Now, finally, we'll create the watchOS version. When creating apps for watchOS, we need to keep in mind that the screen space available on watches is small and make appropriate design choices based on that constraint.

In this recipe, we will create the watchOS version of our insect app.

Getting ready

Download the chapter materials from GitHub:

<https://github.com/PacktPublishing/SwiftUI-Cookbook-2nd-Edition-/tree/main/Chapter14-Cross-Platform-SwiftUI/03-Create-the-watchOS-version>

Open the `StartingPoint` folder and double-click on **Cross-Platform.xcodeproj** to open the app that we built in the preceding *Creating the macOS version of the iOS app* recipe of this chapter. We will be continuing from where we left off in the previous recipe.

The complete project can be found in the `Complete` folder to use as a reference.

How to do it...

We will create a watchOS app based on our initial iOS app by sharing some views and creating custom views where the original iOS views would not be appropriate. The steps are as follows:

1. Create a new **Target...** in Xcode:

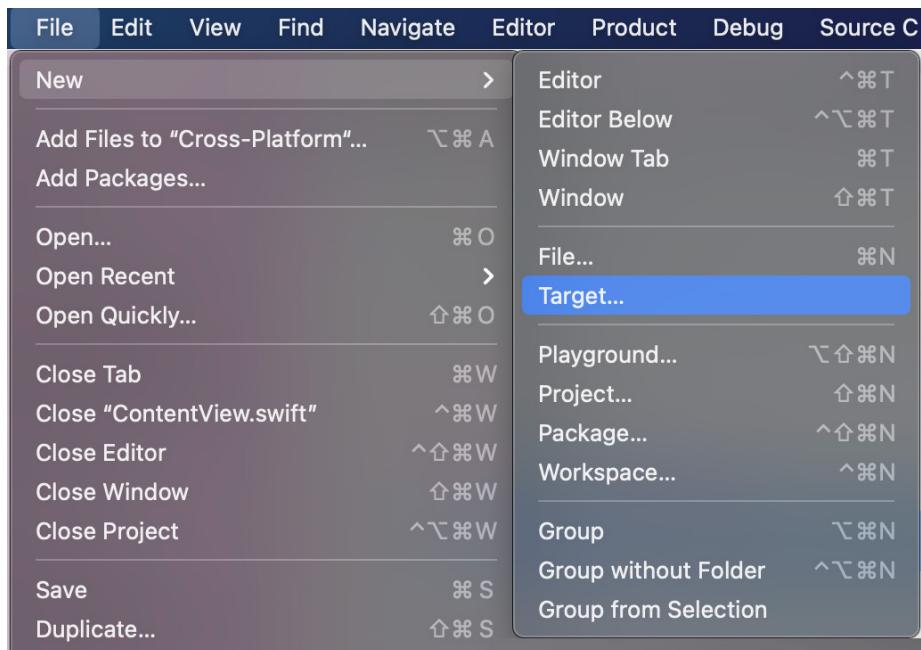


Figure 14.15 – Creating a new target

2. Choose the **watchOS** template, scroll down, and select **Watch App for iOS App**, then click **Next**:

Choose a template for your new target:

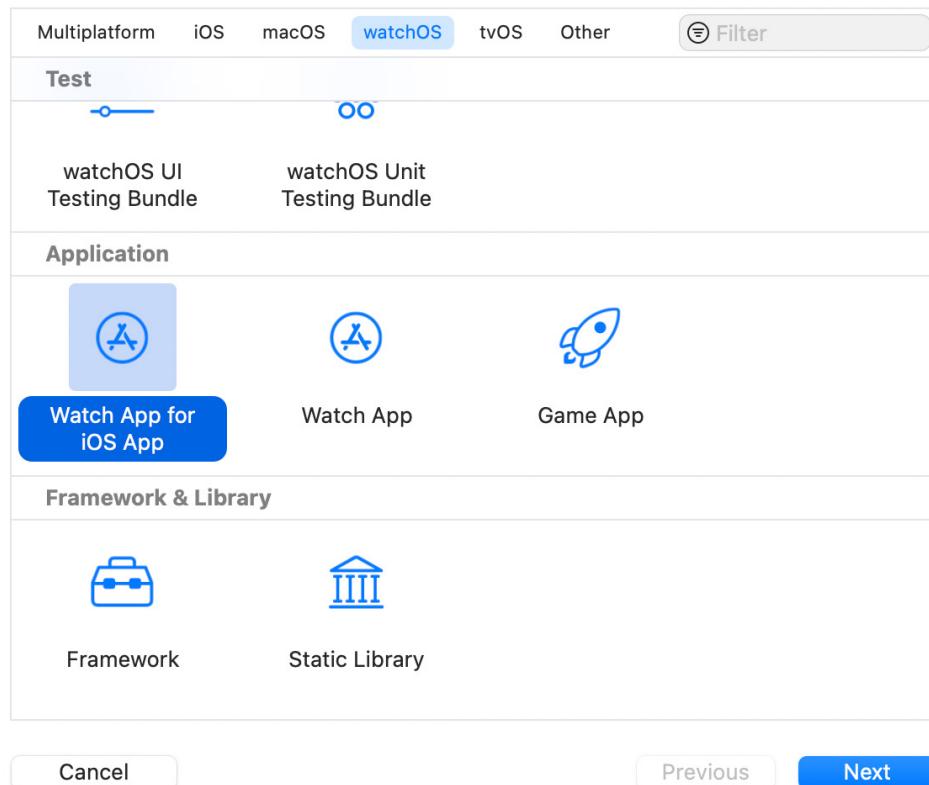


Figure 14.16 – Selecting watchOS app

3. In the next screen, enter the product name, `watchOS-Cross-Platform`.
4. Click **Activate** in the pop-up window that appears:

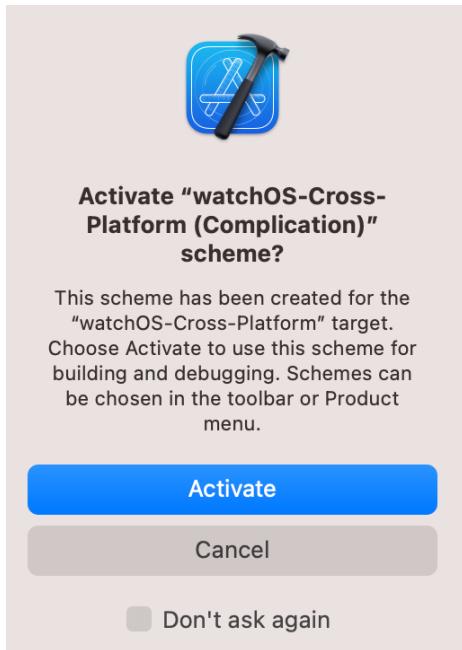


Figure 14.17 – watchOS Activate popup

5. The watchOS folders should appear in the navigation pane. Select the `ContentView.swift` file in the `watchOS-Cross-Platform Extension` folder. Run it in the Xcode canvas. The preview should look as follows:



Figure 14.18 – Initial watchOS preview

- For the next step, make sure the Navigator and Inspector panes are open. We will be using both panes to make certain files and resources available across targets:



Figure 14.19 – Navigator and Inspector panes in Xcode

- Let's share some files between the iOS and watchOS platforms. In the navigation pane, select the following files from our Cross-Platform folder: `Insect.swift`, `InsectData.swift`, `insectData.json`, `InsectCellView.swift`, `InsectListView.swift`, and `Assets.xcassets`.
- In the Inspector pane, check the **watchOS-Cross-Platform WatchKit Extension** checkbox to share the files between the iOS and watchOS versions of the app:

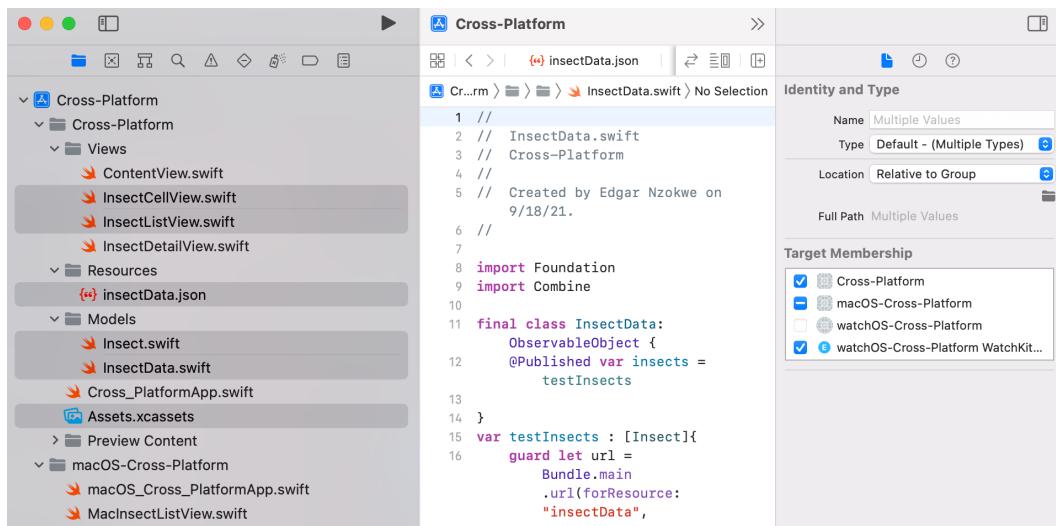


Figure 12.21 – Selecting multiple files in Xcode Navigator

- Create a new `WatchInsectDetailView` SwiftUI view in the `watchOS-Cross-Platform Extension` folder. Make sure the right platform target is selected (**watchOS-Cross-Platform Extension**):

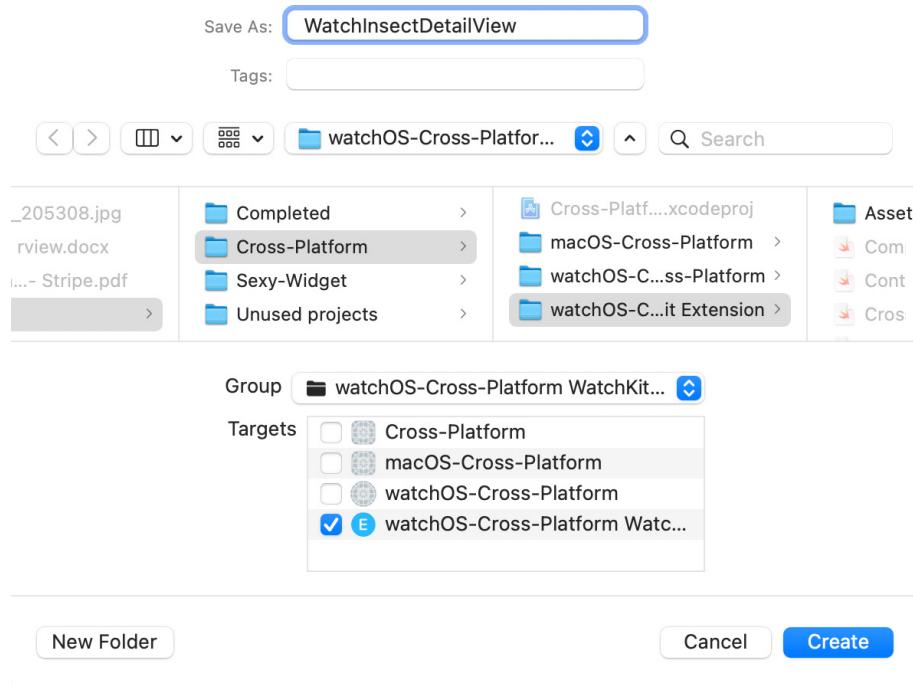


Figure 14.21 – Creating a new macOS view and selecting the right target

10. The `WatchInsectDetailView.swift` and `InsectCellView.swift` files have similar designs, but the former should use smaller font sizes to accommodate for the small display size of the Apple Watch. Replace the content of `WatchInsectDetailView` with the following:

```

var insect:Insect
var body: some View {
    VStack{
        Text(insect.name)
        Image(insect.imageName)
            .resizable()
            .aspectRatio(contentMode: .fit)
            .clipShape(Circle())
        HStack {
            Text("Habitat")
            Text(insect.habitat)
        }
    }
}

```

11. To preview the design in the Xcode canvas, add a sample insect to the preview function call:

```
struct WatchInsectDetailView_Previews: PreviewProvider {  
    static var previews: some View {  
        WatchInsectDetailView(insect: testInsect)  
    }  
}
```

12. Trying to build the project at this point fails. We get an **Unresolved identifier** error from `InsectListView.swift`. The error happens because `InsectDetailView.swift` is not available to our watchOS app. The `InsectListView.swift` file will need some tweaking.
13. From the `Cross-Platform | InsectListView.swift` file, change the `InsectListView` type declaration to make it generic:

```
struct InsectListView<DetailView: View>: View
```

14. Below the `@EnvironmentObject` declaration, add a property closure that creates the detail view:

```
let detailViewProducer: (Insect) -> DetailView
```

15. Use the `detailViewProducer` property to create detail views for insects. In the `NavigationLink` struct, change the destination to the following:

```
List{  
    ForEach(insectData.insects){insect in  
        NavigationLink(  
            destination: self.detailViewProducer(insect)  
            .environmentObject(self.insectData)){  
                InsectCellView(insect: insect)  
            }  
    }  
}
```

16. Now that we are using the same file across platforms, create a `typealias` that captures the platform being used:

```
#if os(watchOS)  
typealias PreviewDetailView = WatchInsectDetailView
```

```
#else
typealias PreviewDetailView = InsectDetailView
#endif
```

17. Modify the `InsectListView_Previews` to use the `typealias` for previews:

```
struct InsectListView_Previews: PreviewProvider {
    static var previews: some View {
        InsectListView{PreviewDetailView(insect:$0)}
            .environmentObject(InsectData())
    }
}
```

18. The `.navigationBarTitle()` modifier does not work on watchOS, so let's delete it.
19. Select `ContentView.swift` (from the `Cross-Platform` folder) and replace the `body` variable's content with a closure to create a detail view:

```
var body: some View {
    InsectListView{InsectDetailView(insect: $0) }
        .environmentObject(InsectData())
}
```

20. With all errors resolved, the `WatchInsectDetailView` from earlier can now be previewed. Click on `WatchInsectDetailView.swift` and open the canvas preview. The preview should look as follows:



Figure 14.22 – WatchInsectDetailView preview

21. You can reduce the font size of the insect name in `InsectCellView.swift` so that it works better with the small screen size. Remember that the change will be reflected in all other platforms where the `InsectCellView.swift` file is used.
22. Now, let's open the `ContentView.swift` file located in `watchOS-Cross-Platform Extension` and replace the `Text` view with our `InsectListView`:

```
var body: some View {
    InsectListView{WatchInsectDetailView(insect: $0) }
    .environmentObject(InsectData())
}
```

23. Add the `.environmentObject()` modifier to the preview:

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

Build and run the app or run the app live on the canvas. You should see a list of insects, and clicking on an insect should present the detail view for that insect.

How it works...

When building Cross-Platform iOS applications, it is important to first figure out how to make the app modular. Making the app modular by building smaller components improves reusability. For example, we used the `InsectCellView.swift` and `InsectListView.swift` files from the iOS app we built in our watchOS app. Selecting the files and then selecting **watchOS-Cross-Platform Extension** from the Xcode file inspector pane made the code from both files reachable from our watchOS code. However, adding `InsectListView.swift` to our watchOS target caused a compiling problem. Even though they share the same list view, clicking on a list item in iOS will lead to the iOS-specific details view `InsectDetailView`. In contrast, a click on the list in watchOS should open up the `watchInsectDetailView` struct. The issue was solved by changing the `InsectListView` type declaration to a generic type and adding a property closure that creates the detail view we need.

We also had to make some changes for the `InsectListView.swift` struct to enable the canvas preview. We added a conditional compilation to check the current debug operating system and set the `typealias` to the appropriate detail view.

When reusing the `InsectCellView`, we ran into a problem where some of the text was too big for the watchOS screen. Changing the text in `InsectCellView` would change the design of the iOS version too. If we decide to create a separate version of `InsectCellView`, we'll need to also create a different version of `InsectListView`, thereby repeating code in multiple locations.

Important Note

Xcode sometimes gets bogged down and will fail to build your code after you fix an error. When this happens, click on **Product** in the menu bar and **Clean Build Folder** ($\uparrow+\mathbb{H}+K$). Also, remember to clean your builds when it seems like code updates no longer fix existing issues.

15

SwiftUI Tips and Tricks

In the previous chapters, we tried to collect different problems, and therefore different recipes, grouping them together within a common theme. However, in this chapter, the recipes are not connected, apart from the fact that they are solutions for real-world problems.

We'll start viewing two ways of testing SwiftUI's views, using a visual technique called snapshot testing, and a more conventional one that inspects the views' hierarchy and structure.

Sometimes we must show some kind of documentation in the app, so we'll see how to present **Portable Document Format (PDF)** documents.

With Swift 1.0, Xcode introduced the possibility of creating coding with Playground, decreasing the feedback time from writing code to seeing its results and improving the productivity of mobile developers. We'll see a recipe on how to use SwiftUI in a Playground.

The default fonts provided by SwiftUI are pretty cool, but sometimes we need more flexibility. So, we'll see how to use custom fonts in SwiftUI.

Finally, we'll implement a Markdown editor that shows an attributed preview while adding a text with Markdown tags.

In this chapter, we will cover different topics in SwiftUI and real-world problems that you are likely to encounter during the development of your SwiftUI apps.

We will cover these topics in the following recipes:

- Snapshot testing SwiftUI views
- Unit testing SwiftUI with `ViewInspector`
- Showing a PDF in SwiftUI
- Implementing SwiftUI views using Playground
- Using custom fonts in SwiftUI
- Implementing a Markdown editor with Preview

Technical requirements

The code in this chapter is based on Xcode 13.

You can find the code in the book's GitHub repository under the following path:

<https://github.com/PacktPublishing/SwiftUI-Cookbook-2nd-Edition/tree/main/Chapter15-SwiftUI-Tips-and-Tricks/>.

Snapshot testing SwiftUI views

In this recipe, we are going to explore a powerful way of testing views: **snapshot testing**.

While snapshot testing is more common in other technologies—for example, JavaScript and the Jest testing library (<https://jestjs.io/docs/en/snapshot-testing.html>)—its usage in the iOS community is still a niche practice.

The aim of snapshot testing is to verify that the layout of a view is preserved while the code of the app is evolved and changed. To achieve this, we take a snapshot of the view that acts as a reference image.

A snapshot test consists of two steps:

- Taking a screenshot of a screen in a particular state
- Verifying that the screen in the same state has the same look as the screenshot taken in the previous step

Another name for snapshot testing is characterization testing. If you want to know more, here is a definition by Michael Feathers, the guru of improving legacy code: <https://michaelfeathers.silvrback.com/characterization-testing>.

In a SwiftUI implementation of snapshot testing, we take a snapshot of a component when we are sure that it looks correct; then, after any other change in our code, we run a check against that snapshot for the same component. If our code change made a change to the layout of the component, we are notified by a failed test. Otherwise, if the test succeeds, we are sure that our code changes didn't interfere with the layout of the component.

For this recipe, we'll use the **Swift Snapshot Testing** library, which you can find here: <https://github.com/pointfreeco/swift-snapshot-testing>.

Even though it still doesn't support SwiftUI natively, its composable architecture allows us to adapt it to SwiftUI without too much effort.

Getting ready

Let's start by creating an Xcode SwiftUI project called `SnapshotTestingSwiftUI`, remembering to enable the **Include Tests** checkbox, as shown in the following screenshot:

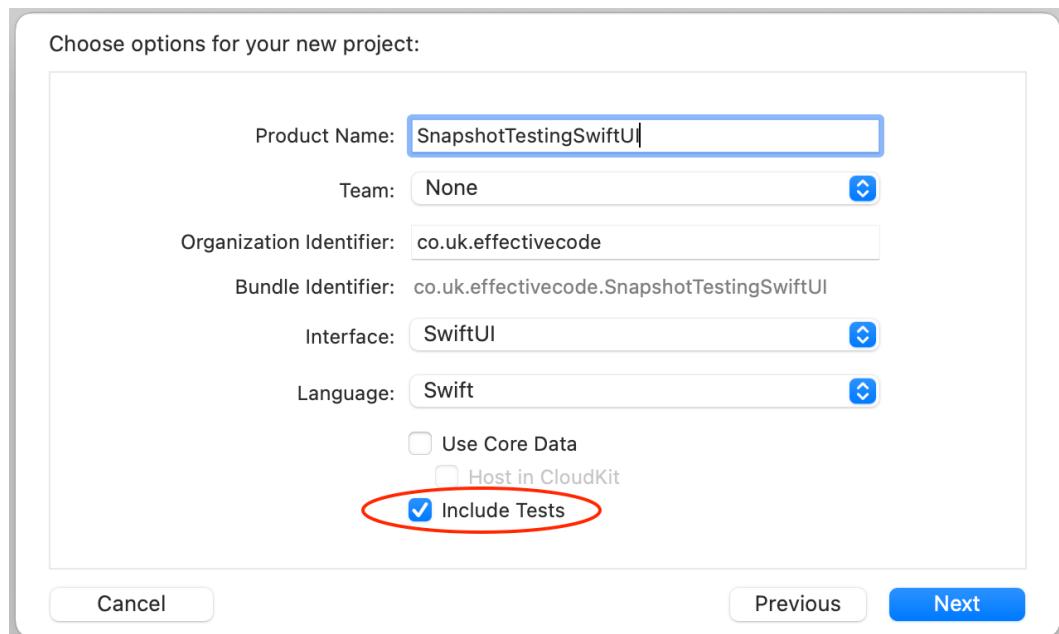


Figure 15.1 – SnapshotTestingSwiftUI app with unit tests enabled

Add the `swift-snapshot-testing` package, setting the URL to `https://github.com/pointfreeco/swift-snapshot-testing`:

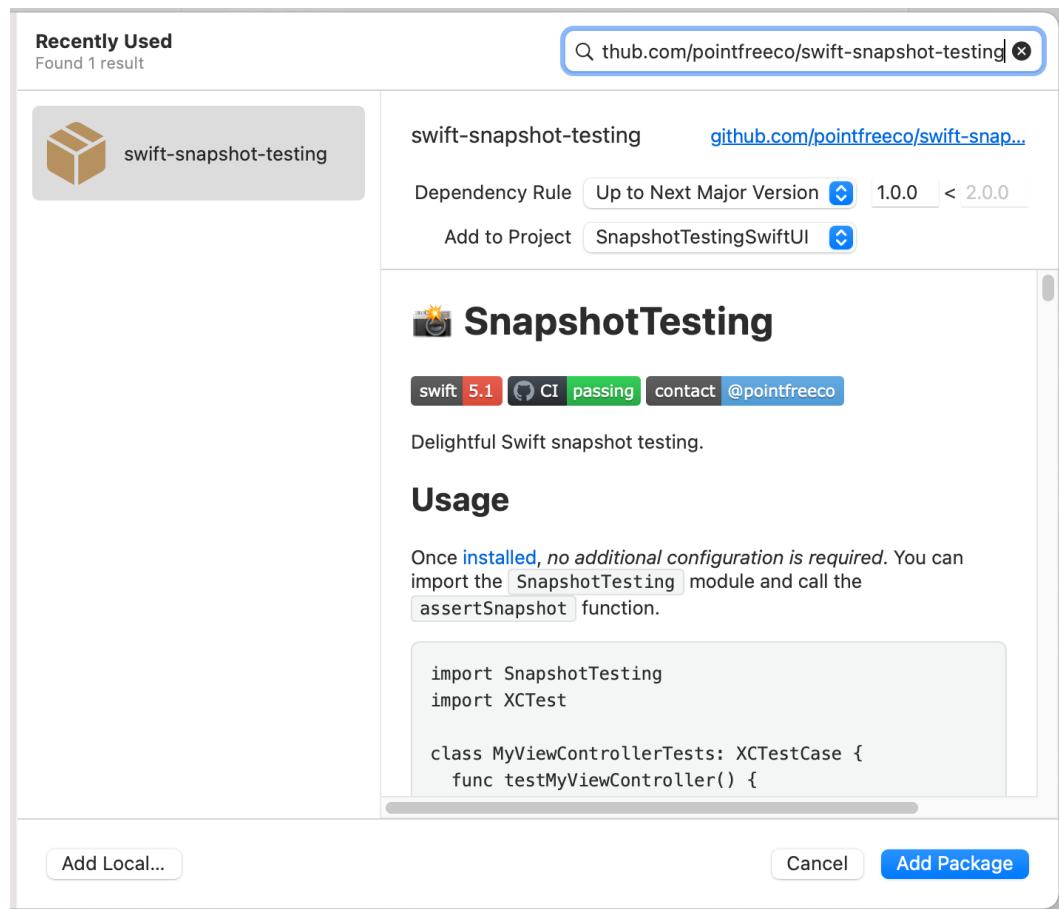


Figure 15.2 – Adding the package URL

On the final screen, make sure you add the package to the test target (*not* to the app target):

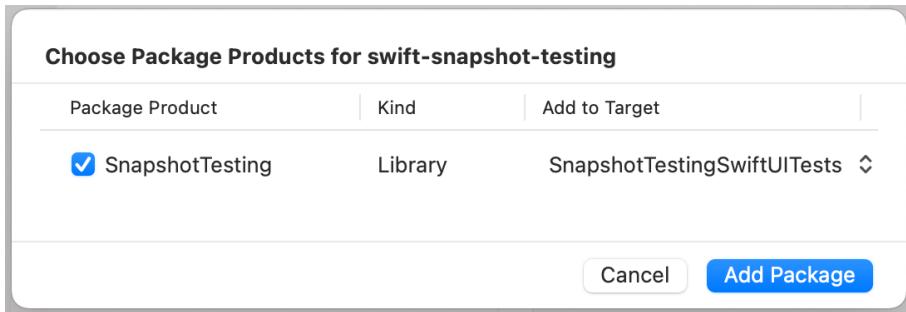


Figure 15.3 – Adding the package to the test target

We are now ready to prepare some tests.

How to do it...

We are going to implement a simple view with three cards, mimicking three different credit cards.

We will then run a snapshot test to take the golden plate, an image file that we expected to be correct, to protect us from accidentally changing that view.

Finally, we are going to change the order of the cards and see what happens when we run the test. Here are the steps:

1. Start by defining a list of cards:

```
struct ContentView: View {
    let cards: [(title: String, color: Color)] = [
        ("Visa Card", .yellow),
        ("Mastercard Credit Card", .red),
        ("Apple Credit Card", .black),
    ]
    //...
}
```

2. Then, for each card, create a rounded rectangle of the color defined in the list:

```
struct ContentView: View {
//...
var body: some View {
    VStack(spacing: 16) {
        ForEach(0..

```

```
Text(cards[index].title)
    .font(.system(.title))
    .frame(maxWidth: .infinity,
           maxHeight: .infinity)
    .foregroundColor(.white)
    .background(cards[index].color)
    .cornerRadius(16)
}
}.padding(.horizontal, 16)
}
}
```

We can now run the app to see a view with the three cards:

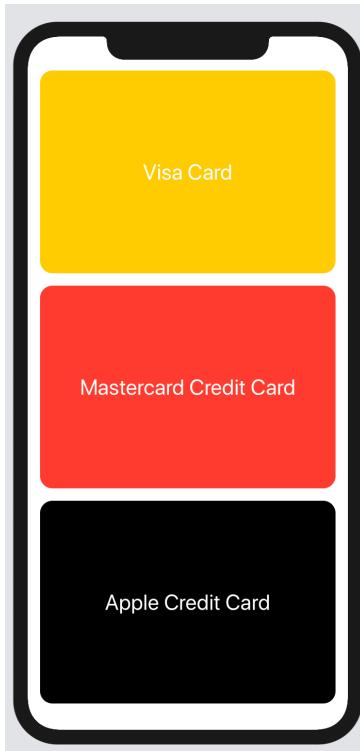


Figure 15.4 – The app with three credit cards

5. The first time we run the test, it fails since it doesn't find the reference image. While failing, it also takes a screenshot of the current state of the screen:

The screenshot shows a portion of an Xcode code editor. The code is a Swift file named `SnapshotTestingSwiftUITests.swift`. It contains a class `SnapshotTestingSwiftUITests` with an `setUp` method setting `diffTool` to `"ksdiff"`. A `testContentView` method is shown, which fails because no reference image was found on disk. A tooltip provides the path to the recorded screenshot: `/Users/giordanoscalzo/Dropbox/SwiftUIBook/Repo/SwiftUI-Cookbook-2nd-Edition-/Chapter15-SwiftUI-Tips-and-Tricks/01-Snapshot-testing-SwiftUI-views/SnapshotTestingSwiftUI/SnapshotTestingSwiftUITests/_Snapshots_/SnapshotTestingSwiftUITests/testContentView.1.png`. Below the tooltip, a message says "Re-run 'testContentView' to test against the newly-recorded snapshot."

```
15 // MARK: - Tests
16
17 class SnapshotTestingSwiftUITests: XCTestCase {
18     override class func setUp() {
19         diffTool = "ksdiff"
20     }
21
22     func testContentView() throws {
23         assertSnapshot(matching: ContentView(),
24
25             // MARK: - Failed
26             // MARK: - Description
27             // MARK: - Details
28             // MARK: - Screenshot
29         }
30     }
31 }
```

Figure 15.5 – Recording a reference image

Since a new screenshot was taken, the test now runs fine.

6. What would happen if we changed the order of the cards by mistake? Add the following code to simulate such a mistake:

```
struct ContentView: View {
    let cards: [(title: String, color: Color)] = [
        ("Apple Credit Card", .black),
        ("Visa Card", .yellow),
        ("Mastercard Credit Card", .red),
    ]
    //...
}
```

7. We run the test and it fails again since the current screen doesn't match the reference screenshot. The failure is explained in the following message:

```
✖ class SnapshotTestingSwiftUITests: XCTestCase {
21     override class func setUp() {
22         diffTool = "ksdiff"
23     }
24
25     func testContentView() throws {
26         assertSnapshot(matching: ContentView(),
27             ...
28     }
29 }
30
```

✖ failed - Snapshot does not match reference.

ksdiff "/Users/giordanoscalzo/Dropbox/SwiftUIBook/Repo/SwiftUI-Cookbook-2nd-Edition-/Chapter15-SwiftUI-Tips-and-Tricks/01-Snapshot-testing-SwiftUI-views/SnapshotTestingSwiftUI/SnapshotTestingSwiftUITests/_Schemas__/SnapshotTestingSwiftUITests/testContentView.1.png" "/Users/giordanoscalzo/Library/Developer/CoreSimulator/Devices/ED4CA109-2417-4999-8AC8-285751116470/data/Containers/Data/Application/55EACC90-C090-40C7-87F1-B02889D1063E/tmp/SnapshotTestingSwiftUITests/testContentView.1.png"

Newly-taken snapshot does not match reference.

Figure 15.6 – Failing snapshot test

8. This step is optional, but if we have **Kaleidoscope** (<https://www.kaleidoscopeapp.com>) installed, we can run the command indicated in the error message (`ksdiff .../path1/img.png .../path/img.png`) in Terminal to compare the two images:

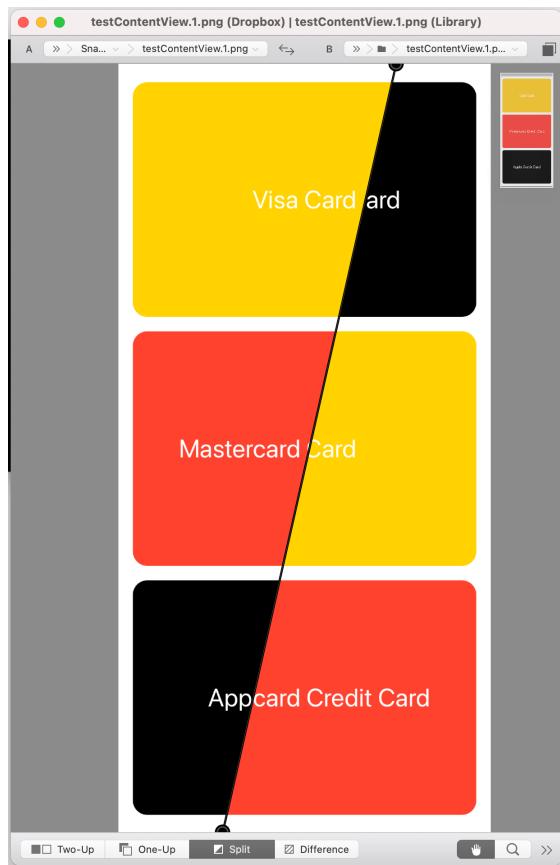


Figure 15.7 – Comparing test images

How it works...

Swift Snapshot Testing has a mechanism called `pullback`, which means that the test operation on the component we want to test, such as a SwiftUI View, is transformed into a test operation on something that the library already supports, such as a `UIViewController` or `UIImage`.

In our case, the component already supported is `UIViewController`, so the `pullback` bridges the View to `UIViewController`, embedding it in `UIHostingController`.

Regardless of the extension that you can copy in your code and use whenever you need to, it is essential to understand the process.

First, you must decide which device you will use for testing. This is because the screenshot density depends on the density of the device screen, and screenshots taken with an iPhone 13 Pro Max cannot be compared with those taken with an SE 2, for example. Then, you must run the tests once to create reference snapshots.

After that, you must check that the snapshots are what you expect. Human visual confirmation is essential because if a snapshot is wrong, we are going to release an app with an incorrect rendering. And because our tests will be green, we will think that our app is correct, whereas it will have some rendering errors. So, immediately check and fix any new failing test.

I think it is essential to have a way of comparing images to spot the difference. I'm pretty happy with Kaleidoscope (<https://www.kaleidoscopeapp.com>), but you can use whatever you want.

If you set the `diffTool` global variable, the content will be presented in the message of the failing test, as shown in the following screenshot:

```


1  class SnapshotTestingSwiftUITests: XCTestCase {
2     override class func setUp() {
3         diffTool = "MyDiffTool"
4     }
5
6     func testContentView() throws {
7         assertSnapshot(matching: ContentView(),
8
9             failed - Snapshot does not match reference.
10            MyDiffTool "/Users/giordanoscalzo/Dropbox/SwiftUIBook/Repo/SwiftUI-
11            Cookbook-2nd-Edition-/Chapter15-SwiftUI-Tips-and-Tricks/01-Snapshot-
12            testing-SwiftUI-views/SnapshotTestingSwiftUI/SnapshotTestingSwiftUITests/
13            __Snapshots__/SnapshotTestingSwiftUITests/testContentView.1.png" "/Users/
14            giordanoscalzo/Library/Developer/CoreSimulator/Devices/
15            ED4CA109-2417-4999-8AC8-28575116470/data/Containers/Data/
16            Application/E91A0DD1-7912-4BD7-9373-4E8EC27E811/tmp/
17            SnapshotTestingSwiftUITests/testContentView.1.png"
18
19            Newly-taken snapshot does not match reference.


```

Figure 15.8 – Custom diff images app

In general, snapshot testing not only gives you a safety net when you want to refactor some views, but it also has the effect of making you think of how to separate the presentation logic from the business logic, making the app more modular and thereby more flexible and adaptable to changes.

Unit testing SwiftUI with ViewInspector

Unit testing or test-driven development are not buzzwords anymore, and in the iOS ecosystem, they are almost taken for granted. Any iOS developer should know about test tools and how to apply them.

XCTest and **XCUITest** are mature frameworks, and so you'd expect to have something similar for SwiftUI.

Unfortunately, SwiftUI doesn't come with any test capabilities from Apple. To test a SwiftUI view, you could rely on either **UI** testing, which is flaky by nature, or use snapshot testing, as you can see in the *Snapshot testing SwiftUI views* recipe.

However, the beauty of open source is that given a problem, the community somehow finds a solution. This is the case with SwiftUI unit testing and **ViewInspector**, a framework for the inspection and testing of SwiftUI views.

In this recipe, we'll implement a simple SwiftUI view with some interaction, and we'll see how to test its structure and activity.

Getting ready

Let's start by creating an Xcode SwiftUI project called **TestingSwiftUI**, remembering to enable **Include Tests**, as shown in the following screenshot:

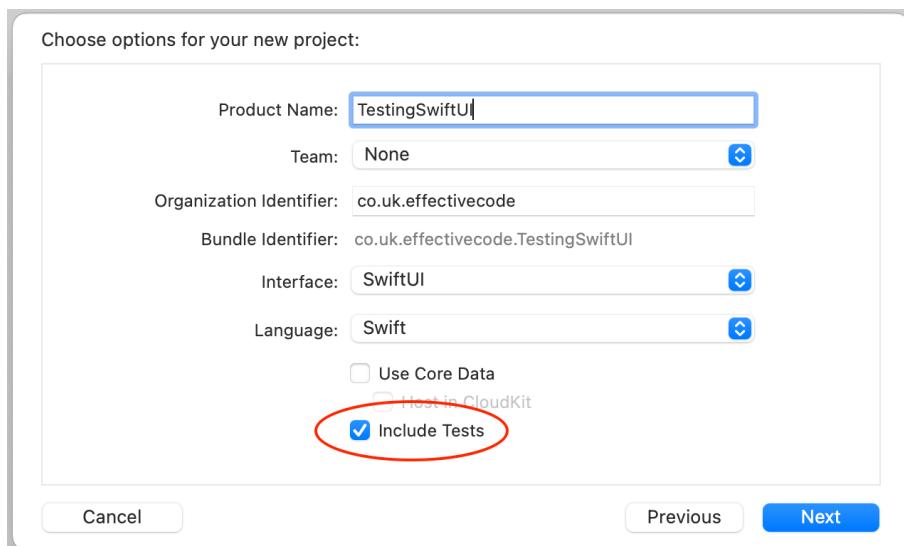


Figure 15.9 – TestingSwiftUI app with unit tests enabled

Add the `ViewInspector` package, setting the URL to `https://github.com/nalexn/ViewInspector`:

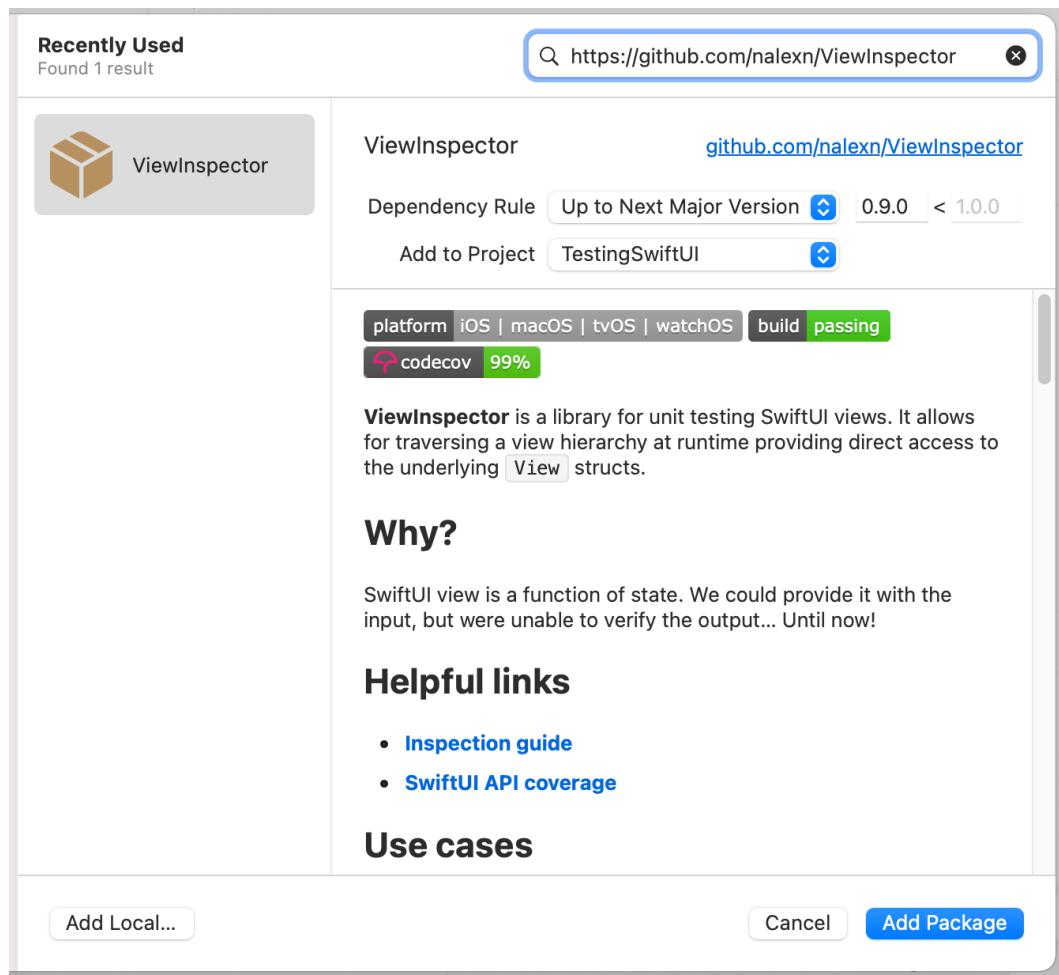


Figure 15.10 – Adding the package URL

On the final screen, make sure to add the package to the test target and not to the app target:

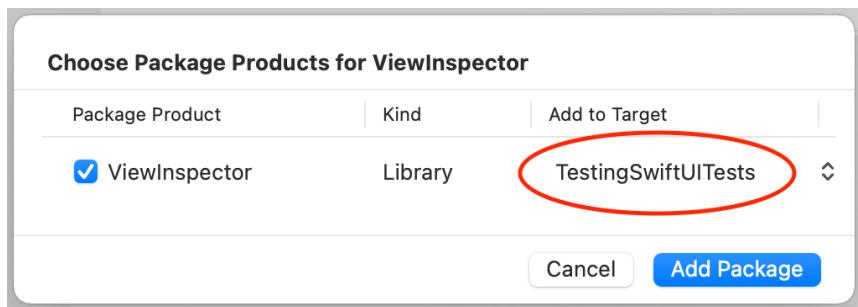


Figure 15.11 – Adding the package to the test target

We are now ready to prepare some tests.

How to do it...

We are going to implement an app that asks you for your country of origin—imagine that you are in an airport before entering a new country.

It presents a list of buttons with a few countries, and after selecting one, it changes the text at the center of the screen.

To do this, follow these steps:

1. Let's start by defining a list of possible countries we can select from, and adding a Text view in the middle of the page where we put the selected country:

```
struct ContentView: View {
    private let countries = [
        "USA",
        "France",
        "Germany",
        "Italy"
    ]
    @State
    var originCountry: Int = 0

    var body: some View {
        VStack(spacing: 12) {
```

```
    Text("What is your country of origin?")
    Spacer()
    Text(countries[originCountry])
        .font(.system(size: 40))
    Spacer()
}
}
}
```

2. Now, add a list of buttons to select the countries—one for each country:

```
 VStack(spacing: 12) {
    Text("What is your country of origin?")
    HStack(spacing: 12) {
        ForEach(0..
```

3. Inside each button, configure the Text object, changing the background color if it is the button currently selected:

```
 Button {
    originCountry = idx
} label: {
    Text(countries[idx])
        .frame(width: 80, height: 40)
        .background((originCountry == idx ?
            Color.red :
            Color.blue)
        .opacity(0.6))
```

```
.cornerRadius(5)  
.foregroundColor(.white)  
}
```

4. The app is now finished. As we start tapping the buttons, beginning from **USA** as the selected country, we see that the central text changes:

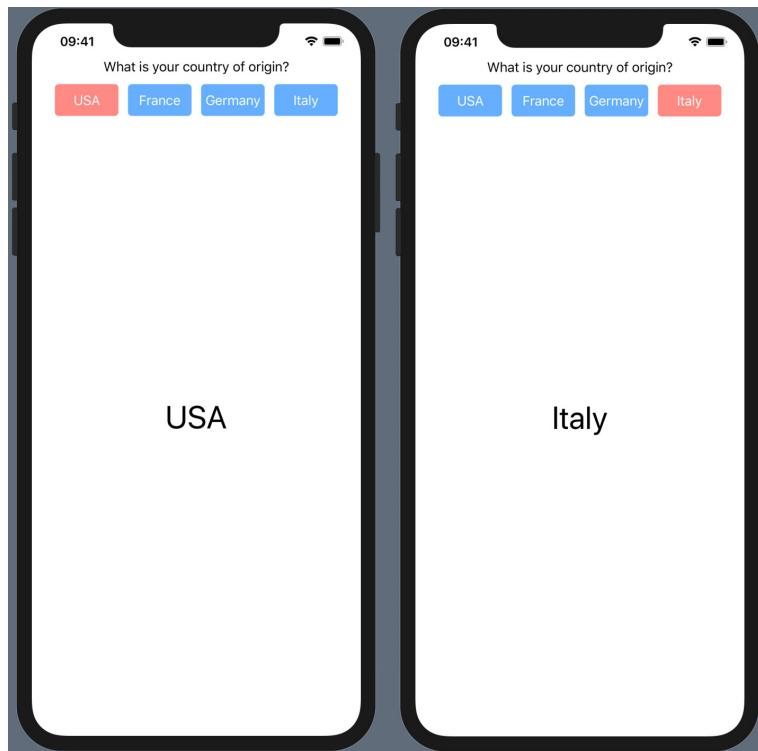


Figure 15.12 – Selecting a country of origin

5. Now, let's move on to the tests, where we prepare to import the frameworks and extend the `ContentView` struct with the `Inspectable` protocol:

```
import XCTest  
import SwiftUI  
import ViewInspector  
@testable import TestingSwiftUI  
  
extension ContentView: Inspectable { }
```

6. For the first test, we are going to check that the buttons contain the expected labels and that the central text is set to "USA":

```
class TestingSwiftUIAppTests: XCTestCase {
    func testStartWithUSASelected() throws {
        let view = ContentView()

        let buttons = try
            view.inspect().vStack().hStack(1).forEach(0)
        XCTAssertEqual(try
            buttons.button(0).labelView()
            .text().string(), "USA")
        XCTAssertEqual(try
            buttons.button(1).labelView()
            .text().string(), "France")
        XCTAssertEqual(try
            buttons.button(2).labelView()
            .text().string(), "Germany")
        XCTAssertEqual(try
            buttons.button(3).labelView()
            .text().string(), "Italy")

        let country = try
            view.inspect().vStack().text(3)
        XCTAssertEqual(try country.string(), "USA")
    }
}
```

7. The next aspect we want to test is that after selecting a country, the Text view at the center of the screen changes its label to the name of the selected country. This feature relies on a `@State` variable. Unfortunately, `@State` variables aren't currently supported by `ViewInspector` out of the box, and we must slightly change the `ContentView` code to overcome this limitation. In particular, we must create a function that *captures* the `body` variable when it is created to be inspected in the test:

```
struct ContentView: View {
    //...
```

```
var didAppear: ((Self) -> Void)?  
  
var body: some View {  
    VStack(spacing: 12) {  
        //...  
    }  
    .onAppear { didAppear?(self) }  
}
```

8. With this closure in place, we can write a test that assures that the central text is in sync with the selected button:

```
func testSelectItaly() throws {  
    var view = ContentView()  
  
    let exp = view.on(\.didAppear) { view in  
        XCTAssertEqual(try  
            view.actualView().originCountry, 0)  
        try  
            view.actualView().inspect().vStack()  
            .hStack(1).forEach(0).button(3).tap()  
        XCTAssertEqual(try  
            view.actualView().originCountry, 3)  
    }  
  
    ViewHosting.host(view: view)  
    wait(for: [exp], timeout: 0.1)  
}
```

We can run the tests now, and they will both be green.

How it works...

The important thing here is to extend the `View` we want to test with the `Inspectable` protocol, which adds a few functions to inspect the content of the view.

The added `inspect()` function returns a `struct` that mimics the actual view structure and that we can use to extract the sub-views. In our case, `VStack` contains `Text`, `HStack`, `Spacer`, and another `Text` struct. They can be reached using the proper function and the related index, like this:

```
try view.inspect().vStack().text(0)
try view.inspect().vStack().hStack(1)
try view.inspect().vStack().spacer(2)
try view.inspect().vStack().text(3)
```

Regarding testing the interactive behavior, this is not so easy to do. Therefore, we added a closure to capture the actual view that can be used in a block-testing function:

```
let exp = view.on(\.didAppear) { view in
    XCTAssertEqual(try view.actualView().originCountry, 0)
    try view.actualView().inspect().vStack().hStack(1)
        .forEach(0).button(3).tap()
    XCTAssertEqual(try view.actualView().originCountry, 3)
}
```

The `.actualView()` function returns the view that was captured in the callback we added to our code. I admit that the tests look a bit convoluted; however, `ViewInspector` is a helpful tool to have in your toolbox, ready to be used when you want to inspect and test your SwiftUI view layout structure.

See also

`ViewInspector` is a useful library written by Alexey Naumov, who also has a blog dedicated to SwiftUI at this link: <https://nalexn.github.io>.

The `ViewInspector` repository also hosts a user guide: <https://github.com/nalexn/ViewInspector/blob/master/guide.md>.

Also, a list of supported SwiftUI API can be found here: <https://github.com/nalexn/ViewInspector/blob/master/readiness.md>.

Showing a PDF in SwiftUI

Since iOS 11, Apple has provided **PDFKit**, a robust framework to display and manipulate PDF documents in your applications.

As you can imagine, PDFKit is based on UIKit. However, in this recipe, we'll see how easy it is to integrate it with SwiftUI.

Getting ready

Let's create a new SwiftUI app in Xcode called **PDFReader**.

For this, we need a PDF document to present. We provide an example in the <https://github.com/PacktPublishing/SwiftUI-Cookbook-2nd-Edition-/blob/main/Resources/Chapter15/recipe3/PDFBook.pdf> repository but feel free to use the PDF document of your choice.

Copy the document into the project:

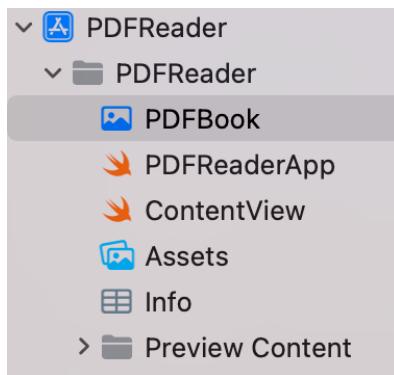


Figure 15.13 – PDF document in the Xcode project

With the PDF document in the project, we are ready to implement our PDF viewer in SwiftUI.

How to do it...

PDFKit provides a class called **PDFView** to render a PDF document.

Because the **PDFView** class is a subclass of **UIView**, it must be encapsulated in a **UIViewRepresentable** class to be used in SwiftUI.

We are going to implement a `PDFKitView` view to represent a viewer for a PDF document. That view will then be used in other SwiftUI views to present a PDF document. Follow these steps:

1. Let's define a struct called `PDFKitView` to bridge `PDFView` to SwiftUI:

```
import SwiftUI
import PDFKit

struct PDFKitView: UIViewRepresentable {
    let url: URL

    func makeUIView(context: UIViewControllerRepresentableContext<PDFKitView>) -> PDFView {
        let pdfView = PDFView()
        pdfView.document = PDFDocument(url: self.url)
        return pdfView
    }

    func updateUIView(_ uiView: PDFView, context: UIViewControllerRepresentableContext<PDFKitView>) {
    }
}
```

2. The newly created component can then be used anywhere in our SwiftUI code. Add it to the `ContentView`:

```
struct ContentView: View {
    let documentURL = Bundle.main.url(forResource: "PDFBook",
                                        withExtension: "pdf")!
    var body: some View {
        VStack(alignment: .leading) {
            Text("The Waking Lights")
                .font(.largeTitle)
```

```
Text ("By Urna Semper")
    .font (.title)
    PDFKitView(url: documentURL)
}
}
```

As you can see on running the app, the PDF document is accurately rendered in our PDFKitView component:

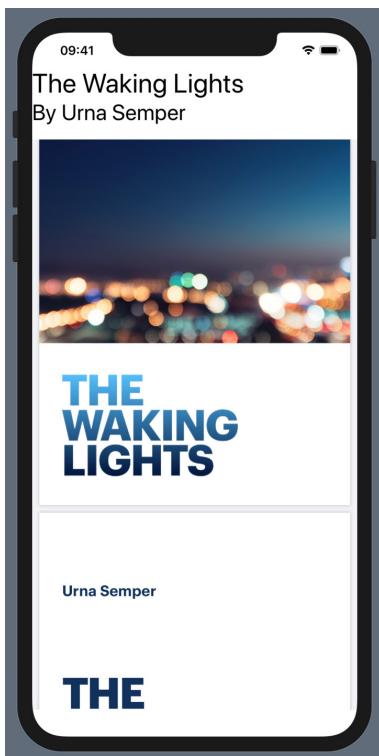


Figure 15.14 – PDF document rendered in SwiftUI

How it works...

This recipe is straightforward, thanks to its encapsulated nature.

We firstly define a PDFKitView struct extending `UIViewRepresentable`, which creates and holds an instance of the `PDFView` UIKit class. After creating it, we set the URL of the document to present in the View. This PDFKitView view is then used in the `ContentView` as a normal SwiftUI view.

Even though the sample PDF document we are using is local, the UIKit `PDFView` class renders a document from a URL. So, we must provide the local document using a URL.

However, this gives us the possibility of having a remote document: why not try to present the document from a remote location? Would you make any code changes?

There's more...

The use we make of the `UIKit PDFView` class is elementary, but it can do more, such as zooming, navigating through pages, highlighting sections, and so on.

As an exercise, I suggest that you start from this recipe and implement a custom fullscreen PDF reader, where we present a page in fullscreen and can navigate through each page using custom buttons. It should be a matter of exposing the `PDFView` class functions in `PDFKitView` and connecting them to normal SwiftUI buttons.

Implementing SwiftUI views using Playground

One of the most surprising and useful innovations brought by Swift is the **Playground**, a simple file that contains Swift code and that can be seen and modified in real time without waiting for the full cycle of a rebuild.

Again, Playground supports UIKit, but since SwiftUI can be quickly embedded in a `UIKit` component, we can use Playground for rapidly prototyping SwiftUI views too.

Getting ready

Create a Playground file (**File | New | Playground**) in Xcode called `SwiftUIPlayground`.

How to do it...

The power of Playground is that you can make a change and immediately see the result. Importing the `PlaygroundSupport` framework, you can also render a `UIViewController` class to be shown during the Playground session.

We are going to define a simple SwiftUI view to be rendered in Playground. Follow these steps:

1. Start by importing the needed frameworks:

```
import PlaygroundSupport  
import SwiftUI
```

2. Create an extension to the Text component to customize its appearance:

```
extension Text {  
    func customize(_ color: Color) -> some View {  
        self  
            .font(.system(.title))  
            .frame(width: 300, height: 150)  
            .foregroundColor(.white)  
            .background(color)  
            .cornerRadius(10)  
    }  
}
```

3. With this handy extension ready, create a simple View in the ContentView:

```
struct ContentView: View {  
    var body: some View {  
        VStack(spacing: 12) {  
            Text("SwiftUI")  
                .customize(.yellow)  
            Text("in a")  
                .customize(.green)  
            Text("Playground!")  
                .customize(.red)  
        }  
    }  
}
```

4. The ContentView struct we created can be shown in Playground, as follows:

```
let viewController = UIHostingController(  
    rootView: ContentView())  
  
PlaygroundPage.current.liveView = viewController
```

Running Playground by tapping on the arrow at the bottom left, we can run the code and see the SwiftUI view we implemented:

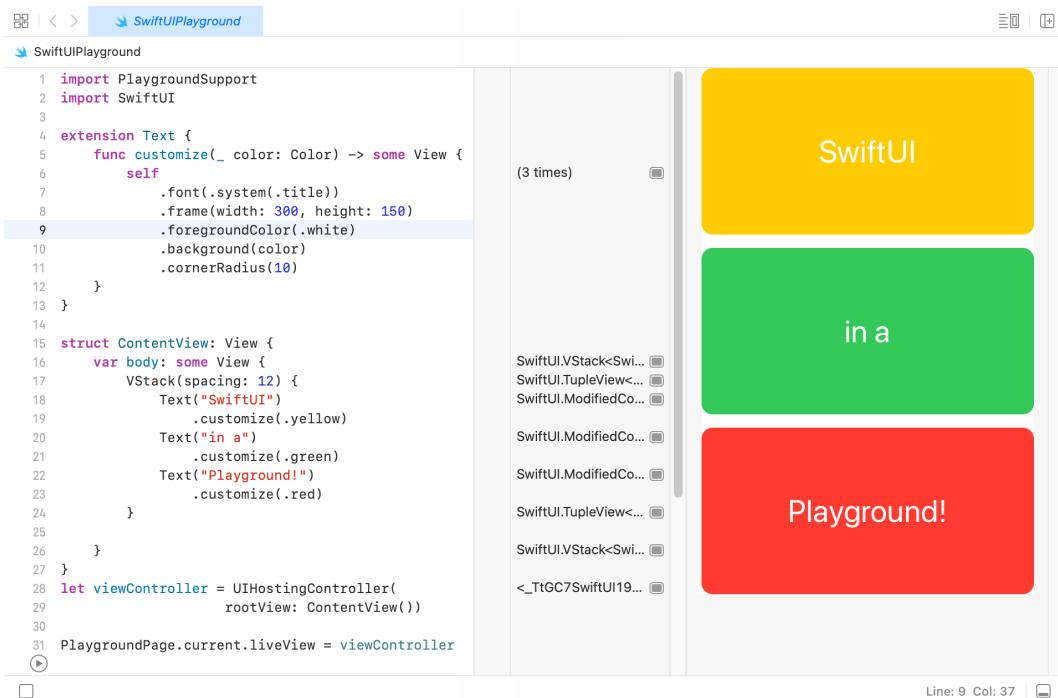


Figure 15.15 – A SwiftUI view in a Playground

How it works...

Playground supports rendering a `UIViewController` class via the `liveView` property of the `PlaygroundPage` class.

With the `UIHostingController` class, we can embed a SwiftUI view in a `UIViewController` class. Using these two functionalities, we can then use Playground and SwiftUI together.

Now that you have the necessary code, you could change the colors or the label, and see the changes in real time.

Using custom fonts in SwiftUI

iOS comes with many preinstalled fonts that can be safely used in SwiftUI. However, the design sometimes needs some custom fonts because they might be a part of the brand of the app maker, for example.

In this recipe, we'll see how to import a couple of custom fonts to use for building a menu page of an Italian restaurant.

Getting ready

Create a SwiftUI app called `RestaurantMenu`.

Copy the *Sacramento*, *Oleo Script Regular*, and *Oleo Script Bold* custom fonts, which you can find in the repository at <https://github.com/PacktPublishing/SwiftUI-Cookbook-2nd-Edition/tree/main/Resources/Chapter15/recipe5>.

The fonts must be added to the project:

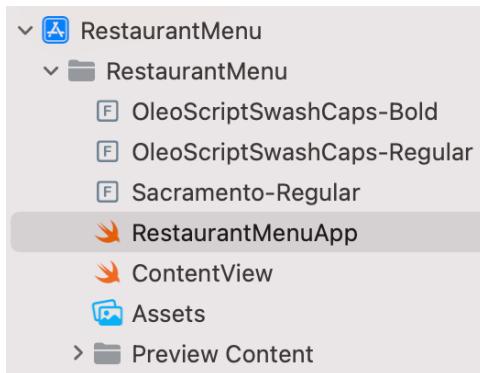


Figure 15.16 – Custom fonts in the project

Finally, the fonts must be copied during the installation, so add them in the **Copy Bundle Resources** phase:

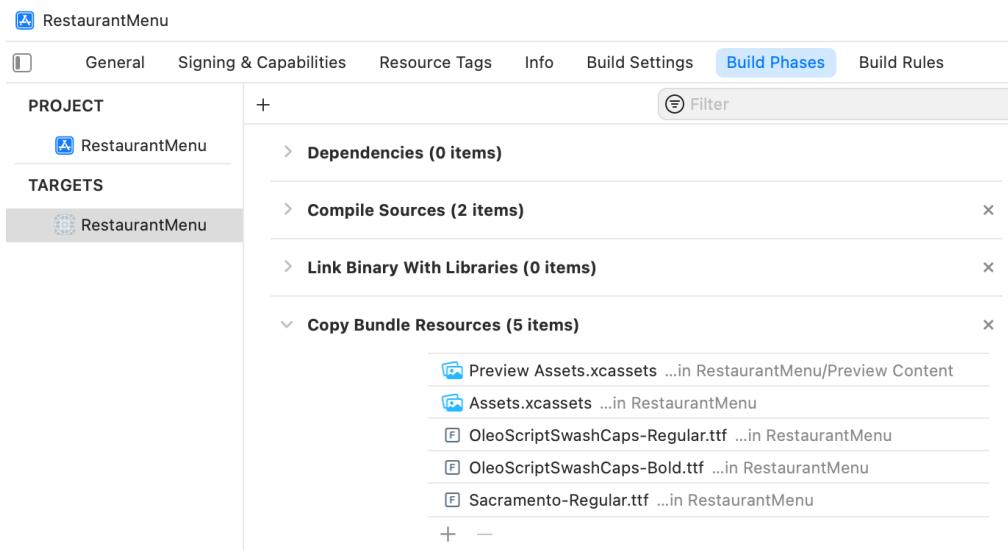


Figure 15.17 – Copying fonts during the installation

How to do it...

Besides the *semantic* fonts, such as *title*, *caption*, *footnote*, and so on, SwiftUI allows us to define other fonts using the static `.custom()` function.

We'll use it for our custom fonts. Follow these steps:

1. Implement the static functions to return the custom fonts:

```
extension Font {  
    static func oleoBold(size: CGFloat) -> Font {  
        .custom("OleoScriptSwashCaps-Bold", size: size)  
    }  
    static func oleoRegular(size: CGFloat) -> Font {  
        .custom("OleoScriptSwashCaps-Regular", size: size)  
    }  
    static func sacramento(size: CGFloat) -> Font {  
        .custom("Sacramento-Regular", size: size)  
    }  
}
```

2. Create a view to present an entry in the menu:

```
struct ItemView: View {  
    let name: String  
    let price: String  
  
    var body: some View {  
        HStack {  
            Text(name)  
            Spacer()  
            Text(price)  
        }  
        .font(.sacramento(size: 40))  
    }  
}
```

3. Add the menu items to the ContentView:

```
struct ContentView: View {  
    var body: some View {
```

```
    VStack {
        Text("Casa Mia")
            .font(.oleoBold(size: 80))
        Text("Restaurant")
            .font(.oleoRegular(size: 60))
        ItemView(name: "Pizza Margherita", price: "$10")
        ItemView(name: "Fettuccine Alfredo", price: "$14")
        ItemView(name: "Pollo Arrosto", price: "$19")
        ItemView(name: "Insalata Caprese", price: "$12")
        ItemView(name: "Gelato", price: "$9")
    }
    .padding(.horizontal)
}
}
```

4. Running the app, however, we see that the custom fonts aren't used. To make them work, we must register the `.ttf` (**True Type Font**) files. Add the following function to the `@main` struct:

```
private func loadFonts(withExtension ext: String) {
    let fonts = Bundle.main
        .urls(forResourcesWithExtension: ext,
              subdirectory: nil)
    fonts?.forEach { url in
        print(url)
        var errorRef: Unmanaged<CFError>?
        CTFontManagerRegisterFontsForURL(url as CFURL,
                                         .process, &errorRef)
        if let errorRef = errorRef {
            let error = errorRef.takeRetainedValue()
            print(error)
        }
    }
}
```

- Finally, call the function in the `init()` of the `@main` struct:

```
@main
struct RestaurantMenuApp: App {
    init() {
        loadFonts(withExtension: "ttf")
    }
    //...
}
```

Running the app, we can see the sleek interface created by the custom fonts:



Figure 15.18 – A restaurant menu page

How it works...

SwiftUI provides a list of semantic fonts, such as *body*, *callout*, and so on, that allow the styling of different parts of a page with a lot of text.

It also has a `static` function called `custom()` whose parameters are the name of the font and its size. This function returns a font read from the custom font built with the app, which is either a `.ttf` or `.otf` (**Open Type Font**) file.

Passing the name of the font every time we have to use it can be error-prone, so it is better to create a `static` function in an extension of the `Font` class. This way, it can be statically checked during the compilation, reducing the possibility of errors.

Implementing a Markdown editor with Preview

Markdown is a lightweight markup language to create rich text using a text editor. In recent years, it has become ubiquitous: you can use it in GitHub, when asking and replying to questions in Stack Overflow, for messaging in Discord, and more.

SwiftUI provides support to Markdown tags in the `Text` components.

In this recipe, we'll experiment with Markdown, implementing a simple text editor.

Getting ready

This recipe doesn't require any particular preliminary operations apart from creating an Xcode project called `MarkdownEditor`.

How to do it...

Taking inspiration from the GitHub pull request comment textbox, we are going to implement an app with two tabs: a text editor and a preview where the text is rendered as rich text, with bold, italics, and so on.

`EditorView` has a tab bar above the editor itself for adding tags for bold, italics, strikethrough, and code.

Let's get started. Follow these steps:

1. Start by creating `EditorView`:

```
struct EditorView: View {  
    @Binding var text: String
```

```
var body: some View {
    VStack(alignment: .leading) {
        Tabbar(text: $text)
        TextEditor(text: $text)
            .font(.title)
    }
}
```

2. Create a Tabbar view, containing a list of buttons:

```
struct Tabbar: View {
    @Binding var text: String
    var body: some View {
        HStack {
            TabButton(label: "***B***") {
                text+="***"
            }
            TabButton(label: "*I*") {
                text+="*"
            }
            TabButton(label: "****B****") {
                text+="****"
            }
            TabButton(label: "~~S~~") {
                text+="~~"
            }
            TabButton(label: "'C'") {
                text+="'"
            }
        }
    }
}
```

3. Add a TabButton view, customizing the button's appearance and passing a callback to be called when the button is activated:

```
struct TabButton: View {  
    let label: String  
    let onTap: () -> Void  
  
    var body: some View {  
        Button {  
            onTap()  
        } label: {  
            Text(.init(label))  
                .frame(width:30)  
        }  
        .buttonStyle(.bordered)  
    }  
}
```

4. Create a PreviewView to render the rich text from the Markdown written in the editor:

```
struct PreviewView: View {  
    var text: String  
    var body: some View {  
        VStack {  
            Text(.init(text))  
                .font(.title)  
                .frame(maxWidth: .infinity,  
                       alignment: .leading)  
            Spacer()  
        }  
    }  
}
```

5. Finally, in the ContentView, add the following code to show the editor and the preview views:

```
struct ContentView: View {  
    @State private var tabShown = 0  
    @State private var text = ""  
  
    var body: some View {  
        VStack {  
            Picker("", selection: $tabShown) {  
                Text("Editor").tag(0)  
                Text("Preview").tag(1)  
            }  
            .pickerStyle(.segmented)  
            if tabShown == 0 {  
                EditorView(text: $text)  
            } else {  
                PreviewView(text: text)  
            }  
        }  
        .padding(.horizontal)  
    }  
}
```

Running the app, we can write a text with Markdown tags and visualize it as rich text:

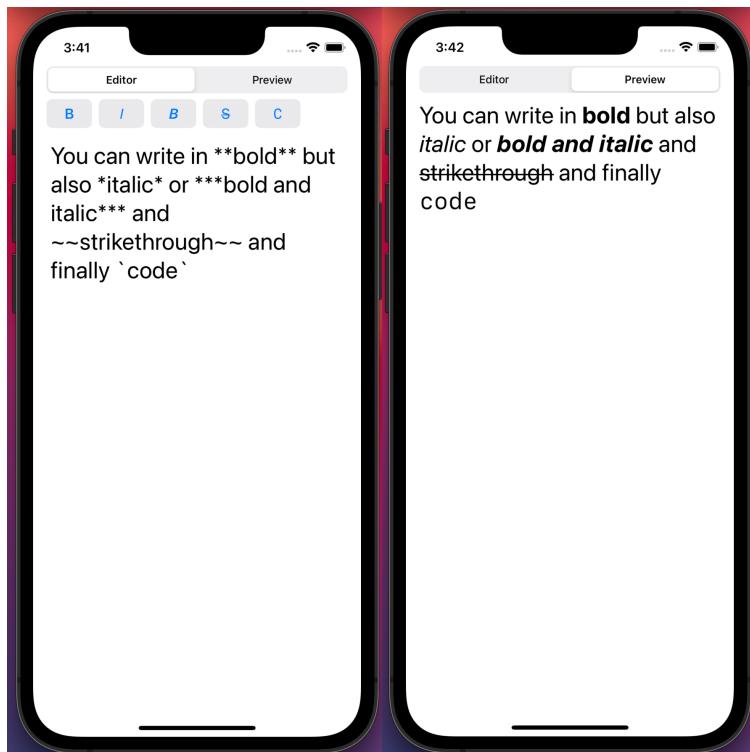


Figure 15.19 – A Markdown editor and a text preview

How it works...

Markdown is a simple language whereby you can describe the style of text, surrounding it with a tag. In our app, we are supporting the following tags:

- ****text**** for bold text
- ***text*** for italic text
- *****text***** for bold and italic text
- **~~text~~** for strikethrough text
- **'text'** for a monospaced text

EditorView presents a list of buttons to add an initial or final tag to the `text` variable. However, the tag can be simply written as normal text and everything would work.

From iOS 15, the `Text` component supports text with Markdown tags. However, to make it work, it must use an initializer that expects a `StringLocalizedKey`. The `EditorView` provides a `String` variable, but we are converting it to a `StringLocalizedKey` when initializing the `Text` with the code `Text(.init(text))`: in this way, the Markdown tags are correctly rendered.

Hi!

We're Giordano and Edgar, the authors of SwiftUI Cookbook Second Edition. We really hope you enjoyed reading this book and found it useful for increasing your productivity and efficiency in SwiftUI.

It would really help us (and other potential readers!) if you could leave a review on Amazon sharing your thoughts on SwiftUI Cookbook Second Edition here.

Go to the link below or scan the QR code to leave your review:

<https://packt.link/r/1803234458>



Your review will help us to understand what's worked well in this book, and what could be improved upon for future editions, so it really is appreciated.

Best Wishes,



Giordano Scalzo



Edgar Nzokwe



Packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

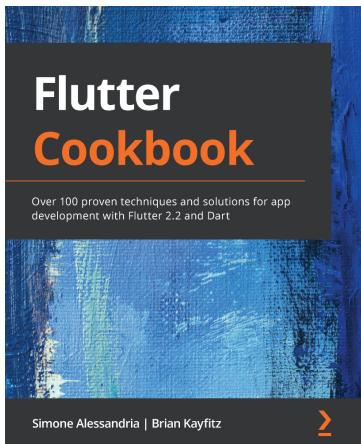
- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

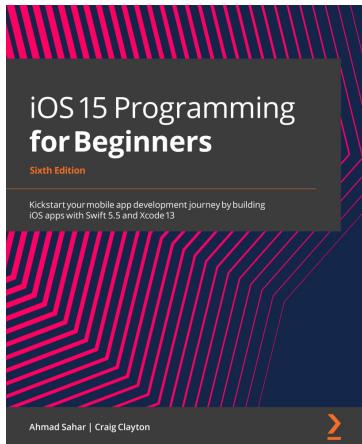


Flutter Cookbook

Simone Alessandria, Brian Kayfitz

ISBN: 978-1-83882-338-2

- Use Dart programming to customize your Flutter applications
- Discover how to develop and think like a Dart programmer
- Leverage Firebase Machine Learning capabilities to create intelligent apps
- Create reusable architecture that can be applied to any type of app
- Use web services and persist data locally
- Debug and solve problems before users can see them
- Use asynchronous programming with Future and Stream
- Manage the app state with Streams and the BLoC pattern



iOS 15 Programming for Beginners

Ahmad Sahar, Craig Clayton

ISBN: 978-1-80181-124-8

- Get to grips with the fundamentals of Xcode 13 and Swift 5.5, the building blocks of iOS development
- Understand how to prototype an app using storyboards
- Discover the Model-View-Controller design pattern and how to implement the desired functionality within an app
- Implement the latest iOS features such as Swift Concurrency and SharePlay
- Convert an existing iPad app into a Mac app with Mac Catalyst
- Design, deploy, and test your iOS applications with design patterns and best practices

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Index

Symbols

@Binding property wrapper
using, to pass state variable to child view 305-310

@EnvironmentObject property wrapper
used, for sharing state objects with multiple views 321-328
working 329

@FetchRequest
Core Data objects, displaying with 484-492

.frame modifier
using 7

.matchedGeometryEffect
hero view transition, creating with 263-271

@ObservedObject property wrapper
CoreLocation wrapper, implementing 310-316

.searchable modifier 80

@SectionedFetchRequest
used, for data presentation in sectioned list 507-511

@StateObject property wrapper
using, to preserve model life cycle 317-320

@State property wrapper
using, to drive view's behavior 301-304

.tabViewStyle() 161

A

actions
adding, to alert buttons 170-173

advanced pickers
using 23-27

alert buttons
actions, adding to 170-173
reference link 170

alerts
presenting 166-169

angular gradient 218

app
debugging, based on Combine 376-381
unit testing, based on Combine 382-390

async await
completion block function,
converting to 407-413

infinite scrolling, implementing with 413-422

async function
integrating 392-396

B

banner
 creating, with spring animation 246-249
bar chart
 building 219-223
basic animations
 creating 236-242
buttons
 adding 18-22
 navigating with 18-22

C

Canvas API
 drawing with 200-203
Cloud Firestore 461
Combine
 app, debugging based on 376-381
 app, unit testing based on 382-390
 form, validating with 352-362
 in SwiftUI project 333-342
 memory, managing to build
 timer app 345-351
Publishers 343
subscription 343, 344
used, for fetching remote data 363-376
completion block function
 converting, to async await 407-413
components
 laying out 3-6
confirmation dialogs
 reference link 176
context menu
 creating 182-184
 reference link 184

Core Data

 about 477
 data presentation in sectioned list, with
 @SectionedFetchRequest 507-511
 integrating, with SwiftUI 478-484

Core Data objects

 adding, from SwiftUI view 492-498
 deleting, from SwiftUI view 504-507
 displaying, with @FetchRequest 484-492

Core Data requests

 filtering, predicate used 499-503

CoreLocation 333

CoreLocation wrapper
 implementing, as @ObservedObject

 property wrapper 310-316

curved custom shape

 drawing 196-199

custom fonts

 using, in SwiftUI 569-574

custom rows

 using, in list 57-61

custom shape

 creating 193-196

custom view transitions

 creating 259-263

D**dark mode**

 layout, previewing 114-117

Debug Area 147**devices**

 layout, previewing 128, 129

disclosure groups

 using, to hide content 99-103

 using, to show content 99-103

distributed Notes app
 implementing, with Firebase and SwiftUI 461-475
 dynamic type sizes
 layout, previewing 117-124
 reference link 124

E

easing functions
 URL 242
 editable Collections
 creating 74-76
 editable List view
 creating 67, 68
 expanding lists
 hierarchical content, displaying 95-99

F

Firebase
 about 424, 433
 Cloud Firestore 461
 distributed Notes app,
 implementing with 461-476
 integrating, into SwiftUI project 433-446
 Realtime Database 461
 used, for integrating social login
 with Google 446-461
 Firestore 424
 floating hearts
 creating, in SwiftUI 282-291
 Focus and Submit
 used, for filling out forms 151-154
 FocusState 154
 FocusState, apple documentation
 reference link 155

form
 filling out, with Focus and Submit 151-154
 items, disabling in 147-150
 items, enabling in 147-150
 sections, hiding 142-146
 sections, showing 142-146
 validating, with Combine 352-362
 form, apple documentation
 reference link 147

G

gestures
 using, with TabView 161-163
 getSnapshot function 112
 getTimeLine function 112
 Giphy
 URL 413
 Google Sign In button 453
 gradient view
 rendering, in SwiftUI 214-218
 graphics
 with SF Symbols 33-36

H

hero view transition
 creating, with .matchedGeometryEffect 263-271
 hierarchical content
 displaying, in expanding lists 95-99
 Human Interface guidelines, alerts
 reference link 170
 Human Interface guidelines, popovers
 reference link 188

Human Interface guidelines, sheets
reference link 181

I

images
using 11-17
infinite scrolling
implementing, with `async await` 413-422
`IntentConfiguration` widgets 103
interpolation 289
iOS 13 170
iOS 14 170
iOS 15 169
iOS app
creating, in SwiftUI 514-524
macOS version, creating 524-534
watchOS version, creating 534-542
items
disabling, in forms 147-150
enabling, in forms 147-150

J

JavaScript Object Notation (JSON) 135

K

Kaleidoscope
reference link 554
Keynote Magic Move 270

L

layout
previewing, at dynamic
type sizes 117-124
previewing, in dark mode 114-117

previewing, in `NavigationView` 124-128
previewing, on devices 128, 129

`LazyHGrid`
used, for displaying tabular
content 86-90
`LazyHStack`
using 82-86
`LazyVGrid`
used, for displaying tabular
content 86-90
`LazyVStack`
using 82-86
linear gradient 218
list
creating, of static items 53-57
custom rows, using in 57-61
rows, adding to 62-64
rows, deleting from 64-66
sections, adding to 71-73
List view
rows, moving in 69-71
LoriKeet 407
Lottie animations
in SwiftUI 271-277
reference link 271

M

macOS version
creating, of iOS app 524-534
Markdown editor
implementing, with Preview 574-579
`MarkdownEditor` 574
mock data
using, for previews 135-140
Model-View-Controller (MVC) 333
Model-View-ViewModel (MVVM) 333

multiple animations
applying, to view 256-258

N

NavigationLink buttons
reference link 23
NavigationView
layout, previewing 124-128
NukeUI
about 414
URL 414

O

objc.io
reference link 291
ObjectiveC runtime 511
Open Type Font (OTF) 574

P

PDF
showing, in SwiftUI 564-567
PDFReader 564
pie chart
building 224-234
placeholder function 112
Playground
used, for implementing
SwiftUI views 567-569
popovers
displaying, on iPad 184-187
predicate
about 499
Core Data requests, filtering
with 499-503

Preview
Markdown editor, implementing
with 574-579
progress ring
implementing 203-207
Publishers
in Combine 343
pull-to-refresh 291, 402

R

radial gradient 218
Random Data Generator
about 402
reference link 402
ReactiveX
URL 345
Realm 478
Realtime Database 461
RemoteConfig 433
remote data
fetching, in SwiftUI 397-402
fetching, with Combine 363-376
visualizing, in SwiftUI 363-376
result builders 345
rows
adding, to list 62-64
deleting, from list 64-66
moving, in List view 69-71
RxMarbles
URL 345

S

San Francisco Symbols (SF Symbols)
about 50
graphics, using 33-36

- ScrollViewReader
 used, for scrolling
 programmatically 90-95
- scroll views
 reference link 53
 using 50-53
- Searchable lists
 using 76-79
- sections
 adding, to list 71-73
- sequence of animations
 delay, applying to view modifier
 animation to create 249-252
 delay, applying to withAnimation
 function to create 252-256
- shapes
 transforming 242-245
- sheets
 used, for presenting views 176-181
- Sign in with Apple
 about 424, 425
 implementing, in SwiftUI app 425-432
- snapshot testing
 SwiftUI views 546-555
- social login
 about 446
 integrating with Google,
 Firebase used 446-461
- SpamWords
 about 397
 URL 397
- spring animation
 banner, creating with 246-249
- state objects
 sharing, with multiple views
 using @EnvironmentObject
 property wrapper 321-328
- StaticConfiguration widgets 103
- static items
 list, creating 53-57
- StaticTodoList 301
- stretchable header
 implementing, in SwiftUI 277-282
- styles
 applying, with ViewModifier 27-30
- subscription
 in Combine 343, 344
- Swift Package Manager (SPM) package
 reference link 434
- Swift Snapshot Testing
 reference link 547
- SwiftUI
 adding, to existing app 39-43
 built-in shapes, using 190-193
 Core Data, integrating with 478-484
 custom fonts, using in 569-574
 data, pulling asynchronously 402-407
 data, refreshing asynchronously 402-407
 distributed Notes app,
 implementing with 461-476
 floating hearts, creating 282-291
 gradient view, rendering 214-218
 integrating 392-396
 iOS app, creating 514-524
 Lottie animations 271-277
 PDF, showing 564-567
 remote data, fetching in 397-402
 remote data, visualizing 363-376
 stretchable header,
 implementing 277-282
 swipeable stack of cards,
 implementing 291-298
 Tic-Tac-Toe game,
 implementing 208-213

-
- UIKit, integrating into 36-39
unit testing, with
 ViewInspector 556-563
- SwiftUI app
 Sign in with Apple,
 implementing 425-432
- SwiftUI confirmation dialogs
 presenting 173-176
- SwiftUI images
 reference link 18
- swiftui-lab.com
 reference link 291
- SwiftUI project
 Combine 333-342
 Firebase, integrating 433-446
- SwiftUI Text
 reference link 11
- SwiftUI view
 Core Data objects, adding from 492-498
 Core Data objects, deleting 504-507
 implementing, with Playground 567-569
 snapshot testing 546-555
- SwiftUI widgets
 creating 103-112
- swipeable stack of cards
 implementing, in SwiftUI 291-298
- ## T
- tabular content
 displaying, with LazyHGrid 86-90
 displaying, with LazyVGrid 86-90
- TabView
 about 161
 gestures, using 161-163
 reference link 161
- used, for navigating between
 multiple views 155-160
- text
 dealing with 7-10
- Tic-Tac-Toe game
 implementing, in SwiftUI 208-213
- Timeline
 about 381
 URL 381
- timer app
 memory in Combine, managing
 to build 345-351
- True Type Font (TTF) 572
- ## U
- UIKit
 integrating, into SwiftUI 36-39
 previews, using 131-135
- UIKit PDFView class 567
- ## V
- ViewBuilder
 reference link 33
 used, for separating presentation
 from content 30-32
- ViewInspector
 SwiftUI, unit testing with 556-563
- ViewModifier
 reference link 30
 used, for applying groups of styles 27-30
- view modifier animation
 delay, applying to create sequence
 of animations 249-252

views

presenting, with sheets 176-181

views and controls (iOS 14+)

exploring 43-48

VStack

element, adding 11

W

watchOS version

creating, of iOS app 534-542

withAnimation function

delay, applying to create sequence

of animations 252-256

WWDC SwiftUI SF Symbols

reference link 36

X

XCTest 556

XCUITest 556

