

EXPERT INSIGHT

SwiftUI Cookbook

A guide for building beautiful and interactive SwiftUI apps

Third Edition



<packt>

Juan C. Catalan

SwiftUI Cookbook

Third Edition

A guide for building beautiful and
interactive SwiftUI apps

Juan C. Catalan



BIRMINGHAM—MUMBAI

SwiftUI Cookbook

Third Edition

Copyright © 2023 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 author, 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.

Senior Publishing Product Manager: Larissa Pinto

Acquisition Editor – Peer Reviews: Tejas Mhasvekar

Project Editor: Parvathy Nair

Content Development Editor: Matthew Davies, Shikha Parashar

Copy Editor: Safis Editor

Technical Editor: Srishty Bhardwaj

Proofreader: Safis Editor

Indexer: Tejal Soni

Presentation Designer: Rajesh Shirasath

Developer Relations Marketing Executive: Sohini Ghosh

First published: April 2015

Second published: February 2021

Third edition: December 2023

Production reference: 1191223

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul's Square

Birmingham

B3 1RB, UK.

ISBN 978-1-80512-173-2

www.packt.com

Contributors

About the author

Juan C. Catalan is a software engineer with more than 18 years of professional experience. He started mobile development back in the days of iOS 3. Juan has worked as a professional iOS developer in many industries. Industries such as medical devices, financial services, real estate, document management, fleet tracking, and industrial automation. He has contributed to more than 30 published apps in the App Store, some of them with millions of users. Juan gives back to the iOS development community with technical talks, mentoring developers, reviewing technical books and now as a book author. He lives in Austin, Texas, with his wife, Donna, where they spend time with their kids.

To my wife, Donna, for her love and support during the writing of this book. You are the sunshine of my life.

To my parents, Rosa and Juan, who gave the best years of their lives raising me. They taught me how to be a better person and how to work hard to achieve my dreams.

To my technical reviewers, Chris and Stewart, for their excellent work reviewing the chapters of the book.

To everyone from Packt Publishing who contributed to the book. A special thanks to Matthew Davies, Parvathy Nair, Shikha Parashar, and Larissa Pinto.

About the reviewers

Stewart Lynch is a retired educator and IT professional who got his start in 1969, programming an IBM 360 computer in FORTRAN while obtaining a degree in Mathematics.

Stewart spent over 30 years in the education sector as a teacher, administrator, and IT director in two large school districts in British Columbia, Canada, before taking an early retirement to work for Canada's largest software company.

Throughout his career in the education sector, Stewart pursued his coding passion, developing software solutions for both staff and students. This passion followed him into the public sector where he both taught coding to customers as well as created value-added software solutions for customers worldwide.

Stewart is almost entirely a self-directed software developer, dedicated to keeping up with the latest developments in iOS and MacOS with Swift and SwiftUI. He has multiple apps on the App Store but is now focused on helping others learn and improve their coding skills through his YouTube channel (<https://youtube.com/@StewartLynch>).

I would like to thank the author of this book, Juan C. Catalan, for being so responsive and getting back to me so quickly with answers to my questions throughout this process.

Chris Barker is a Principal Software Engineer at Jaguar Land Rover, where he leads the Mobile Application Engineering Team across the business. With over 22 years of experience in the IT industry, Chris began his career developing .NET applications for the online retailer dabs.com (now BT Shop).

In 2014, Chris transitioned into mobile app development. Before joining Jaguar Land Rover, he worked on mobile apps for clients such as Louis Vuitton, L'Oréal Paris, SimplyBe, JD Williams, and Jacamo.

Chris is the co-host of NS Manchester, a local iOS developer meet-up in Manchester, UK. He has been involved in authoring, co-authoring, and reviewing books for Packt Publishing since 2020.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

<https://packt.link/swiftUI>



Table of Contents

Preface

xxv

Chapter 1: Using the Basic SwiftUI Views and Controls	1
Technical requirements	2
What's new in SwiftUI	2
Laying out components	4
Getting ready • 4	
How to do it... • 4	
How it works... • 8	
There's more... • 8	
Dealing with text	8
Getting ready • 9	
How to do it... • 9	
How it works... • 12	
See also • 12	
Using images	12
Getting ready • 13	
How to do it... • 13	
How it works... • 16	
See also • 19	
Adding buttons and navigating with them	19
Getting ready • 20	
How to do it... • 20	
How it works... • 24	
See also • 25	
Beyond buttons: using advanced pickers	25
Getting ready • 25	

How to do it... • 25	
How it works... • 27	
Applying groups of styles using ViewModifier	28
Getting ready • 29	
How to do it... • 29	
How it works... • 30	
See also • 31	
Separating presentation from content with ViewBuilder	31
Getting ready • 31	
How to do it... • 31	
How it works... • 33	
See also • 34	
Simple graphics using SF Symbols	34
Getting ready • 34	
How to do it... • 34	
How it works... • 37	
See also • 37	
Integrating UIKit into SwiftUI: the best of both worlds	38
Getting ready • 38	
How to do it... • 38	
How it works... • 40	
See also • 40	
Adding SwiftUI to a legacy UIKit app	40
Getting ready • 40	
How to do it... • 41	
How it works... • 44	
Exploring more views and controls	44
Getting ready • 44	
How to do it... • 44	
How it works... • 47	
Chapter 2: Displaying Scrollable Content with Lists and Scroll Views	51
Technical requirements	52
Using scroll views	52
Getting ready • 52	
How to do it... • 52	
How it works... • 54	

See also • 55	
Creating a list of static items	55
Getting ready • 55	
How to do it... • 56	
How it works... • 58	
Using custom rows in a list	58
Getting ready • 58	
How to do it... • 58	
How it works... • 62	
Adding rows to a list	63
Getting ready • 63	
How to do it... • 63	
How it works... • 66	
Deleting rows from a list	67
Getting ready • 67	
How to do it... • 67	
How it works... • 68	
There's more... • 69	
Creating an editable List view	69
Getting ready • 69	
How to do it... • 69	
How it works... • 70	
There's more... • 70	
Moving the rows in a List view	71
Getting ready • 71	
How to do it... • 71	
How it works... • 72	
Adding sections to a list	73
Getting ready • 73	
How to do it... • 73	
How it works... • 75	
Creating editable collections	75
Getting ready • 75	
How to do it... • 75	
How it works... • 76	
Using searchable lists	77
Getting ready • 77	

How to do it... • 77	77
How it works... • 80	80
Using searchable lists with scopes	81
Getting ready • 81	
How to do it... • 81	
How it works... • 83	
See also • 84	
<hr/>	
Chapter 3: Exploring Advanced Components	85
<hr/>	
Technical requirements	85
Using LazyHStack and LazyVStack	86
Getting ready • 86	
How to do it... • 86	
How it works... • 89	
There's more... • 89	
Displaying tabular content with LazyHGrid and LazyVGrid	89
Getting ready • 89	
How to do it... • 90	
How it works... • 92	
Scrolling programmatically	92
Getting ready • 93	
How to do it... • 93	
How it works... • 100	
Displaying hierarchical content in expanding lists	101
Getting ready • 101	
How to do it... • 101	
How it works... • 103	
There's more... • 105	
Using disclosure groups to hide and show content	105
Getting ready • 105	
How to do it... • 105	
How it works... • 108	
Creating SwiftUI widgets	109
Getting ready • 109	
How to do it... • 109	
How it works... • 121	
See also • 121	

Chapter 4: Viewing while Building with SwiftUI Preview in Xcode 15 123

Technical requirements	124
Using the live preview canvas in Xcode15	124
Getting ready • 124	
How to do it... • 124	
How it works... • 137	
See also • 138	
Previewing a view in a NavigationStack	138
Getting ready • 138	
How to do it... • 138	
How it works... • 142	
Previewing a view with different traits	143
Getting ready • 143	
How to do it... • 143	
How it works... • 145	
Previewing a view on different devices	146
Getting ready • 146	
How to do it... • 146	
How it works... • 148	
See also • 149	
Using previews in UIKit	149
Getting ready • 149	
How to do it... • 149	
How it works... • 153	
Using mock data for previews	153
Getting ready • 153	
How to do it... • 153	
How it works... • 157	
There's more... • 157	

Chapter 5: Creating New Components and Grouping Views with Container Views 159

Technical requirements	159
Showing and hiding sections in forms	160
Getting ready • 160	
How to do it... • 160	

How it works... • 164	
There's more... • 164	
See also • 165	
Disabling and enabling items in forms	165
Getting ready • 165	
How to do it... • 166	
How it works... • 168	
Filling out forms easily using Focus and Submit	169
Getting ready • 169	
How to do it... • 169	
How it works... • 172	
There's more... • 172	
See also • 172	
Creating multi-column lists with Table	172
Getting ready • 173	
How to do it... • 173	
How it works... • 187	
There's more... • 189	
See also • 189	
Using Grid, a powerful two-dimensional layout	189
Getting ready • 189	
How to do it... • 190	
How it works... • 193	
See also • 195	
Learn more on Discord	195
<hr/>	
Chapter 6: Presenting Views Modally	197
Technical requirements	197
Presenting alerts	198
Getting ready • 198	
How to do it... • 198	
How it works... • 200	
<i>iOS 15 and later</i> • 201	
<i>iOS 13 and 14</i> • 201	
See also • 201	
Adding actions to alert buttons	201
Getting ready • 201	

How to do it... • 202	
How it works... • 203	
Presenting confirmation dialogs	204
Getting ready • 205	
How to do it... • 205	
How it works... • 206	
See also • 207	
Presenting new views using sheets and full-screen covers	207
Getting ready • 207	
How to do it... • 207	
How it works... • 213	
See also • 214	
Displaying popovers	214
Getting ready • 214	
How to do it... • 214	
How it works... • 216	
See also • 217	
Creating context menus	217
Getting ready • 217	
How to do it... • 217	
How it works... • 219	
See also • 219	

Chapter 7: Navigation Containers**221**

Technical requirements	221
Simple navigation with NavigationStack	222
Getting ready • 222	
How to do it... • 222	
How it works... • 235	
See also • 235	
Typed data-driven navigation with NavigationStack	236
Getting ready • 236	
How to do it... • 236	
How it works... • 251	
See also • 252	
Untyped data-driven navigation with NavigationStack	252
Getting ready • 252	

How to do it... • 252	
How it works... • 258	
See also • 259	
Multi-column navigation with NavigationSplitView	259
Getting ready • 259	
How to do it... • 260	
How it works... • 267	
See also • 269	
Navigating between multiple views with TabView	269
Getting ready • 269	
How to do it... • 269	
How it works... • 273	
There's more... • 274	
See also • 274	
Programmatically switching tabs on a TabView	274
Getting ready • 275	
How to do it... • 275	
How it works... • 276	
See also • 277	
Chapter 8: Drawing with SwiftUI	279
Technical requirements	279
Using SwiftUI's built-in shapes	280
Getting ready • 280	
How to do it... • 280	
How it works... • 282	
Drawing a custom shape	283
Getting ready • 283	
How to do it... • 284	
How it works... • 285	
Drawing a curved custom shape	287
Getting ready • 287	
How to do it... • 287	
How it works... • 290	
Drawing using the Canvas API	290
Getting ready • 290	
How to do it... • 290	

How it works... • 292	
There's more... • 293	
Implementing a progress ring	293
Getting ready • 293	
How to do it... • 293	
How it works... • 296	
Implementing a Tic-Tac-Toe game in SwiftUI	297
Getting ready • 297	
How to do it... • 297	
How it works... • 302	
There's more... • 302	
Rendering a gradient view in SwiftUI	302
Getting ready • 302	
How to do it... • 302	
How it works... • 307	
Chapter 9: Animating with SwiftUI	309
Technical requirements	310
Creating basic animations	310
Getting ready • 310	
How to do it... • 310	
How it works... • 313	
There's more... • 314	
See also • 314	
Transforming shapes	314
Getting ready • 315	
How to do it... • 315	
How it works... • 317	
Creating a banner with a spring animation	318
Getting ready • 318	
How to do it... • 318	
How it works... • 320	
Applying a delay to an animation view modifier to create a sequence of animations	321
Getting ready • 321	
How to do it... • 321	
How it works... • 323	

Applying a delay to a <code>withAnimation</code> function to create a sequence of animations	324
Getting ready • 324	
How to do it... • 324	
How it works... • 327	
There's more... • 327	
Applying multiple animations to a view	327
Getting ready • 328	
How to do it... • 328	
How it works... • 329	
Chained animations with <code>PhaseAnimator</code>	330
Getting ready • 330	
How to do it... • 330	
How it works... • 334	
Custom animations with <code>KeyframeAnimator</code>	334
Getting ready • 334	
How to do it... • 335	
How it works... • 338	
Creating custom view transitions	339
Getting ready • 339	
How to do it... • 340	
How it works... • 342	
Creating a hero view transition with <code>.matchedGeometryEffect</code>	342
Getting ready • 343	
How to do it... • 343	
How it works... • 348	
Implementing a stretchable header in SwiftUI	349
Getting ready • 350	
How to do it... • 350	
How it works... • 354	
Implementing a swipeable stack of cards in SwiftUI	354
Getting ready • 354	
How to do it... • 354	
How it works... • 360	
Chapter 10: Driving SwiftUI with Data	361
 Technical requirements	362

Using @State to drive a view's behavior	362
Getting ready • 363	
How to do it... • 363	
How it works... • 365	
See also • 366	
Using @Binding to pass a state variable to child views	366
Getting ready • 366	
How to do it... • 366	
How it works... • 370	
Implementing a CoreLocation wrapper as@ObservedObject	370
Getting ready • 370	
How to do it... • 371	
How it works... • 375	
Using @StateObject to preserve the model's life cycle	376
Getting ready • 376	
How to do it... • 376	
How it works... • 379	
Sharing state objects with multiple Views using@EnvironmentObject	380
Getting ready • 381	
How to do it... • 381	
How it works... • 387	
See also • 387	
Using Observation to manage model data	387
Getting ready • 388	
How to do it... • 389	
How it works... • 399	
See also • 400	
Chapter 11: Driving SwiftUI with Combine	403
Technical requirements	404
Introducing Combine in a SwiftUI project	404
Getting ready • 405	
How to do it... • 405	
How it works... • 411	
<i>Publishers in Combine</i> • 412	
<i>Subscriptions in Combine</i> • 412	
See also • 414	

Managing the memory in Combine to build a timer app	414
Getting ready • 414	
How to do it... • 414	
How it works... • 418	
See also • 419	
Validating a form using Combine	419
Getting ready • 420	
How to do it... • 420	
How it works... • 428	
There's more... • 428	
Fetching remote data using Combine and visualizing it in SwiftUI	428
Getting ready • 429	
How to do it... • 430	
How it works... • 438	
There's more... • 440	
Debugging an app based on Combine	440
Getting ready • 441	
How to do it... • 441	
How it works... • 445	
There's more... • 445	
Unit testing an app based on Combine	446
Getting ready • 446	
How to do it... • 447	
How it works... • 453	
Chapter 12: SwiftUI Concurrency with <code>async await</code>	455
Technical requirements	456
Integrating SwiftUI and an <code>async</code> function	456
Getting ready • 456	
How to do it... • 456	
How it works... • 458	
Fetching remote data in SwiftUI	459
Getting ready • 460	
How to do it... • 460	
How it works... • 463	
Pulling and refreshing data asynchronously in SwiftUI	464
Getting ready • 464	

How to do it... • 464	
How it works... • 467	
Converting a completion block function to async await	468
Getting ready • 468	
How to do it... • 469	
How it works... • 471	
See also • 473	
Implementing infinite scrolling with async await	473
Getting ready • 474	
How to do it... • 475	
How it works... • 478	
See also • 479	
<hr/>	
Chapter 13: Handling Authentication and Firebase with SwiftUI	481
<hr/>	
Technical requirements	482
Implementing Sign in with Apple in a SwiftUI app	483
Getting ready • 483	
How to do it... • 484	
How it works... • 488	
Integrating Firebase into a SwiftUI project	489
Getting ready • 490	
How to do it... • 492	
How it works... • 503	
There's more... • 504	
See also • 504	
Using Firebase to sign in users with Google Sign-In	504
Getting ready • 505	
How to do it... • 507	
How it works... • 515	
See also • 516	
Implementing a Notes app with Firebase and SwiftUI	517
Getting ready • 517	
How to do it... • 518	
How it works... • 532	
There's more... • 533	
See also • 534	

Chapter 14: Persistence in SwiftUI with Core Data and SwiftData	535
Technical requirements	536
Integrating Core Data with SwiftUI	536
Getting ready • 537	
How to do it... • 537	
How it works... • 541	
There's more... • 541	
Showing Core Data objects with @FetchRequest	541
Getting ready • 541	
How to do it... • 541	
How it works... • 547	
Adding Core Data objects from a SwiftUI view	547
Getting ready • 547	
How to do it... • 547	
How it works... • 552	
Filtering Core Data requests using a Predicate	553
Getting ready • 553	
How to do it... • 553	
How it works... • 555	
Deleting Core Data objects from a SwiftUI view	556
Getting ready • 556	
How to do it... • 557	
How it works... • 559	
Presenting data in a sectioned list with @SectionedFetchRequest	560
Getting ready • 560	
How to do it... • 560	
How it works... • 562	
Getting started with SwiftData	563
Getting ready • 564	
How to do it... • 564	
How it works... • 571	
There's more... • 573	
See also • 573	

Chapter 15: Data Visualization with Swift Charts	575
Technical requirements	576
Understanding the basics of Swift Charts	576
Getting ready • 576	
How to do it... • 576	
How it works... • 581	
See also • 584	
Customizing charts: axes, annotations, and rules	584
Getting ready • 584	
How to do it... • 584	
How it works... • 596	
See also • 599	
Different types of charts: marks and mark configuration	599
Getting ready • 600	
How to do it... • 600	
How it works... • 620	
See also • 622	
Histograms with data bins	623
Getting ready • 623	
How to do it... • 623	
How it works... • 627	
There's more • 630	
See also • 630	
Pie charts and donut charts	630
Getting ready • 630	
How to do it... • 630	
How it works... • 635	
See also • 637	
Interactive charts: selection	637
Getting ready • 637	
How to do it... • 637	
How it works... • 664	
There's more • 666	
See also • 666	

Interactive charts: scrollable content	667
Getting ready • 667	
How to do it... • 667	
How it works... • 673	
There's more • 675	
See also • 675	
Chapter 16: Creating Multiplatform Apps with SwiftUI	677
Technical requirements	678
Creating an iOS app in SwiftUI	678
Getting ready • 678	
How to do it... • 679	
How it works... • 687	
Creating the macOS version of the app with a new target	688
Getting ready • 688	
How to do it... • 689	
How it works... • 696	
Creating a multiplatform version of the app sharing the same target	697
Getting ready • 697	
How to do it... • 697	
How it works... • 707	
There's more... • 708	
Creating the watchOS version of the iOS app	708
Getting ready • 709	
How to do it... • 710	
How it works... • 717	
Chapter 17: SwiftUI Tips and Tricks	719
Technical requirements	719
Using XCTest to test SwiftUI apps	720
Getting ready • 720	
How to do it... • 721	
How it works... • 731	
There's more... • 732	
See also • 734	

Using custom fonts in SwiftUI	734
Getting ready • 734	
How to do it... • 737	
How it works... • 739	
Showing a PDF in SwiftUI	740
Getting ready • 740	
How to do it... • 740	
How it works... • 742	
There's more... • 743	
Implementing a Markdown editor with preview functionality	743
Getting ready • 743	
How to do it... • 743	
How it works... • 746	
Other Books You May Enjoy	751
Index	757

Preface

SwiftUI is the modern way to build user interfaces for all Apple platforms. SwiftUI code is easy to read and write because it uses Swift's declarative programming syntax. The new edition of this comprehensive cookbook includes fully updated content, code samples and a companion GitHub repository for SwiftUI 5, iOS 17, Xcode 15, and Swift 5.9.

This book covers the foundations of SwiftUI, as well as the new features of SwiftUI 5 introduced in iOS 17. Particular attention is given to illustrating the interaction between SwiftUI and the rest of an app's code, such as concurrency with Combine and `async/await`, persistence with Core Data and Swift Data, and data visualization with Swift Charts.

With over 100 recipes for real-world user interfaces packed with best-coding practices, you'll learn how to use SwiftUI to build visually compelling apps, from simple lists to complex interactive charts.

Who this book is for

This book is for mobile developers who want to learn SwiftUI as well as for experienced iOS developers transitioning from UIKit to SwiftUI. The book assumes knowledge of the Swift programming language. 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 the basic building blocks for creating SwiftUI apps and how to integrate UIKit components in SwiftUI views.

Chapter 2, Displaying Scrollable Content with Lists and Scroll Views, explains how to use using `List` views and `ScrollView` containers to display content when it does not fit on a single screen. You'll also be able to learn about editable and searchable lists.

Chapter 3, Exploring Advanced Components, explains how to use `LazyStack` and `Lazy Grid` 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 `DisclosureGroup`. Finally, you'll learn how to display glanceable content outside your app using SwiftUI Widgets.

Chapter 4, Viewing while Building with SwiftUI Preview in Xcode 15, explains how to use the preview macro introduced with Swift 5.9 with the powerful live preview in the Xcode 15 canvas.

Chapter 5, Creating New Components and Grouping Views with Container Views, explains how to use Form views for user interaction, Table for multi-column lists, and Grid for powerful two-dimensional layouts.

Chapter 6, Presenting Views Modally, explains how to display custom views for narrow and focused user interaction, using alerts, confirmation dialogs, sheets, popovers, and context menus.

Chapter 7, Navigation Containers, explains how to use NavigationStack for hierarchical, data-driven navigation, NavigationSplitView for multi-column user interfaces, and TabView for navigation among non-hierarchical content.

Chapter 8, Drawing with SwiftUI, explains how to implement drawings in SwiftUI by using built-in shapes and drawing custom paths and polygons.

Chapter 9, Animating with SwiftUI, explains how to implement basic animations, spring animations, implicit and delayed animations, as well as how to combine transitions, create custom transitions, and create asymmetric transitions. Additionally, you'll also be able to learn how to perform chained animations with PhaseAnimator and full-custom animations with KeyFrameAnimator.

Chapter 10, Driving SwiftUI with Data, explains how to use the SwiftUI binding mechanism to populate and change views when the bounded data changes. At the end of the chapter, you'll learn about the Observation framework and the Observable macro, introduced with iOS 17.

Chapter 11, 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 12, SwiftUI Concurrency with async await, explains how to implement Swift concurrency with async await, fetching from a network resource, and creating an infinite scrolling page.

Chapter 13, Handling Authentication and Firebase with SwiftUI, explains how to implement authentication in your app and store app data in a cloud database.

Chapter 14, Persistence in SwiftUI with Core Data and Swift Data, explains how to implement persistence using SwiftUI and Core Data. You'll learn how to save, delete, and modify objects in a Core Data local database. You'll also learn how to use Swift Data, the new persistence framework introduced with iOS 17.

Chapter 15, Data Visualization with Swift Charts, explains how to use Swift Charts for data visualization. You'll learn how to plot different types of charts using a few building blocks and powerful view modifiers. You'll discover interactive charts and range charts using real-world examples. You'll also learn how to transform your data to simplify the visualization process.

Chapter 16, Creating Multiplatform Apps with SwiftUI, explains how to create a multiplatform SwiftUI app that works on iOS 17, macOS 14 Sonoma, and watchOS 10.

Chapter 17, SwiftUI Tips and Tricks, covers several tips and tricks of SwiftUI that will help you to solve several common problems, such as testing SwiftUI views, using custom fonts, showing PDF documents, and using Markdown text.

To get the most out of this book

Inform the reader the things that they need to know before they start: spell out what knowledge you are assuming. Tell the reader anything they need to know before they start.

This book has been completely revised for Swift 5, iOS 17, Xcode 15, and Swift 5.9.

To build all the apps in this book, you will need:

- A Mac computer running macOS Ventura 13.5 or later. For *Chapter 16*, you'll need macOS 14.0 Sonoma or later.
- Xcode 15.0 or later.

The book extensively uses the live preview of the Xcode canvas to introduce the learning material and occasionally uses the iOS simulator. If you want to test the apps on your iPhone or iPad, you will need to obtain a free Apple Developer account from Apple at <https://developer.apple.com>.

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 to avoid any potential errors related to the copying and pasting of code.

Download the example code files

The code bundle for the book is hosted on GitHub at <https://github.com/PacktPublishing/SwiftUI-Cookbook-3rd-Edition>. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: <https://packt.link/gbp/9781805121732>.

Conventions used

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

CodeInText: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. For example: “Mount the downloaded WebStorm-10*.dmg disk image file as another disk in your system.”

A block of code is set as follows:

```
Chart {
    ForEach(data) { entry in
        BarMark(
            x: .value("Question", entry.question),
            y: .value("Count", entry.count)
        )
    }
}
```

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 TodoItem: Identifiable {  
    let id = UUID()  
    var title: String  
}
```

Bold: Indicates a new term, an important word, or words that you see on the screen. For instance, words in menus or dialog boxes appear in the text like this. For example: “Tap on the + button a few times and then on the **Refresh** button”.



Warnings or important notes appear like this.



Tips and tricks 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 step-by-step instructions required to build the app of the recipe. Screenshots and explanatory figures are provided along with the code snippets.

How it works...

This section usually consists of a detailed explanation of the steps followed in the previous section. It explains in detail the code snippets and technologies used in the recipe.

There's more...

This optional section covers additional topics related to the recipe to make you more knowledgeable about the technologies used to build the app.

See also

This optional section provides helpful links to other useful information about the technologies used in the recipe.

Get in touch

Feedback from our readers is always welcome.

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

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

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

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

Share your thoughts

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

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

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere? Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily.

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below



<https://packt.link/free-ebook/9781805121732>

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

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 widely by the Apple developer community. Apple also releases updates every year, adding new and exciting capabilities to SwiftUI.

SwiftUI is a UI framework that ditches UIKit concepts such as Auto Layout for an easier-to-use declarative programming model. SwiftUI is Apple's preferred way to build user interfaces. It is a platform-agnostic framework that allows the fast and easy creation of applications that work across Apple platforms (iOS, iPadOS, macOS, WatchOS, and tvOS).

There is no question today about the need to learn about SwiftUI. Apple released the fifth iteration of the framework in 2023 and it is adding more features every year. Here are some other compelling reasons to be proficient in SwiftUI:

- **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 has already been adopted by the industry as it was released four years ago. Just looking at job postings for iOS developers, you'll find out that most of them require experience with SwiftUI. Learning about SwiftUI is a must for current iOS development jobs. In a few more years, SwiftUI will dominate the app development for all the Apple platforms, the same way that Swift took over from Objective-C a few years ago.
- **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 without having to recompile the whole app. You can quickly prototype apps and make any changes required by your users. 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 user 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 symbol)**
- Integrating UIKit into SwiftUI—the best of both worlds
- Adding SwiftUI to a legacy UIKit app
- Exploring more views and controls

Technical requirements

The code in this chapter is based on Xcode 15.0 and iOS 17.0. You can download and install the latest version of Xcode from the App Store. You'll also need to be running macOS Ventura (13.4) or newer.

Simply search for Xcode in the App Store and select and download the latest version. Launch Xcode and follow any additional installation instructions that your system may prompt you with. Once Xcode has fully launched, you're ready to go.

All the code examples for this chapter can be found on GitHub at <https://github.com/PacktPublishing/SwiftUI-Cookbook-3rd-Edition/tree/main/Chapter01-Using-the-basic-SwiftUI-Views-and-Controls/>.

What's new in SwiftUI

SwiftUI has been evolving since the day it was announced in 2019. Every year, Apple adds new APIs and SwiftUI becomes more and more powerful. At the time of this writing, it is possible to create an iOS app exclusively using SwiftUI, without having to integrate UIKit. Since the previous edition of this book, Apple added new functionality to SwiftUI.

In WWDC 2022, Apple added new features to SwiftUI, improved some existing features, and even deprecated some of the APIs just introduced a few years ago. These are the most relevant features:

- **Swift Charts**, which allows you to create data visualizations across all Apple platforms
- New data-driven navigation with `NavigationStack` and `NavigationSplitView`

- Enhancements to **Form**, to support multi-platform apps with a single code base:
 - **LabeledContent** view to display pairs of data, like key-value pairs or title and description, inside forms
 - Deeper customization of multi-line **TextField** instances
 - New **MultiDatePicker** control to select more than one date
 - New mixed-state controls for **Toggle** and **Picker** views and a new format for **Stepper** views.
- **Table**, introduced in macOS 12, now available on iPadOS 16 and iOS 16, and new toolbar customization for iPadOS
- **PhotosPicker** view, a multi-platform and privacy-preserving API for picking photos and videos
- **ShareLink** view, which enables the presentation of the share sheet even in WatchOS
- **Transferable** protocol, to share data between apps
- **ShapeStyle** extensions like **gradient** and **shadow**, which can also be applied to **SFSymbols**
- **Grid**, to arrange content in a two-dimensional way
- **Layout** protocol to create full-custom layouts

In WWDC 2023, Apple added more functionality to SwiftUI. These are the most relevant features:

- Interactivity and animation added to widgets
- Improvement to Xcode previews with a new `Preview(_:traits:body:)` macro that supports UIKit and AppKit out of the box
- Native SwiftUI support for MapKit
- Interactivity and pie charts added to **Swift Charts**
- Navigation, date pickers, and toolbars available in WatchOS 10
- **SwiftData**, a successor of **CoreData**, used to persist data between app launches
- The **Observable** macro, and new **@State**, **@Environment**, and **@Bindble** property wrappers, offering a new way of sharing data throughout the app
- New powerful animations with the new spring animation, the **PhaseAnimator** struct, and the **Keyframe Animator** and **CustomAnimation** protocol
- Inspector, a new modal presentation with the `inspector(isPresented:content:)` view modifier
- Symbol Effects, animated symbols added to **SF Symbols 5**
- New powerful scrolling APIs: transition effects, scroll position, paged scrolling, and inset control
- Enhancements to list and tables: item selection, expanding sections programmatically, column visibility, column header visibility, hierarchical rows, and alternating row background
- New dialog customizations, new gestures, and new input events

Laying out components

In this very first recipe of the book, we will start laying out components. In SwiftUI, our user interface is composed of different elements. It is very important to understand how to group components together in different layouts. SwiftUI uses three basic layout components, `VStack`, `HStack`, and `ZStack`. Use the `VStack` view to arrange components on a vertical axis, `HStack` to arrange components on 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 spacing 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.
3. The following screen asks us to choose an Xcode template. Select **iOS**, and then **App**. Click **Next**.
4. A screen to select the options for the project will appear. Enter the product name, **TheStacks**.
5. Make sure that **Interface** is set to **SwiftUI**, **Language** is set to **Swift**, and **Storage** is set to **None**. Click **Next**.
6. Select the folder location to store the project and choose if you want to create a Git repository or not. Then click **Create**.

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 content of the `body` variable with a `VStack` and some `Text` views:

```
var body: some View {
    VStack {
        Text("VStack Item 1")
        Text("VStack Item 2")
        Text("VStack Item 3")
    }
    .background(.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:

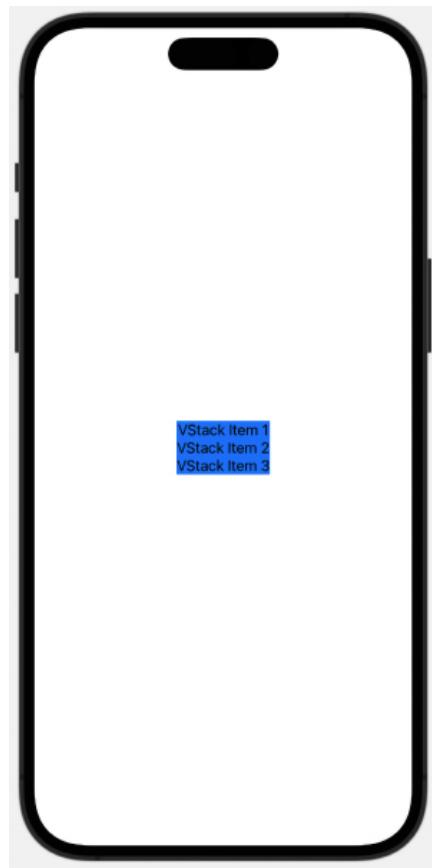


Figure 1.1: VStack with three items

4. Add a **Spacer** and a **Divider** between **VStack Item 2** and **VStack Item 3**:

```
Spacer()  
Divider()  
.background(.black)
```

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

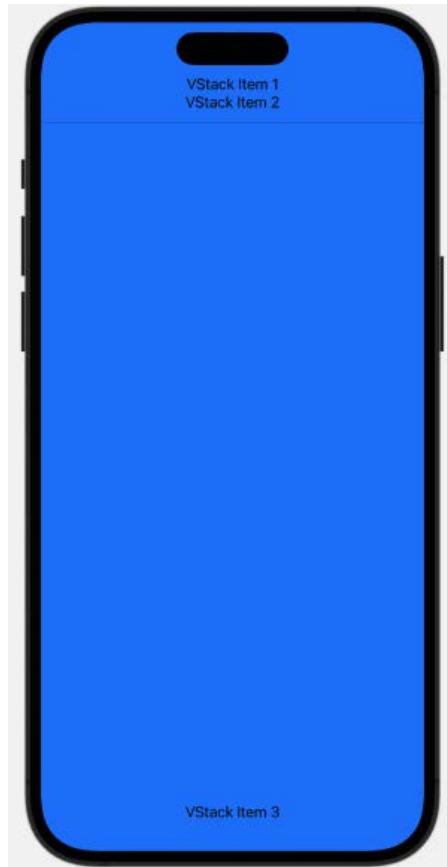


Figure 1.2: VStack + Spacer + Divider

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

```
HStack{  
    Text("HStack Item 1")  
    Divider()  
        .background(.black)  
    Text("HStack Item 2")  
    Divider()  
        .background(.black)  
    Spacer()  
    Text("HStack Item 3")  
}  
.background(Color.red)  
ZStack{
```

```
Text("ZStack Item 1")
    .padding()
    .background(.green)
    .opacity(0.8)
Text("ZStack Item 2")
    .padding()
    .background(.green)
    .offset(x: 80, y: -400)
}
```

7. The preview should look like the following screenshot (it may vary depending on the device selected for previews):

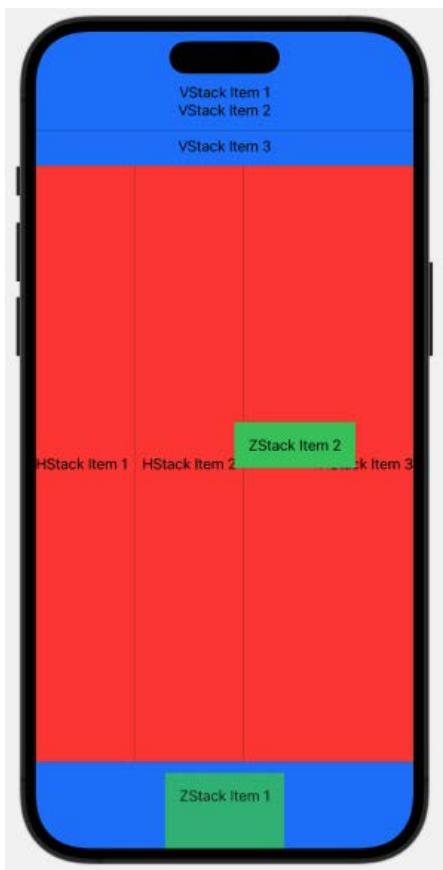


Figure 1.3: VStack, HStack, and ZStack

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...

In Xcode 15, a new iOS app project with SwiftUI selected as the interface option starts with a VStack that includes an Image view and a Text view located at the center of the screen. We replaced the content provided by the template with our own VStack, with three embedded Text views. SwiftUI container views like VStack 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(.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(.black), .padding(), and .offset(...).

The HStack container 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 masked for privacy reasons.

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 masked, 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. Replace the initial `ContentView` body variable with our own `VStack`. The `ContentView` should look like the following code:

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

4. Now, we will start adding more views to the `VStack`. Each view should be added immediately after the previous one. 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` views with modifiers to the list:

```
Text("Changing text color and make it bold")  
    .foregroundStyle(.blue)  
    .bold()  
Text("Use kerning to change space between characters in the 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(.yellow)  
    .multilineTextAlignment(.trailing)  
    .lineSpacing(10)
```

Now is the moment to test the app. We can choose to run the app in a simulator or click the Play button in the canvas preview, which allows for interactivity. Play with the app and enter some text in the `SecureField` and `TextField`. Text entered in the `SecureField` will be masked, while text in the `TextField` will be shown.

The resulting preview should look like this:

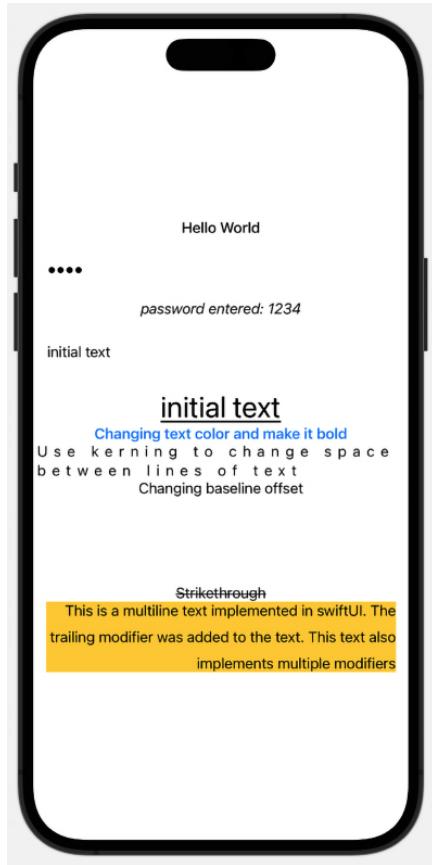


Figure 1.4: `FormattedText` preview

How it works...

Text views have several modifiers for font, spacing, and other formatting requirements. When in doubt, position the cursor on the line of code that includes the `Text` view, and press the `Esc` key to reveal a list of all available modifiers. This is shown in the following example:

```

15     Text("Hello World")|  

16         .fontWeight(M font(_ font:)  

17             M bold()  

18             M bold(_ isActive:)  

19             M fontWeight(_ weight:)  

20             M italic()  

21             M padding(_ insets:)  

22             M padding(_ edges: _ length:)  

23             M tag(_ tag:)  

24             M accessibilityHeading(_ level:)  

25             M underline(M accessibilityLabel(_ label:)  

26             Text("Changing" M font(_ font: Font?) -> Text  

27                 .foreground(Sets the default font for text in the view.  

28                 .bold()  

29             Text("Use kerning to change space between characters in the text")  

30                 .kerning(7)

```

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 parameters. 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 wrapped value of bound state variables, which is the underlying value referenced by the state variable, is accessed without having to use the `$` symbol. This is a convenience shortcut provided by Swift, as shown in the following code snippet:

```
Text(someText)
```

See also

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

Using images

Apps need to be appealing to users and need to engage them to interact with the app. To that purpose, a well-crafted and beautiful user interface with simple and intuitive interactions is very desirable.

Images play an important part in an app's user interface as they add color and simplicity and they convey messages in a graphical way. It is fundamental to master how to use images in your apps.

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 **UsingImages**.

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 content of the `body` variable with an empty `VStack`.

```
var body: some View {  
    VStack {  
    }  
}
```

2. Download the project images from the GitHub link at <https://github.com/PacktPublishing/SwiftUI-Cookbook-3rd-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:

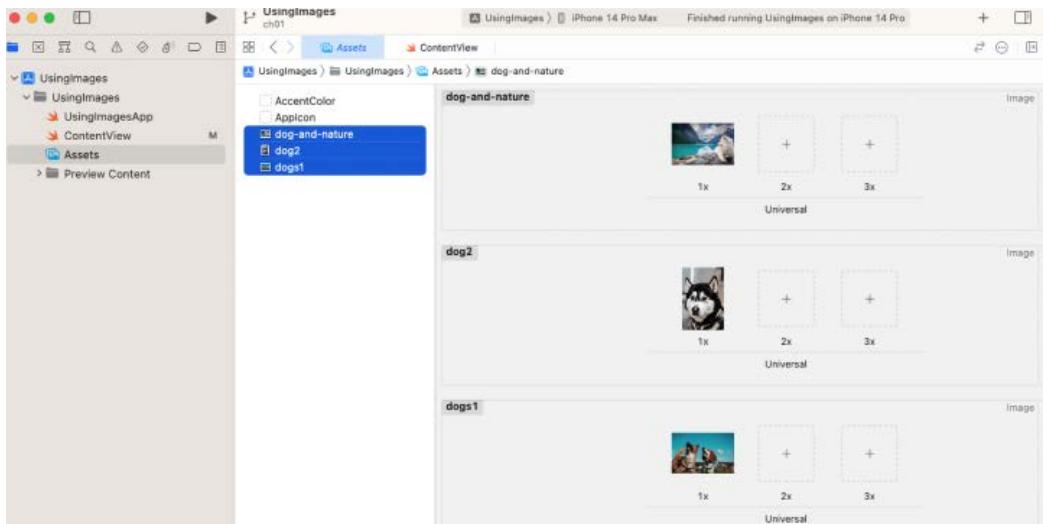


Figure 1.6: Assets.xcassets folder in Xcode

4. Add an `Image` view to `VStack`:

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

5. Observe the result in the canvas preview.

6. In iOS 17, the `ImageResource` and `ColorResource` structs were introduced, backward compatible to iOS 11 for UIKit and iOS 13 for SwiftUI. Xcode 15 automatically generates instances of `ImageResource` and `ColorResource` for images and colors in asset catalogs. For example, the three images in our asset catalog, shown in *Figure 1.6*, generate the following code:

```
// MARK: - Image Symbols -  
  
@available(iOS 11.0, macOS 10.7, tvOS 11.0, *)  
extension ImageResource {  
  
    /// The "dog-and-nature" asset catalog image resource.  
    static let dogAndNature = ImageResource(name: "dog-and-nature",  
bundle: resourceBundle)  
  
    /// The "dog2" asset catalog image resource.  
    static let dog2 = ImageResource(name: "dog2", bundle: resourceBundle)  
  
    /// The "dogs1" asset catalog image resource.  
    static let dogs1 = ImageResource(name: "dogs1", bundle:  
resourceBundle)  
  
}  
  
#if canImport(SwiftUI)  
@available(iOS 13.0, macOS 10.15, tvOS 13.0, watchOS 6.0, *)  
extension SwiftUI.Image {  
  
    /// Initialize an 'Image' with an image resource.  
    init(_ resource: ImageResource) {  
        self.init(resource.name, bundle: resource.bundle)  
    }  
  
}
```

7. This is auto-generated code, compiled along with our own code. Thanks to the new initializer for `Image`, which takes an `ImageResource` instead of `Image("dogs1")`, we can write `Image(.dogs1)`. The advantage of this approach is the compile-time checking of the correct image name, which eliminates runtime errors from typos.
8. 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()
```

9. 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)
```

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

```
Image(.dogAndNature)  
    .resizable()  
    .aspectRatio(contentMode: .fit)  
    .frame(width:300, height:200)  
    .clipShape(Circle())  
    .overlay(Circle().stroke(.blue, lineWidth: 6))  
    .shadow(radius: 10)
```

11. We can also use a `UIImage` instance to initialize an `Image` view. This is useful if the `UIImage` was generated with legacy code or programmatically. In our example, we use the `UIImage` convenience initializer, which takes an `ImageResource` instance. For example, to create a `UIImage` from the `dogs2` image in our asset catalog, we would use: `UIImage(resource: .dogs2)`.

12. Use the `UIImage` and display it 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: UIImage(resource: .dog2))  
        .resizable()  
        .aspectRatio(contentMode: .fit)  
        .frame(width: 200, height: 200)  
    }  
}  
}
```

13. The completed application should then look like this:



Figure 1.7: Using Images 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 14 Pro Max screen:



Figure 1.8: The 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()` or `.aspectRatio()` modifiers may lead to different results.

Note that if you set the project deployment target to iOS 14, the `ImageResource` struct works without issues since Apple made the struct available for older versions of iOS. This allows us to use the new APIs in older versions of iOS in case your app needs to support them.

See also

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

Adding buttons and navigating with them

In this recipe, we will learn how to use the various buttons available in SwiftUI. We will use a `Button` view to trigger the change of a count when clicked and implement a `NavigationStack` 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 **Buttons**.

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 `NavigationLink` to show the view that implements the 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, which is available for macOS, iOS and iPadOS.

7. Open the `ContentView.swift` file and create a `NavigationStack` to navigate between the SwiftUI views we added to the project. The `ContentView` struct should look like this:

```
struct ContentView: View {  
    var body: some View {  
        NavigationStack {  
            VStack(spacing: 44) {  
                NavigationLink("Buttons") {  
                    ButtonView()  
                }  
                NavigationLink("EditButtons") {  
                    EditButtonView()  
                }  
                NavigationLink("MenuButtons") {  
                    MenuButtonView()  
                }  
                NavigationLink("PasteButtons") {  
                    PasteButtonView()  
                }  
                NavigationLink("Details about text") {  
                    TextDetailsView()  
                }  
            }  
        }  
    }  
}
```

```
        Text("Very long text that should not be displayed in  
a single line because it is not good design")  
            .padding()  
            .navigationTitle(Text("Detail"))  
        }  
    }  
    .navigationTitle(Text("Main View"))  
}  
}  
}
```

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

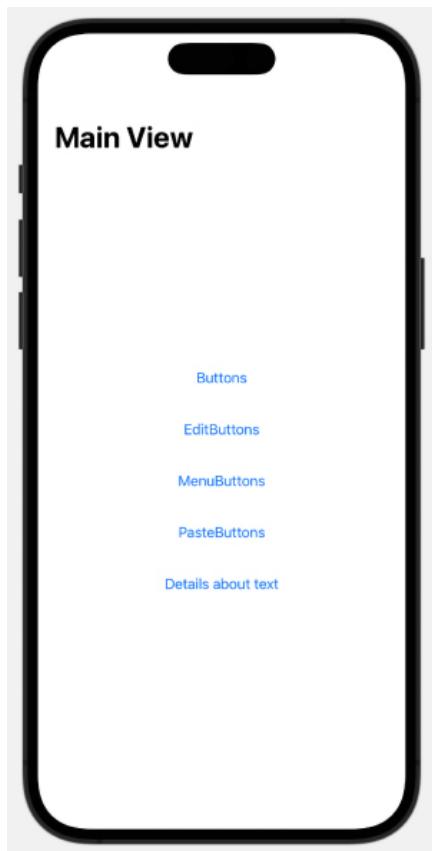


Figure 1.13: ButtonsApp ContentView

9. Open the `ButtonView.swift` file in the project navigator and replace the existing struct with the following code:

```
struct ButtonView: View {  
    @State var count = 0  
    var body: some View {
```

```
    VStack {
        Text("Welcome to your second view")
        Text("Current count value: \((count)")
            .padding()
        Button {
            count += 1
        } label: {
            Text("Tap to Increment count")
                .fontWeight(.bold)
                .foregroundStyle(.white)
                .padding()
                .background(.blue)
                .clipShape(Capsule())
        }
    }.navigationBarTitle("Button View")
}

#Preview {
    NavigationStack {
        ButtonView()
    }
}
```

10. Open the `EditButtonView.swift` file in the project navigator and replace the existing struct with the following code that implements an `EditButton`:

```
struct EditButtonView: View {
    @State private var animals = ["Cats", "Dogs", "Goats"]
    var body: some View {
        List{
            ForEach(animals, id: \.self){ animal in
                Text(animal)
            }
        .onDelete(perform: removeAnimal)
    }
    .toolbar {
        EditButton()
    }
    .navigationTitle("EditButtonView")
}
func removeAnimal(at offsets: IndexSet){
```

```
        animals.remove(atOffsets: offsets)
    }
}

#Preview {
    NavigationStack {
        EditButtonView()
    }
}
```

11. Open the `MenuButtonView.swift` file and replace the existing struct with the following code for `MenuButtonView`:

```
struct MenuButtonView: View {
    var body: some View {
        Menu("Choose a country") {
            Button("Canada") { print("Selected Canada") }
            Button("Mexico") { print("Selected Mexico") }
            Button("USA") { print("Selected USA") }
        }
        .navigationTitle("MenuButtons")
    }
}

#Preview {
    NavigationStack {
        MenuButtonView()
    }
}
```

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

```
struct PasteButtonView: View {
    @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()
        }
        .navigationTitle("PasteButton")
    }
}
```

```
    }

    #Preview {
        NavigationStack {
            PasteButtonView()
        }
    }
}
```

Go back to `ContentView`, 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 `NavigationStack` or `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`. Since our label is a simple `Text` view, we use the convenience initializer `init(_:destination:)` of `NavigationLink` to keep our code more concise.

`NavigationLink` buttons can be used to move from one SwiftUI view to another—for example, moving from `ContentView` to `EditButtonView`. They can also be used to display text details without creating a SwiftUI view in a separate 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 `.navigationTitle("Main View")` modifier adds a title to the `ContentView` screen.

The `.navigationTitle()` modifier is also added to `EditButtonView` and other views. Since these views do not contain `NavigationStack` 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`. To solve this, we use a `NavigationStack` in the `PreviewProvider` structs. To make the previews more useful, note how we have enclosed the view in a `NavigationStack` so we can see the title in the canvas preview window.

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, Displaying Scrollable Content with Lists and Scroll Views*, but `EditButtonView` provides a peek into how to create an editable list.

The `MenuButtonView` uses the `Menu` struct, introduced in iOS 14, to display a floating menu of actions. Check out the *Exploring more views and controls* recipe at the end of this chapter for more information on `Menu`.

`PasteButtons` are only available on macOS. 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>.
- More information regarding Menu can be found here: <https://developer.apple.com/documentation/swiftui/menu/>.

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 **UsingPickers**.

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

2. Replace the body of the **ContentView** struct with a **Form** view. Then, add a **Section** view and a **Picker** view to the form:

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

```
        .pickerStyle(.segmented)
        Text("Current choice: \(transitModes[choice])")
    }
}
```

3. Under the existing `Section` view, add another `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()...)
}
```

8. The result should be a beautiful form, like what is shown here:

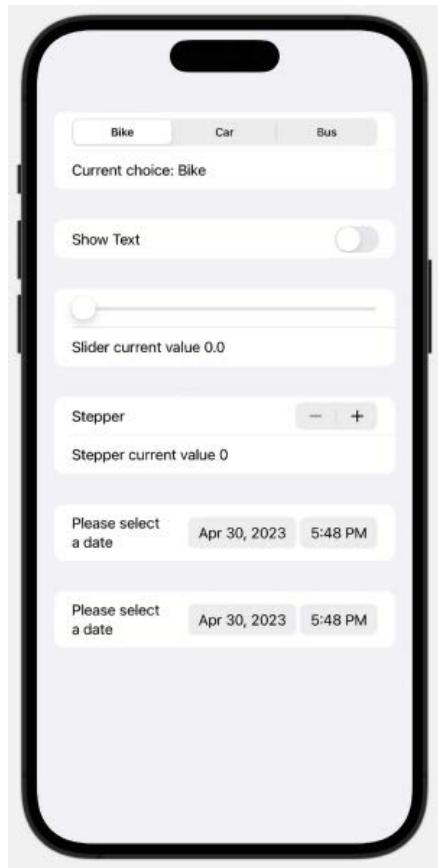


Figure 1.14: SwiftUI Form with Pickers

How it works...

Form views group controls that are used for data entry, and Section views create hierarchical view content. Section views can be embedded inside a Form view to display information grouped together. The default presentation style for a Form with embedded Section views is to include a gray *padding* area between each section for visual grouping, as shown in *Figure 1.14*.

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

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

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("\(transitModes[index])")`. It is also worth noting that we need to apply a `.segmented` style to the picker using the `.pickerStyle()` modifier, to use the visual segmented style everyone is used to in iOS.

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 `@State` property of the `Toggle` reflects the current state 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 need to 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 container view. Styles can be overridden using the `.pickerStyle()` modifier.

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. Replace the current body of the `ContentView` view with:

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

2. At the end of 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)  
            .foregroundStyle(.black)  
            .padding()  
            .background(bgColor)  
            .cornerRadius(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. The extension should be created outside the struct or Xcode will issue an error:

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

5. Replace the modifier on the `Text` view with the `backgroundStyle()` modifier that you just created, which will add your custom styles:

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

6. The result should look like this:

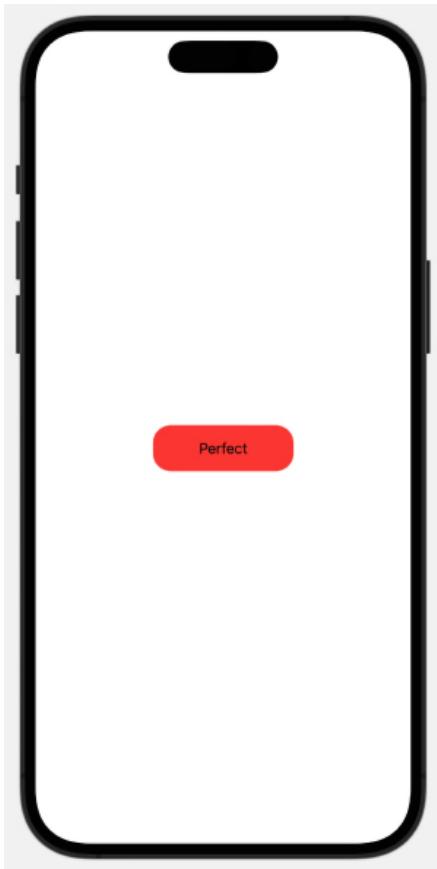


Figure 1.15: Custom view modifier

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

How it works...

A `view modifier` 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 `ViewModifier` protocol and apply our styles in the implementation of the required `body` function. 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, which 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 {
    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 custom 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 `#Preview` macro from the file.
5. Modify the existing struct with the `BlueCircle ViewModifier`:

```
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")
        }
    }
}
```

7. The resulting preview should look like this:

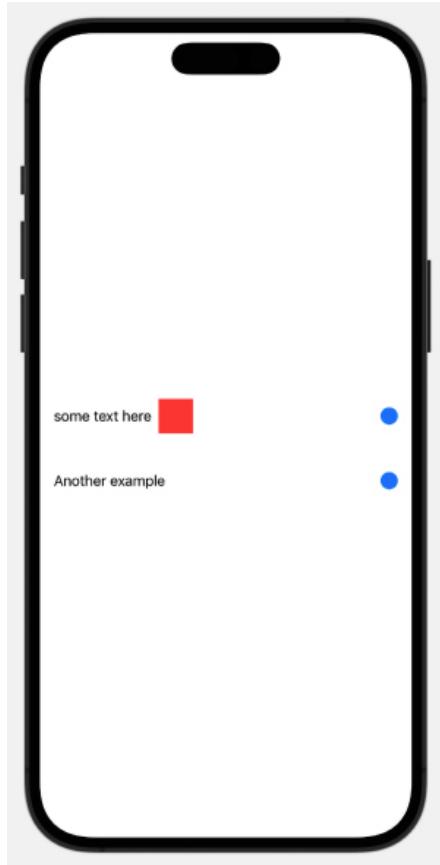


Figure 1.16: ViewBuilder result preview

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 view passed to our `ViewBuilder` will be placed.

See also

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

Simple graphics using SF Symbols

The SF Symbols 5 library provides a set of over 5,000 free consistent and highly configurable symbols. Each year, Apple adds more symbols and symbol variants to the collection.

You can download and browse through a list of SF symbols using the macOS app available for download here: <https://developer.apple.com/sf-symbols/>.

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 **UsingSF Symbols**.

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 entire body content 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. We continue working on our `VStack` content and embed another `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")
}
.foregroundStyle(.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 Picker view with a segmented style that changes the appearance of the Wi-Fi SF Symbol based on the picker selection:

```
HStack {
    Picker("Pick One", selection: $wifiSelection) {
        Text("No Wifi").tag(0)
        Text("Searching").tag(1)
        Text("Wifi On").tag(2)
    }
    .pickerStyle(.segmented)
    .frame(width: 240)
    .padding(.horizontal)
    Group {
        switch wifiSelection {
        case 0:
            Image(systemName: "wifi")
                .symbolVariant(.slash)
        case 1:
            Image(systemName: "wifi")
                .symbolEffect(.variableColor.iterative.reversing)
        default:
            Image(systemName: "wifi")
```

```
        .foregroundStyle(.blue)
    }
}
.foregroundStyle(.secondary)
.font(.title)
}
.padding()
```

5. Let's add the `@State` property to fix the Xcode error. Immediately below the declaration of the `ContentView` struct, add the `wifiSelection` property:

```
@State private var wifiSelection = 0
```

6. 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 levels, 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 levels 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 the `.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 another HStack with a segmented picker for a Wi-Fi system image where we change the appearance based on the status of the `wifiSelection` state variable. The picker reads the state variable and changes the `wifi` symbol appearance from a slashed symbol when “No Wifi” is selected to a variable color animated symbol when “Searching” is selected to a solid blue symbol when “Wifi On” is selected. Here, we used the new **Symbols** framework introduced in iOS 17, and the `.symbolEffect` view modifier to add an animation to a symbol. When we want to add animations to a symbol, the SF Symbols Mac app allows us to configure all the animations and preview the result. We can even export the animation configuration to add it in Xcode.

See also

- SF Symbols in SwiftUI: <https://developer.apple.com/videos/play/wwdc2021/10349>
- What's new in SF Symbols 5: <https://developer.apple.com/videos/play/wwdc2023/10197>
- Animate symbols in your app: <https://developer.apple.com/videos/play/wwdc2023/10258>

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. Many improvements and new APIs have been added to SwiftUI since its introduction, to the point that, at the time of this writing, we can create an app in SwiftUI without using any UIKit components.

However, if you're dealing with legacy code written in UIKit, and have the need to integrate the code in your SwiftUI app, Apple provides a way to do this. UIViews and UIViewControllerControllers can be seamlessly placed inside SwiftUI views and vice versa.

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()
        }
    }
}
```

4. Let's open the `ContentView.swift` file and replace the struct with 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 private var animate = true
    var body: some View {
        VStack{
            ActivityIndicatorAnimating: animate)
            HStack{
                Toggle(isOn: $animate){
                    Text("Toggle Activity")
                }
                }.padding()
            }
        }
    }
```

5. The resulting `ContentView` preview should look like this:

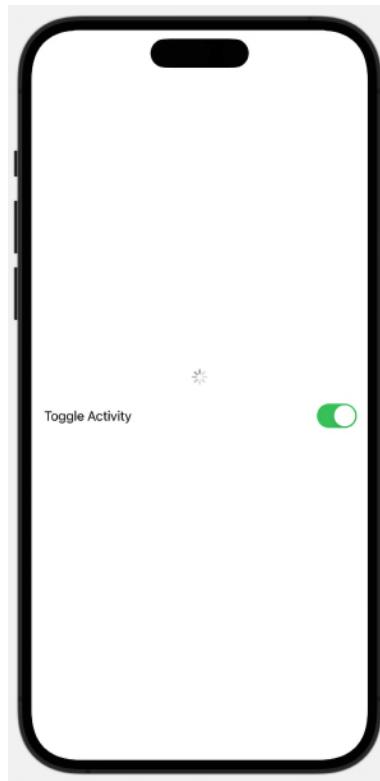


Figure 1.18: UIKit `UIActivityIndicatorView` inside our SwiftUI view

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`. The purpose of the recipe was to show how to integrate a UIKit view with SwiftUI.

See also

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

Apple's tutorial on how to integrate UIKit and SwiftUI:

<https://developer.apple.com/tutorials/swiftui/interfacing-with-uikit>

Adding SwiftUI to a legacy UIKit 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. This recipe assumes prior knowledge of UIKit and it is most useful to developers who want to integrate SwiftUI into a legacy UIKit app. If this is not your case, feel free to skip to the next recipe.

We'll be making use of 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 a 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-3rd-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:

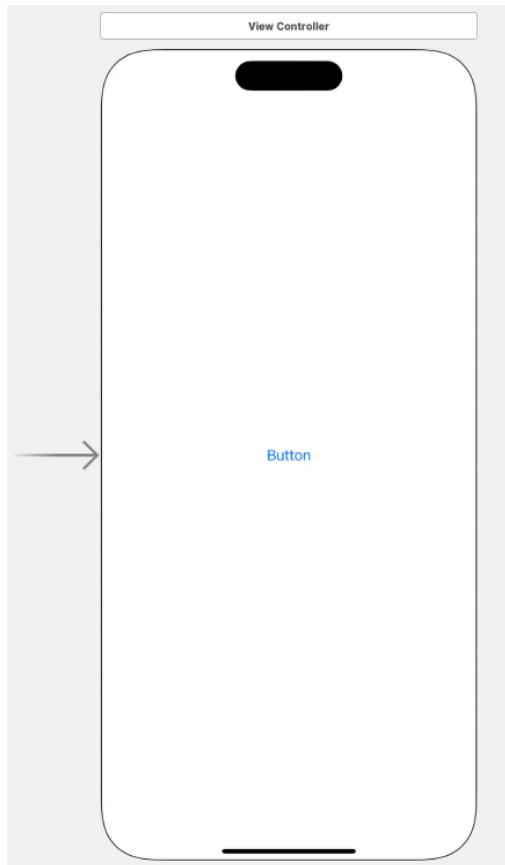


Figure 1.19: UIKit View Controller

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 right of the Xcode window.

- b. In the new window, select the Objects library, type hosting in the search bar, select **Hosting View Controller**, and drag it out to the storyboard:

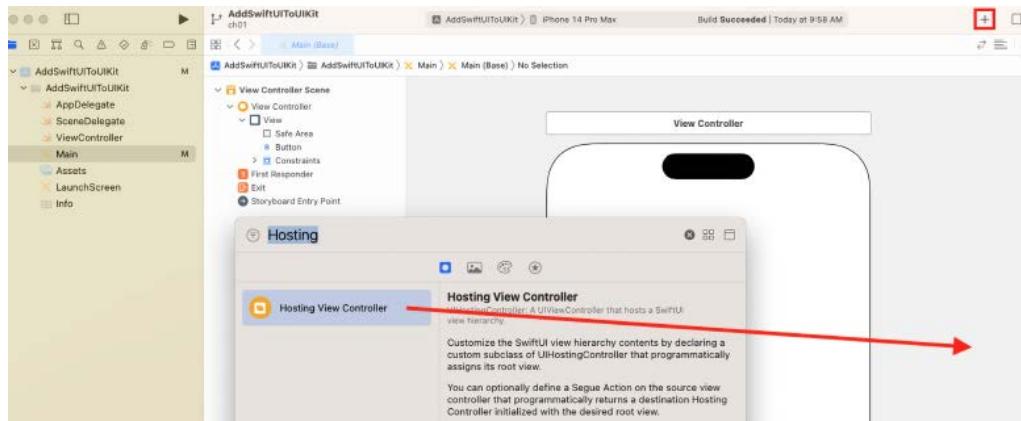


Figure 1.20: Creating a UIKit Hosting View Controller

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:

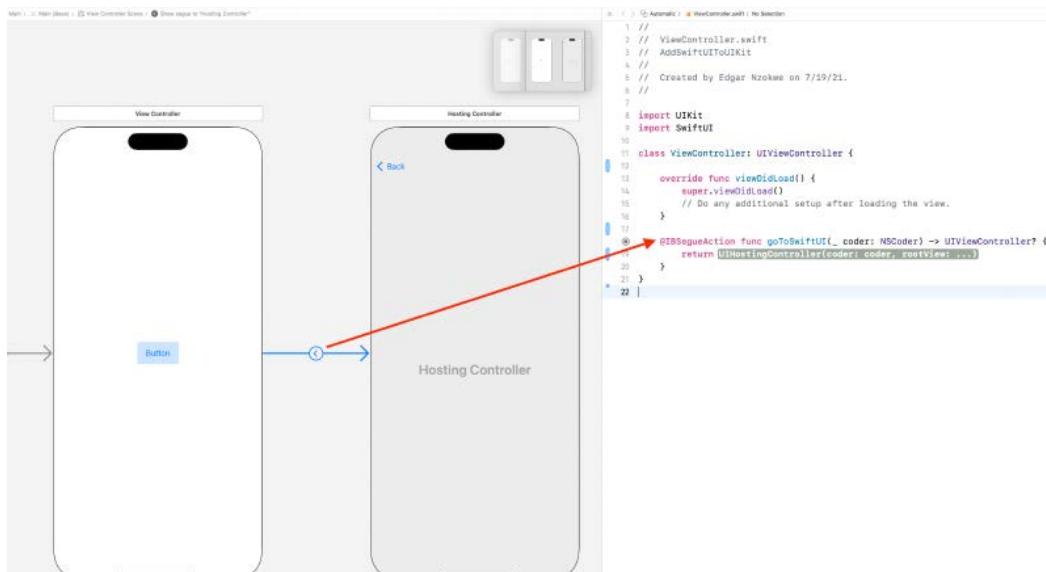


Figure 1.22: Xcode with the Assistant editor open

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:

```
@IBSegueAction func goToSwiftUI(_ coder: NSCoder) ->
UIViewController? {
    return #UIHostingController(coder: coder, rootView: ...)#
}
```

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

```
import SwiftUI
```

12. Within the `goToSwiftUI` function, create a 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 `UIHostingController`, which is a special `ViewController` 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)
}
```

13. At this point, the code will not compile because we have not yet implemented a `Greetings` view. Let's resolve that now.
14. Create a `SwiftUI` view to display a message:
 - a. Click **File | New | File** and select **SwiftUI View**.
 - b. Name the view `Greetings.swift`.

15. Add a `View` component that displays some text passed to it:

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

#Preview {
    Greetings(textFromUIKit: "Hello, World!")
}
```

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 **Navigation View 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

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` is 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 group the controls in `Section` instances in a `List` view. `Section` allows us to include an optional header. The steps are given here:

- Just below the `ContentView` struct declaration, 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"
```

- Replace the body contents with a `List` view with a `Section` view, two `ProgressView` views, and a `Button` view:

```
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 Link:

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

6. Next, add a TextEditor:

```
Section(header: Text("TextEditor")) {
    TextEditor(text: $someText)
```

```
        Text("current editor text:\n\n(someText)")  
    }
```

7. 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 'real magic'"){  
                someText = "real magic"  
            }  
            Button("Turn first picker gray") {  
                color = Color.gray  
            }  
        }  
    }  
}
```

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

```
List {  
    ...  
}  
.listStyle(.grouped)
```

9. The resulting view app preview should look like this:

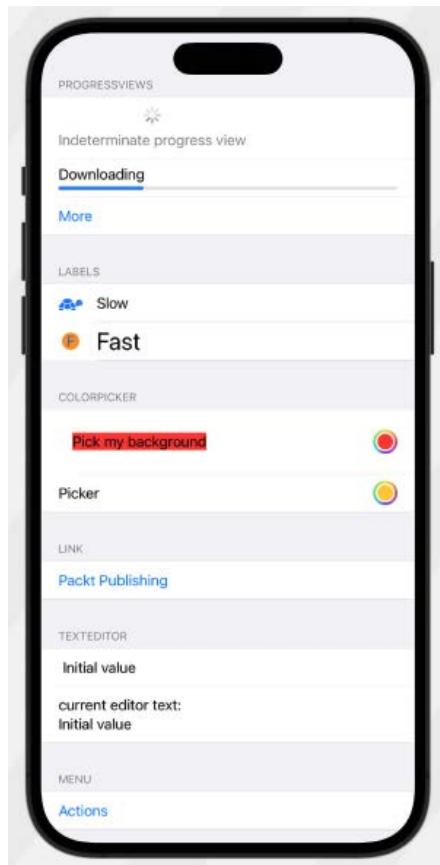


Figure 1.23: More Views and Controls app preview

How it works...

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

Indeterminate `ProgressView` requires 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()
}
```

`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 'real magic') {
            someText = "real magic"
        }
        Button("Turn first picker gray") {
            color = Color.gray
        }
    }
}
```

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.

After learning the basics of SwiftUI, we concluded the chapter with this recipe, where we used several SwiftUI view components that we could incorporate into our apps.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

<https://packt.link/swiftUI>



2

Displaying Scrollable Content with Lists and Scroll Views

In this chapter, we'll learn how to display scrollable content with scroll views and lists in SwiftUI. Scrollable content is a solution for when we need to show information to the user that doesn't fit on a single screen. Using a drag gesture, the user can scroll through the content we display, usually a list of items. We have plenty of examples of this in iOS, such as the mail app, showing a list of our emails, or the stocks app, showing a list of our favorite stocks.

`ScrollView` is a powerful container view used to scroll content and is used when the content is somehow unrelated. For logically related content, we usually use a `List`.

`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 powerful list features, 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
- Creating searchable lists with scopes

Technical requirements

The code in this chapter is based on Xcode 15.0 and iOS 17.0. You can download and install the latest version of Xcode from the App Store. You'll also need to be running macOS Ventura (13.4) or newer.

Simply search for Xcode in the App Store and select and download the latest version. Launch Xcode and follow any additional installation instructions that your system may prompt you with. Once Xcode has fully launched, you're ready to go.

All the code examples for this chapter can be found on GitHub at <https://github.com/PacktPublishing/SwiftUI-Cookbook-3rd-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 5,000 symbols provided by Apple at the time of this writing.

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. Add a state variable to make the scroll indicators flash programmatically:

```
@State private var flashIndicators = false
```

3. Replace the entire body of ContentView with a VStack, with two ScrollView instances and a Button. Each scroll view will include a ForEach struct:

```
var body: some View {
    VStack {
        ScrollView {
            ForEach(letters, id: \.self) {letter in
                Image(systemName: letter)
                    .font(.largeTitle)
                    .foregroundStyle(.yellow)
                    .frame(width: 50, height: 50)
                    .background(.blue)
                    .symbolVariant(.circle.fill)
            }
        }
        .frame(width:50, height:200)
        ScrollView(.horizontal) {
            HStack{
                ForEach(letters, id: \.self){name in
                    Image(systemName: name)
                        .font(.largeTitle)
                        .foregroundStyle(.yellow)
                        .frame(width: 50, height: 50)
                        .background(.blue)
                        .symbolVariant(.circle.fill)
                }
            }
        }
        .scrollIndicatorsFlash(trigger: flashIndicators)
        .padding(.bottom)
        Button("Flash ") {
            flashIndicators.toggle()
        }
    }
    .scrollIndicators(.hidden, axes: .vertical)
}
```

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

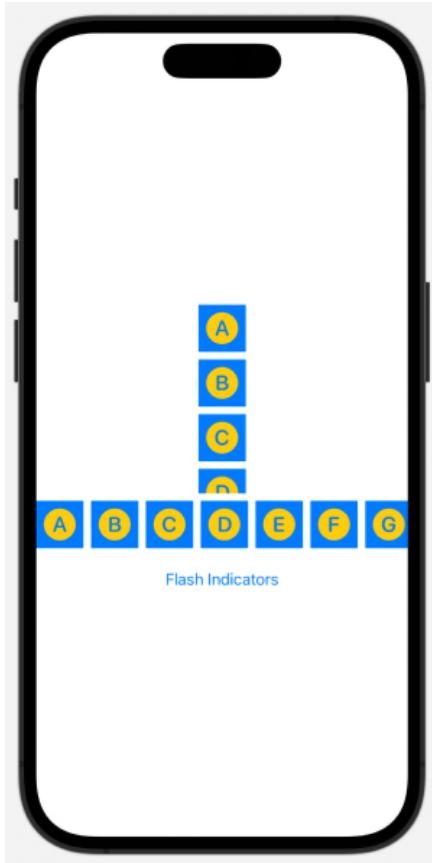


Figure 2.1: The WeScroll app with horizontal and vertical scroll views

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 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, we used the `scrollIndicatorsFlash(trigger:)` modifier to flash the scroll view indicators when the button with the label Flash Indicators is tapped. The button's action changes the value of the `flashIndicators` Boolean state variable, which triggers the scroll indicators to show up momentarily. This is a way to programmatically trigger the visibility of the scroll indicators to give the user feedback on the scrolling capabilities of the view.

Finally, we applied the `scrollIndicators(_:axes:)` modifier to our `VStack`:

```
.scrollIndicators(.hidden, axes: .vertical)
```

This modifier gets two parameters: the first one indicates the visibility of the scroll indicators and the `axes` parameter specifies which indicators are affected. With this particular use of the modifier, we instructed SwiftUI to hide the vertical scroll indicator for all the scrollable views inside the `VStack`. This modifier makes our vertical scroll view hide the scroll indicator, while the horizontal scroll view is not affected and does display the scroll indicator.

A final remark about the two view modifiers we mentioned is that they affect any type of scrollable view, not only `ScrollView` instances but also `List` instances.

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.

When we refer to a *list of static items*, we mean that the information that the list displays is predetermined in our code, and it does not change when the app is running. A good example of this use is the iOS settings app, where we have a list of settings that are fixed by the iOS version. Every time we run the settings app, we see the same list, its content, is **static**.

In contrast, a *list of dynamic items* has its content determined every time the app runs. As such, the app needs to accommodate any number of items in the list. One well-known example is the iOS mail app, where we have a list of our latest emails. The content of this list changes every time we run the app, making its content **dynamic**.

In this recipe, we will create an app that 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. Replace the `ContentView` body with a `List` and use the `ForEach` structure to iterate over our `weatherData` collection. Add some font and padding modifiers to improve the styling too:

```
var body: some View {  
    List {  
        ForEach(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:

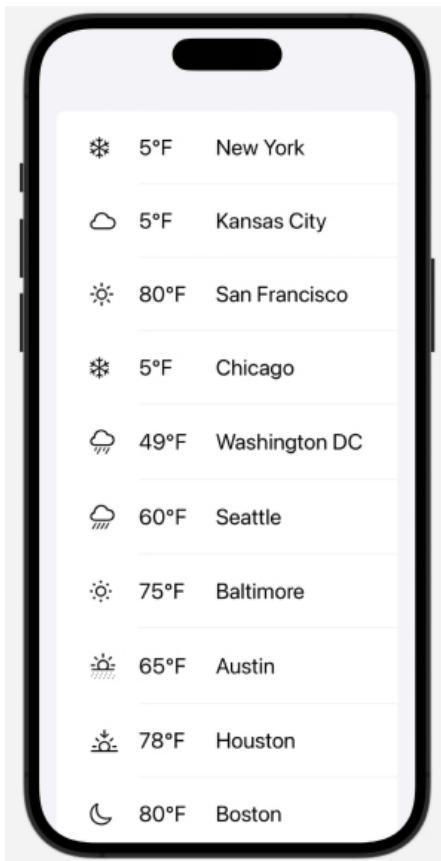


Figure 2.2: Implementing static lists

How it works...

First, we created the `WeatherInfo` struct, which contains properties we'd like to use, such as image name (`image`), temperature (`temp`), 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`. For convenience, we will do this by using an extension with a static property:

```
extension WeatherInfo {  
    static let weatherData = [  
        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:

```
#Preview {
    WeatherRow(weather: WeatherInfo(image: "snow", temp: 5, city: "New
    York"))
}
```

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



Figure 2.3: WeatherRow row preview

8. Open the `ContentView.swift` file and replace the `body` property with a list to display data using the `WeatherRow` component:

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

#Preview {
    ContentView(weatherData: WeatherInfo.weatherData)
}
```

9. Open the `CustomRowsApp.swift` file and modify the initialization of `ContentView` to include the sample data:

```
struct CustomRowsApp: App {
    var body: some Scene {
        WindowGroup {
            ContentView(weatherData: WeatherInfo.weatherData)
        }
    }
}
```

10. The resulting canvas preview should look as follows:

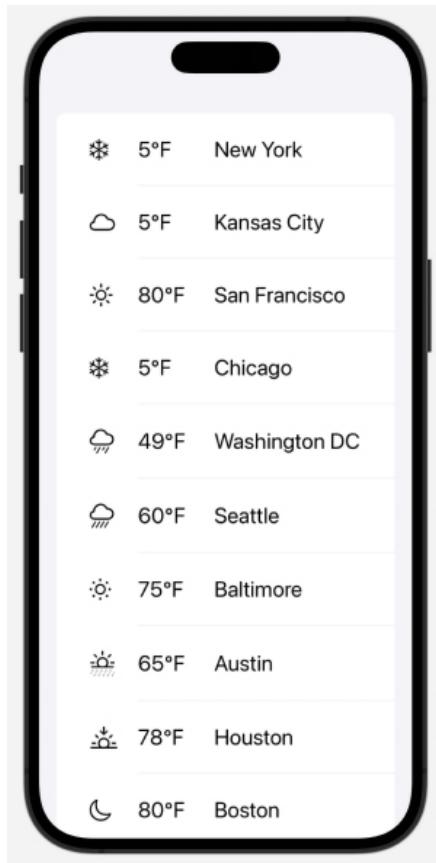


Figure 2.4: CustomRow App preview

Run the app on a device or use the 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 created an array of the `WeatherInfo` instances, `weatherData`, which can be used in other parts of the project previewing and testing areas as we build. We used a static property of `WeatherInfo` so it is available to any type of app without the need to instantiate. We use static properties when we want values available globally to our app. In order to call the static property we need to prepend it with the type name, in our case:

```
WeatherInfo.weatherData
```

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 static `WeatherData` array has been created for preview purposes only. In a production application, your data will likely be 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, and 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. Replace the content of the `body` property of the `ContentView` struct with a `NavigationStack` containing a `List` view:

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

3. Add a `.navigationTitle` and a `navigationBarTitleDisplayMode` modifier to the list with a title and an inline display mode:

```
    .navigationTitle("Number List")
    .navigationBarTitleDisplayMode(.inline)
```

4. Add a `.toolbar` modifier and a `Button` function, containing an action to trigger an element being added to the row:

```
.toolbar{
    Button("Add") {
        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))
}
```

6. The preview should look as follows:

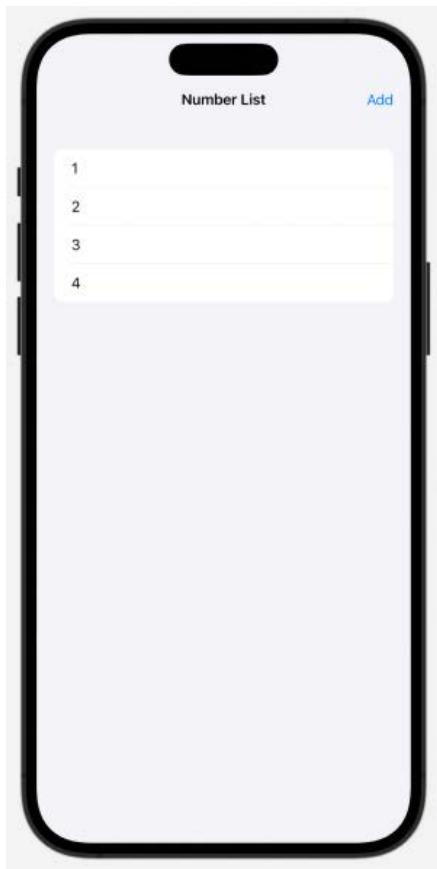


Figure 2.5: *ListRowAdd* preview

You can now run the preview and click the **Add** button and new items will be added 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 `.navigationTitle ("Number List")` modifier adds a title to the top of the list, and with the `.navigationBarTitleDisplayMode(.inline)`, we confine the title 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 `.toolbar(...)` modifier adds a button to the trailing end of the navigation section. The button calls the `addItemToRow` function when clicked.

The toolbar modifier defaults to the `NavigationBar` on iOS and within this, we can place toolbar items, normally using `ToolbarItem()`. However, since there is only one button, this defaults to the `.topBarTrailing` position. We could have used the following, which is equivalent:

```
.toolbar {  
    ToolbarItem(placement: .topBarTrailing) {  
        Button("Add") {  
            addItemToRow()  
        }  
    }  
}
```

Finally, the `addItemToRow` function generates a random number between 5 and 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 private var countries = ["USA", "Canada", "Mexico", "England",  
"Spain", "Cameroon", "South Africa", "Japan", "South Korea"]
```

2. Replace the content of the `body` variable with a `NavigationStack` containing a `List` view that displays our array of countries. Also, include the `onDelete` modifier at the end of the `ForEach` structure:

```
NavigationStack {  
    List {  
        ForEach(countries, id: \.self) {  
            country in  
            Text(country)  
        }  
        .onDelete(perform: deleteItem)  
    }  
    .navigationTitle("Countries")  
    .navigationBarTitleDisplayMode(.inline)  
}
```

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

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

4. The resulting preview should look as follows:

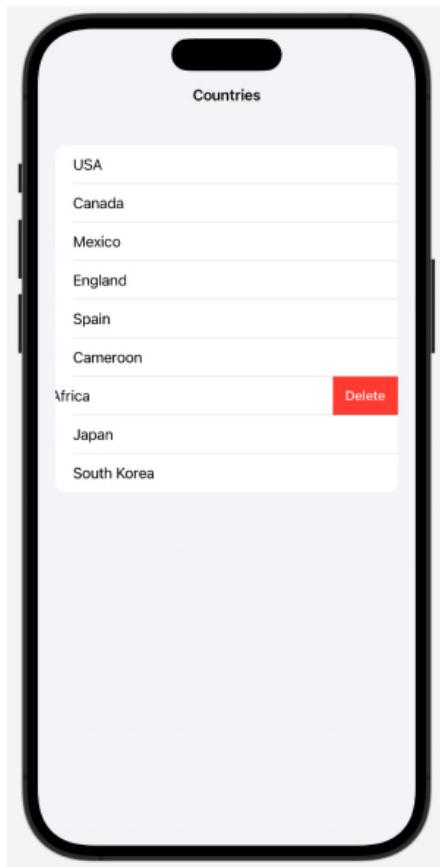


Figure 2.6: *ListRowDelete in action*

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 indexes of the rows to be deleted. The `onDelete` modifier automatically passes the indexes of the items to be deleted.

There's more...

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

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

2. Add a `.toolbar { EditButton() }` modifier to the `body` view, just below the `.navigationBarTitleDisplayMode` 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:

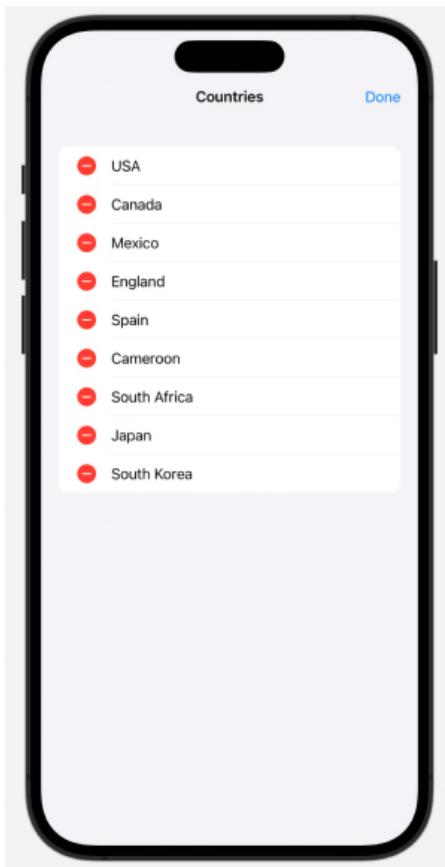


Figure 2.7: *ListRowEdit* app preview during execution

4. Click on the circle to the left of any list item to reveal a delete button to delete the item.

How it works...

The `.toolbar { EditButton() }` modifier adds an **Edit** button to the top-right corner of the display. Once tapped, it triggers the appearance of a minus sign to the left of each item in the modified `List`. Tapping on the minus sign reveals the **Delete** button, which, when tapped, will execute the function in our `.onDelete` modifier and delete the related item from the row.

There's more...

To display the **Edit** button on the left-hand side of the navigation bar will require the use of a `ToolbarItem` since the toolbar button will no longer be in the default `topBarTrailing` position.

Thus, we will need to introduce a `ToolbarItem` with a specific placement, like this:

```
.toolbar {  
    ToolbarItem(placement: .topBarLeading) {  
        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 private var countries = ["USA", "Canada", "Mexico", "England",  
"Spain", "Cameroon", "South Africa", "Japan", "South Korea"]
```

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

```
NavigationStack {  
    List {  
        ForEach(countries, id: \.self) {  
            country in  
                Text(country)  
        }  
        .onMove(perform: moveRow)  
    }  
    .navigationTitle("Countries")  
    .navigationBarTitleDisplayMode(.inline)  
    .toolbar {  
        EditButton()  
    }  
}
```

- 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 preview 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:

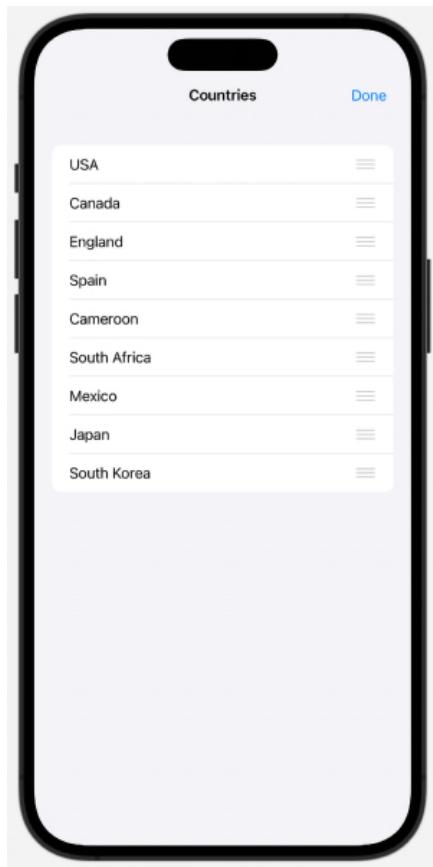


Figure 2.8: MovingListRows

How it works...

To move list rows, you need to wrap the list in a `NavigationStack`, add the `.onMove(perform:)` modifier to the `ForEach` struct, and add a `.toolbar` modifier with an `EditButton` to the list. The `onMove` modifier calls the `moveRow` function when the user drags the row, while `.toolbar` displays the button that triggers the “move mode,” where list row items become movable.

The `moveRow(source: IndexSet, destination: Int)` function takes two parameters, `source` and `destination`, 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 body content with a `NavigationStack`. Wrapping the `List` in a `NavigationStack` allows us to add a title and navigation items to the view:

```
NavigationStack {  
}
```

2. Add a list and section to `NavigationStack` (or body view if you skipped the optional *Step 1*). Also, add a `listStyle` and `navigationTitle` and `navigationBarTitleDisplayMode` modifiers:

```
List {  
    Section(header: Text("North America")){  
        Text("USA")  
        Text("Canada")  
        Text("Mexico")  
        Text("Panama")  
        Text("Anguilla")  
    }  
}  
.listStyle(.grouped)  
.navigationTitle("Continents and Countries")  
.navigationBarTitleDisplayMode(.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")
    }
}
```

4. Your canvas preview should resemble the following:

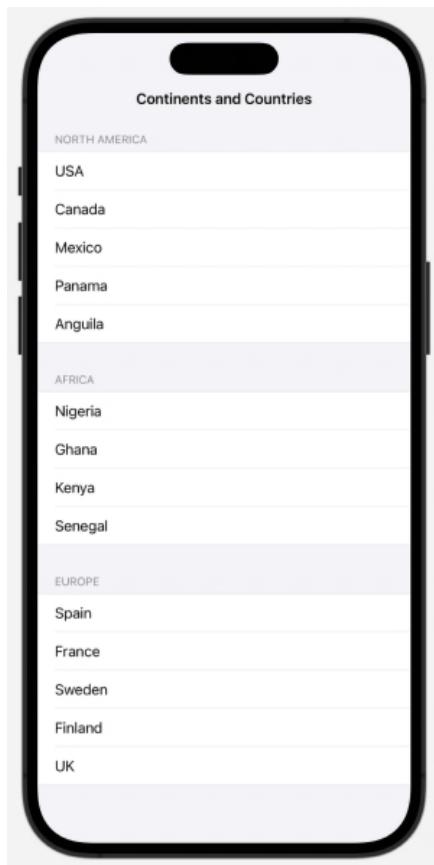


Figure 2.9: *ListWithSections* preview

Looking at the preview, you can see the continent 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  
}
```

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

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

3. Replace the body content with a `List` that displays the collection of todo items in a `TextField` view, which will allow for editing the items:

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

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

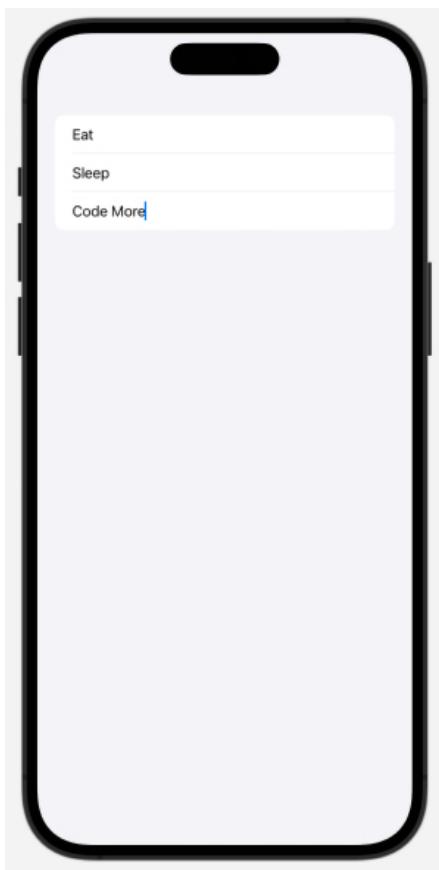


Figure 2.10: Editable collections preview

Tap 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..
```

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.

Ever since the introduction of 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(text:placement:prompt:)` modifier, which marks the view as searchable and configures the display of a search field, 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`.

How to do it...

Let's create an app to search through different types of food. The steps are as follows:

1. Add a new Swift file to the project called `Food`, which will contain the data used for the project. We use a struct to model different types of food. The struct has two properties, a string with the name of the food, and a category, which is an enum representing the type of food:

```
struct Food: Hashable {

    enum Category: String {
        case fruit
        case meat
        case vegetable
    }
    var name: String
    var category: Category
}
```

- Let's add some sample data to illustrate how to search a list. Add a type extension after the struct declaration. Here, we choose to extend an array of Food so our code will look concise thanks to Swift's powerful type inference:

```
extension Food {  
    static let sampleFood: [Food] = [  
        Food(name: "Apple", category: .fruit),  
        Food(name: "Pear", category: .fruit),  
        Food(name: "Orange", category: .fruit),  
        Food(name: "Lemon", category: .fruit),  
        Food(name: "Strawberry", category: .fruit),  
        Food(name: "Plum", category: .fruit),  
        Food(name: "Banana", category: .fruit),  
        Food(name: "Melon", category: .fruit),  
        Food(name: "Watermelon", category: .fruit),  
        Food(name: "Peach", category: .fruit),  
        Food(name: "Pork", category: .meat),  
        Food(name: "Beef", category: .meat),  
        Food(name: "Lamb", category: .meat),  
        Food(name: "Goat", category: .meat),  
        Food(name: "Chicken", category: .meat),  
        Food(name: "Turkey", category: .meat),  
        Food(name: "Fish", category: .meat),  
        Food(name: "Crab", category: .meat),  
        Food(name: "Lobster", category: .meat),  
        Food(name: "Shrimp", category: .meat),  
        Food(name: "Carrot", category: .vegetable),  
        Food(name: "Lettuce", category: .vegetable),  
        Food(name: "Tomato", category: .vegetable),  
        Food(name: "Onion", category: .vegetable),  
        Food(name: "Broccoli", category: .vegetable),  
        Food(name: "Cauliflower", category: .vegetable),  
        Food(name: "Eggplant", category: .vegetable),  
        Food(name: "Swiss Chard", category: .vegetable),  
        Food(name: "Spinach", category: .vegetable),  
        Food(name: "Zucchini", category: .vegetable),  
    ]  
}
```

3. Switch to the `ContentView` file. Before the `ContentView` struct's body, add a `State` variable to hold the search text and add another variable to hold the sample data, in our case an array of `Food` instances, which we populate with the sample data defined in the previous step:

```
@State private var searchText = ""  
let food: [Food] = Food.sampleFood
```

4. Replace the `ContentView` struct's body with a `NavigationStack`, containing a `List`, and to display the search results, a `navigationTitle` modifier, and a `.searchable` modifier:

```
var body: some View {  
    NavigationStack {  
        List {  
            ForEach(searchResults, id: \.self) { food in  
                LabeledContent(food.name) { Text("\\"(food.category.  
rawValue)") }  
            }  
        }  
        .searchable(text: $searchText, prompt: "Search for a food")  
        .navigationTitle("Foods")  
    }  
}
```

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

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

6. The resulting live preview should look as follows:

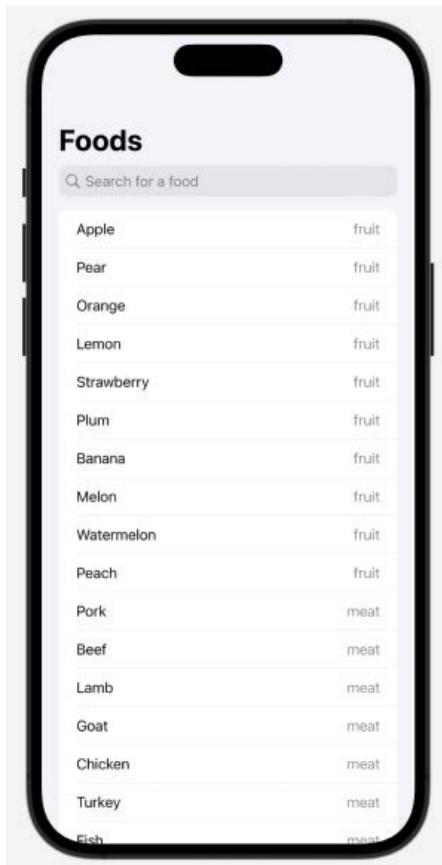


Figure 2.11: Searchable list live preview

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...

We use a `List` view to display a list of foods. The `searchable` modifier applied to the `List` view, adds a search field where the user can enter text to search through the list of foods. The `searchable` modifier takes two parameters, a binding to a string, which represents the text entered in the search field, and a prompt string, which will be displayed as a placeholder when the search field is empty.

We use the `searchText` state variable to hold the value that is being searched for. Each time the user interacts with the search field, the value of `searchText` changes thanks to the binding, which triggers SwiftUI to update the `List` view, because it depends on the computed property, `searchResults`, which depends on `searchText`.

The value of `searchResults` is used by the `List` view in the `ForEach` struct to display a filtered list of items based on the searched text.

For each item in the list, we use the `LabeledContent` container view to display the name and the category of the food.

Using searchable lists with scopes

In this recipe, we will improve our searchable list with the addition of the `searchScopes(_:activation:_:)` modifier, which allows us to narrow the scope of our searches.

The scope modifier could be useful to further reduce the scope of our search and reduce the time it takes to find an item. In our case, we have a list of food items, belonging to three different categories: meat, fruit, and vegetables. Selecting a specific scope narrows the search and gives the user a quicker way of finding the desired item.



This new modifier was introduced in iOS 16.4 and it is a nice mid-cycle addition, introduced with a minor version update of iOS. Most of the updates to iOS are usually announced in June, during the WWDC conference, and then introduced in September when the new version of iOS is released to users. However, Apple also releases new features during the year as is the case for the one we are using in this recipe.

Getting ready

In this recipe, we will continue with the `SearchableLists` project and add more code to it.

How to do it...

Let's improve our searchable list by providing a way to narrow the scope of the search for food, by food category:

1. Starting from the code of the previous recipe, open `ContentView.swift`. Add an enum and a new property, between the `ContentView` struct declaration and the `searchText` property:

```
enum FruitSearchScope: Hashable {
    case all
    case food(Food.Category)
}
@State private var scope: FruitSearchScope = .all
```

2. Add the `searchScopes(_:activation:_:)` modifier to the `List` view, between the `searchable` modifier and the `navigationBarTitle`:

```
.searchScopes($scope, activation: .onSearchPresentation) {
    Text("All").tag(FruitSearchScope.all)
    Text("Fruit").tag(FruitSearchScope.food(.fruit))
```

```
    Text("Meat").tag(FruitSearchScope.food(.meat))
    Text("Vegetable").tag(FruitSearchScope.food(.vegetable))
}
```

3. Modify the `searchResults` computed property to filter the results by the scope. The new code should be like this:

```
var searchResults: [Food] {
    var food: [Food] = self.food
    if case let .food(category) = scope {
        food = food.filter { $0.category == category }
    }
    if !searchText.isEmpty {
        food = food.filter { $0.name.lowercased().
contains(searchText.lowercased())}
    }
    return food
}
```

4. Test the app in the preview canvas. Enter any text in the search field and see how the scope picker appears. Select different options from the picker and observe how the list of foods is filtered by category and foods that contain the text searched for.

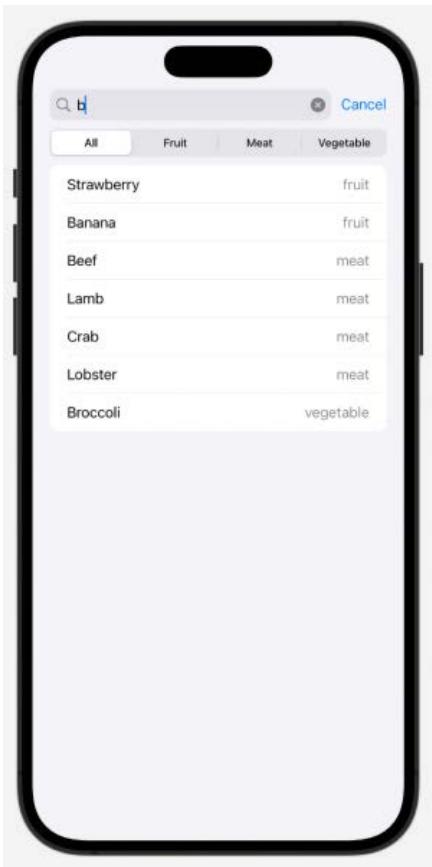


Figure 2.12: Searchable List live preview, filtered by text and scope

How it works...

The `searchScopes(_:activation:_:)` modifier takes three parameters, a binding to a scope variable, which defines the scope of the search, an activation style to decide when the scope selection control will be displayed, and a `viewBuilder` that represents the scoping options SwiftUI uses to populate a **picker**.

To define the scope of the search, we use an `enum`, `FruitSearchScope`, with the cases `.all` and `.food(Food.Category)`. The first case is used to search the entire list of foods, while the second case has an *associated value* representing the food category used to narrow the search. The current scope is stored in the `@State` property scope. We want to populate a picker that will contain the four possible search scopes: all the foods, only the fruits, only the meats, or only the vegetables.

We provide a simple Text view for each scope and we associate a tag of type `FruitSearchScope` to each of them. These views will be used by SwiftUI to populate a picker.

As our last step, we need to modify the search to consider the search scope selected by the user. This only applies if the search scope is the second case, `.food(Food.Category)`. We use an `if-case-let` statement to filter the foods by the category chosen, before searching for the text entered in the search field.

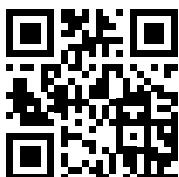
See also

Apple's documentation on adding a search feature to a SwiftUI app: <https://developer.apple.com/documentation/swiftui/search>

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

<https://packt.link/swiftUI>



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 stacks and lazy grids are named lazy because they do not load in memory all the data when they are created. Instead, they load only the data that they need to display on the screen at a given moment. For example, with a list of 100 items, where only a subset of 10 items is displayed on the screen, the lazy container view will load in memory the subset of 10 items plus the item before and the item after the subset. This behavior optimizes memory usage and results in better UI performance; it allows our apps to deal with large datasets without affecting the user experience.

We'll also learn how to present hierarchical data in an expandable list with sections. Finally, we'll introduce widgets, one of my favorite iOS features that's only available in SwiftUI. 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
- 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 15.0 and iOS 17.0. You can download and install the latest version of Xcode from the App Store. You'll also need to be running macOS Ventura 13.5 or newer.

Simply search for Xcode in the App Store and select and download the latest version. Launch Xcode and follow any additional installation instructions that your system may prompt you with. Once Xcode has fully launched, you're ready to go.

All the code examples for this chapter can be found on GitHub at <https://github.com/PacktPublishing/SwiftUI-Cookbook-3rd-Edition/tree/main/Chapter03-Advanced-Components/>.

Using LazyHStack and LazyVStack

SwiftUI 2.0 introduced the LazyHStack and LazyVStack views. They can be used similarly 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 the 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. In the `ContentView`'s `body` property, replace the content with 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. Continuing with the body property of the `ContentView`, add a second `ScrollView` to the `VStack`, just below the `.frame` modifier:

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

4. 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`:

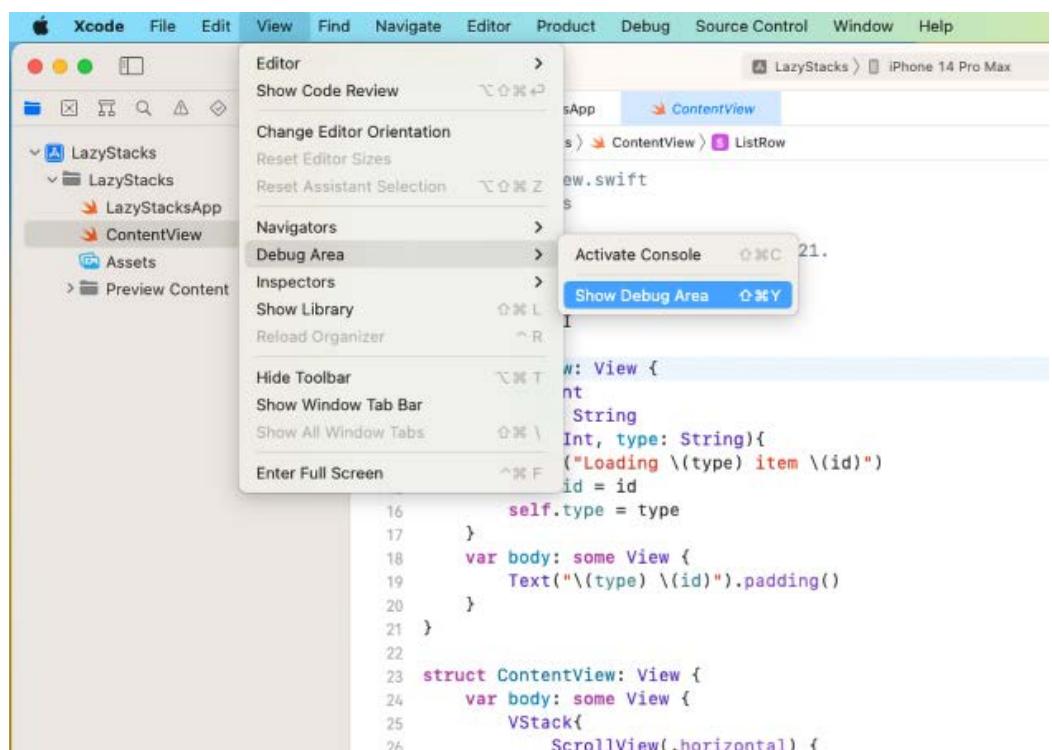


Figure 3.1: Show Debug Area

5. From the Xcode toolbar, select the type of device that you want to use to run the app:



Figure 3.2: Xcode toolbar with iPhone 14 Pro Max selected

6. 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 entire body content 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 try the app in the interactive canvas preview, or you launch the app on a device or a simulator, scroll through the numbers while watching the debug area print statements. Make sure to select between `Executable` or `Previews` to see the print statements. You'll notice that only a subset of the `ListRow` views is initialized as you scroll: that's lazy loading at work. Only the loading of 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 a 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 body content 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)")
        .foregroundStyle(.white)
        .padding(EdgeInsets(top: 30, leading: 15, bottom: 30,
trailing: 15))
        .background(colors[index % colors.count])
        .clipShape(Capsule())
    }
}
```

6. If all these steps were performed successfully, the preview should look as follows:



Figure 3.4: LazyHStack and LazyVStack

Run the app in canvas preview mode and scroll horizontally or vertically through each grid. Scrolling stays 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 columns 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

In iOS 17, `ScrollView` received a series of improvements. One of these is the `scrollPosition(id:)` modifier, which associates a binding to be updated when the scroll view scrolls. The binding allows us to read the ID of the topmost or leading-most view shown in the scroll view at any given moment. We can also set the binding to the ID of our choice and the scroll view will scroll programmatically to show the chosen view.

If we are targeting previous versions of iOS, we must use `ScrollViewReader` to read the scroll position and to scroll programmatically. `ScrollViewReader` allows us to programmatically scroll to a different view in the scroll view, even if the view is not 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, to showcase how to scroll programmatically, we will create an app that displays a list of characters from A to Q vertically. A button at the top will programmatically scroll from the top to the end of the scroll view when tapped. Another button at the bottom of the scroll view will allow us to scroll back up from the bottom to the middle.

We will display two similar scroll views, side by side, in an `HStack`. One scroll view will use the modern approach to scrolling programmatically and the other scroll view will use the legacy approach.

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. Immediately after the `CharacterInfo` declaration, add an extension with a static array of `CharacterInfo` instances called `charArray`. Initialize it with `CharacterInfo` struct representing the SF Symbols for the characters A-Q:

```
extension CharacterInfo {
    static 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),  
]  
}
```

3. In the ContentView struct add a @State variable to keep track of the position of the scroll view:

```
@State private var scrolledID: CharacterInfo.ID?
```

4. Replace the Text view in content of the body struct with an HStack with two embedded VStack instances. Each VStack will include a Text view and a vertical ScrollView. The Text view will be the title of each scroll view:

```
var body: some View {  
    HStack(spacing: 20) {  
        VStack {  
            Text("iOS 17+").  
                .foregroundStyle(.blue).  
                .font(.title)  
            ScrollView {  
                // scrolling content will go here  
            }  
        }  
        VStack {  
            Text("iOS 14+").  
                .foregroundStyle(.blue).  
                .font(.title)  
            ScrollView {  
                // legacy code will go here  
            }  
        }  
    }  
}
```

- Let's add a button at the top of the iOS 17+ scroll view to programmatically scroll to the end. The button will set the `scrolledID` property to 16, which is the ID of the last image view we will add to the scroll view in the next step:

```
var body: some View {
    HStack(spacing: 20) {
        VStack {
            Text("iOS 17+")
                .foregroundStyle(.blue)
                .font(.title)
            ScrollView {
                Button("Go to letter Q") {
                    scrolledID = 16
                }
                .padding()
                .background(.blue)
                .tint(.yellow)
                // more code will follow
            }
            .scrollPosition(id: $scrolledID)
        }
        VStack {
            Text("iOS 14+")
                .foregroundStyle(.blue)
                .font(.title)
            ScrollView {
                // legacy code will go here
            }
        }
    }
}
```

- Now let's use a `ForEach` to create the image views from the `CharacterInfo` instances in `charArray`. Each image view will display the SF symbol for a letter with a circle and fill. To scroll programmatically to any of these image views, we will assign to each one a unique ID:

```
var body: some View {
    HStack(spacing: 20) {
        VStack {
            Text("iOS 17+")
                .foregroundStyle(.blue)
                .font(.title)
```

```

ScrollView {
    LazyVStack {
        Button("Go to letter Q") {
            scrolledID = 16
        }
        .padding()
        .background(.blue)
        .tint(.yellow)
        ForEach(CharacterInfo.charArray) { image in
            Image(systemName: image.name)
                .font(.largeTitle)
                .foregroundStyle(.blue)
                .frame(width: 90, height: 90)
                .background(.yellow)
                .padding()
        }
        // more code to follow
    }
}
.scrollPosition(id: $scrolledID)
}
VStack {
    Text("iOS 14+")
        .foregroundStyle(.blue)
        .font(.title)
    ScrollView {
        // legacy code will go here
    }
}
.padding(.horizontal)
}
.padding(.horizontal)
}

```

7. Finally, let's add a button at the end of the scroll view, which will allow us to scroll halfway up to the letter G with an ID of 6. We will enclose the assignment of the value 6 to the `scrolledID` variable in a `withAnimation` block, which will provide a nice animation for the scrolling action:

```

var body: some View {
HStack(spacing: 20) {
    VStack {
        Text("iOS 17+")
    }
}

```

```
        .foregroundStyle(.blue)
        .font(.title)
    ScrollView {
        LazyVStack {
            Button("Go to letter Q") {
                scrolledID = 16
            }
            .padding()
            .background(.blue)
            .tint(.yellow)
            ForEach(CharacterInfo.charArray) { image in
                Image(systemName: image.name)
                    .font(.largeTitle)
                    .foregroundStyle(.blue)
                    .frame(width: 90, height: 90)
                    .background(Color.yellow)
                    .padding()
            }
            Button("Go to letter G") {
                withAnimation {
                    scrolledID = 6
                }
            }
            .padding()
            .background(.blue)
            .tint(.yellow)
        }
    }
    .scrollPosition(id: $scrolledID)
}
VStack {
    Text("iOS 14+")
        .foregroundStyle(.blue)
        .font(.title)
    ScrollView {
        // legacy code will go here
    }
}
.padding(.horizontal)
}
```

```
.padding(.horizontal)  
}
```

The resulting app preview should look as follows:

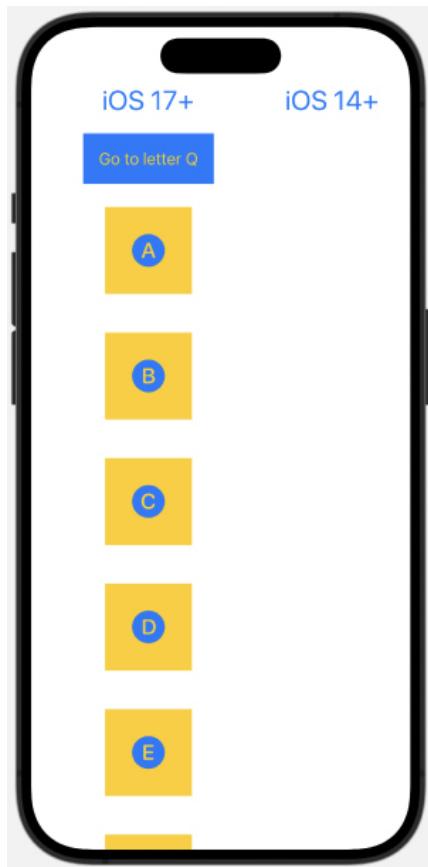


Figure 3.5: ScrollViewReaders app preview

Use the canvas live preview to interact with the app. Tap on the **Go to letter Q** button to programmatically scroll down to the letter **Q**. Then tap on the button at the bottom of the screen to scroll up to the letter **G** with a nice animation.

- Now, let's tackle the scroll view using the legacy `ScrollViewReader` struct. Replace the second `VStack` with the following:

```
 VStack {
    Text("iOS 14+")
        .foregroundStyle(.blue)
        .font(.title)
    ScrollView {
        ScrollViewReader { proxy in
            Button("Go to letter Q") {
                proxy.scrollTo(16)
            }
            .padding()
            .background(.yellow)
            ForEach(CharacterInfo.charArray) { image in
                Image(systemName: image.name)
                    .font(.largeTitle)
                    .foregroundStyle(.yellow)
                    .frame(width: 90, height: 90)
                    .background(.blue)
                    .padding()
            }
            Button("Go to letter G") {
                withAnimation {
                    proxy.scrollTo(6, anchor: .top)
                }
            }
            .padding()
            .background(.yellow)
        }
    }
}.padding(.horizontal)
```

If everything went well, the app preview should look as follows:

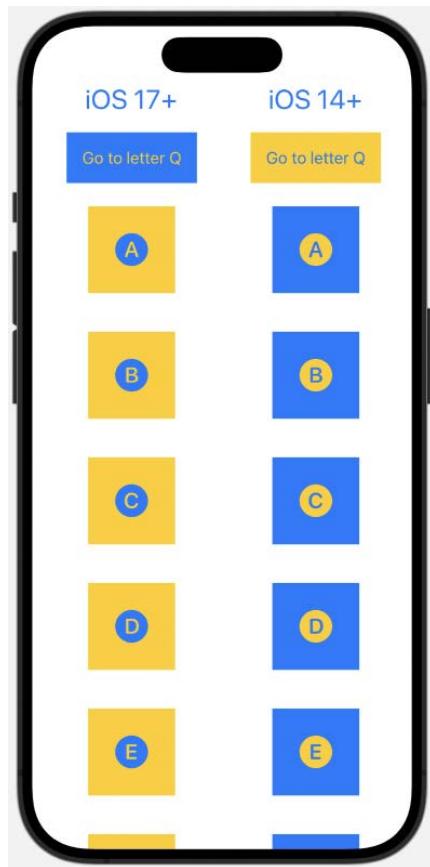


Figure 3.6: ScrollViewReaders app preview

Interact with the app in the live preview of the Xcode canvas and play with both scroll views and its buttons.

How it works...

The `CharacterInfo` struct conforms to the `Identifiable` protocol and has two properties: a `name` and an `id`. The `id` parameter will be used to assign an ID to each of the views inside the scroll view so we can scroll programmatically to any such views.

In the `ForEach` struct, we generate `ImageView` instances from the `CharacterInfo` elements in the `charArray` variable. We use the `name` property to instantiate an `ImageView` with the corresponding SF symbol. We have added 16 images to the scroll view, each one with a different ID.

With the modern approach, we add the `scrollPosition(id:)` modifier to the scroll view and pass a binding to the `scrolledID` variable. As the scroll view scrolls, the value of the variable will change to contain the ID of the topmost image view shown in the scroll view. To scroll programmatically, we will just assign to the `scrolledID` variable the value of the ID of the image view to which we want to scroll. For the button at the top, we assign the value 16, which is associated with the image view representing the letter Q, and for the button at the end, we assign the value 6, which represents the letter G.

With the legacy approach, we need to use `ScrollViewReader`. We embed `ScrollViewReader` in the `ScrollView` so that we have access to an instance of `ScrollViewProxy`, which provides the method `scrollTo()`. This method can be used to programmatically scroll to the child view whose ID has been specified. The `scrollTo()` method also has an optional anchor parameter that is used to specify the position of the item we are scrolling to. For example, `scrollTo(6, anchor: .top)` causes the scroll view to scroll until the image view with ID 6, representing the letter G, is shown at the top of the scroll view. Change the anchor to `.bottom` and use the canvas to see how, when we scroll to the letter G, it shows at the bottom of the screen instead of the top.

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]?
}
```

- 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: "coloncurrencyssign.circle", content: [dollar, yen])
```

- 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")
```

- 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 2. `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 content with a `List` view that displays the contents of our `bagContent` array:

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

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

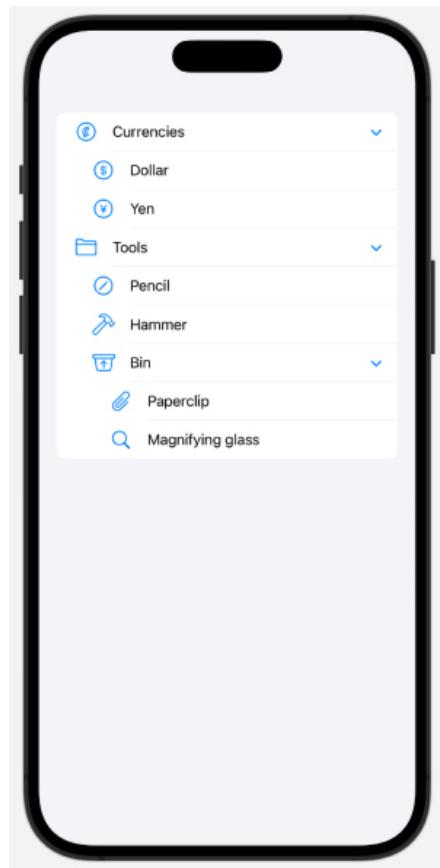


Figure 3.7: Using expanding lists

Run the 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 created 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:

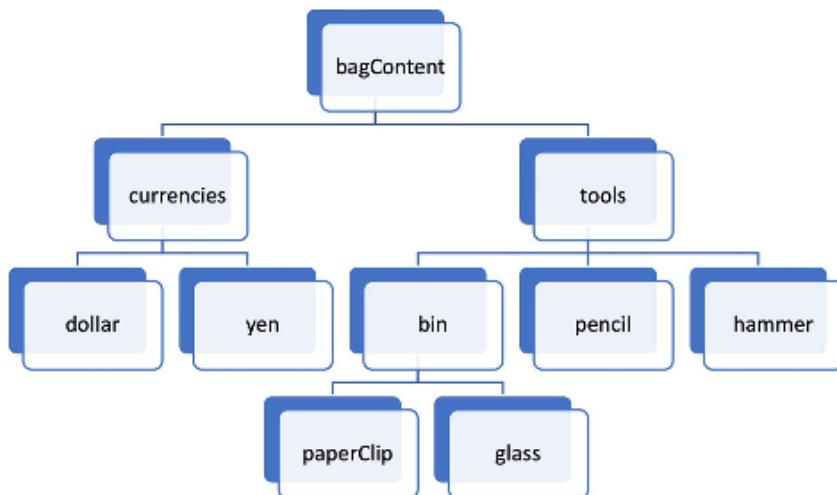


Figure 3.8: Hierarchical view of `bagContent`

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

```
List(bagContents, children: \.content) { row in
    Label(row.name, systemImage row.icon)
}
```

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

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 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. In the `ContentView` struct, replace the content of the `body` variable 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")  
     }  
 }.padding()  
}
```

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

```
 var body: some View {  
     VStack {  
         DisclosureGroup(isExpanded: $showplanets) {  
             Text("Mercury")  
             Text("Venus")  
             DisclosureGroup("Earth") {  
                 Text("North America")  
                 Text("South America")  
                 Text("Europe")  
                 Text("Africa")  
                 Text("Asia")  
                 Text("Antarctica")  
                 Text("Oceania")  
             }  
         } label: {  
             Text("Planets")  
         }  
     }.padding()  
}
```

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:

```
var body: some View {
    VStack {
        DisclosureGroup(isExpanded: $showplanets) {
            // Content not shown here
            Text("Mercury")
            Text("Venus")
            DisclosureGroup("Earth") {
                Text("North America")
                Text("South America")
                Text("Europe")
                Text("Africa")
                Text("Asia")
                Text("Antarctica")
                Text("Oceania")
            }
        } label: {
            Text("Planets")
        }
        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))
                .foregroundStyle(.blue)
        }
    }.padding()
}
```

The resulting preview should be as follows:

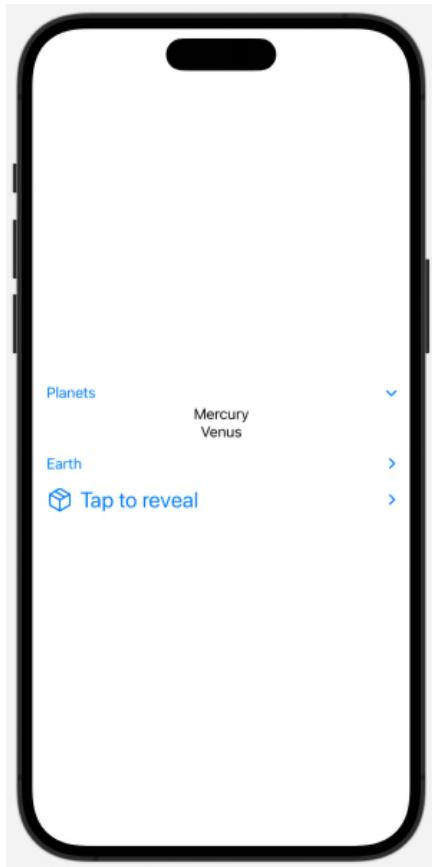


Figure 3.9: DisclosureGroup preview

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 finding out 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

Apple provides the `WidgetKit` framework to show glanceable and relevant content from an app as widgets in iOS and macOS and complications on watchOS. Examples of the most popular widgets are Apple's weather and stock apps.

There are two kinds of widget 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-3rd-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 sample tasks and a standalone sample task are declared in the file. We use these to populate the views of the app.

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

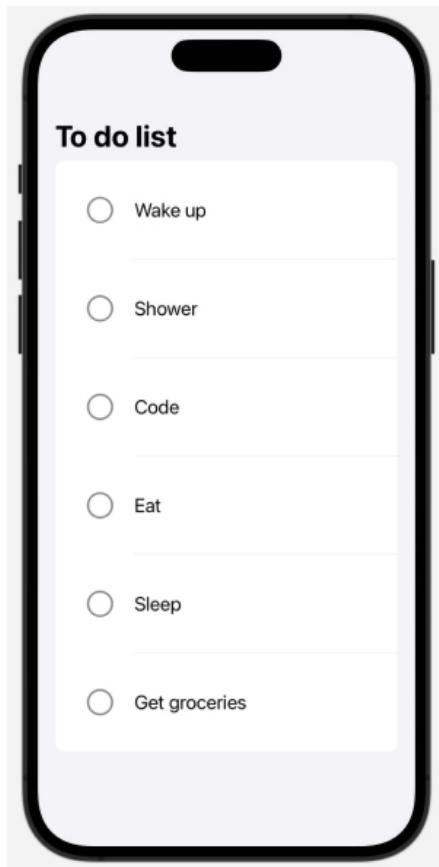


Figure 3.10: The TodoList app's original state

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:

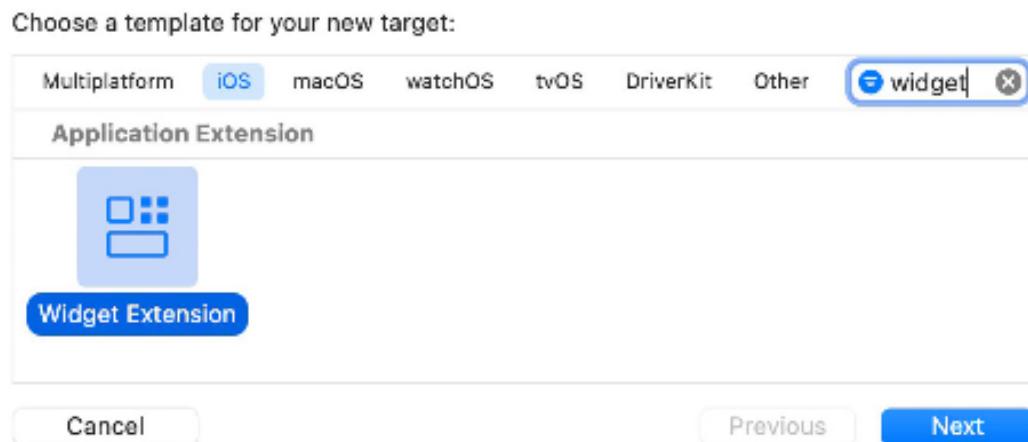


Figure 3.11: Selecting the Widget Extension target

5. Click **Next**. Set Product Name to TodoWidget and make sure that **Include Live Activity** and **Include Configuration Intent** are unchecked:

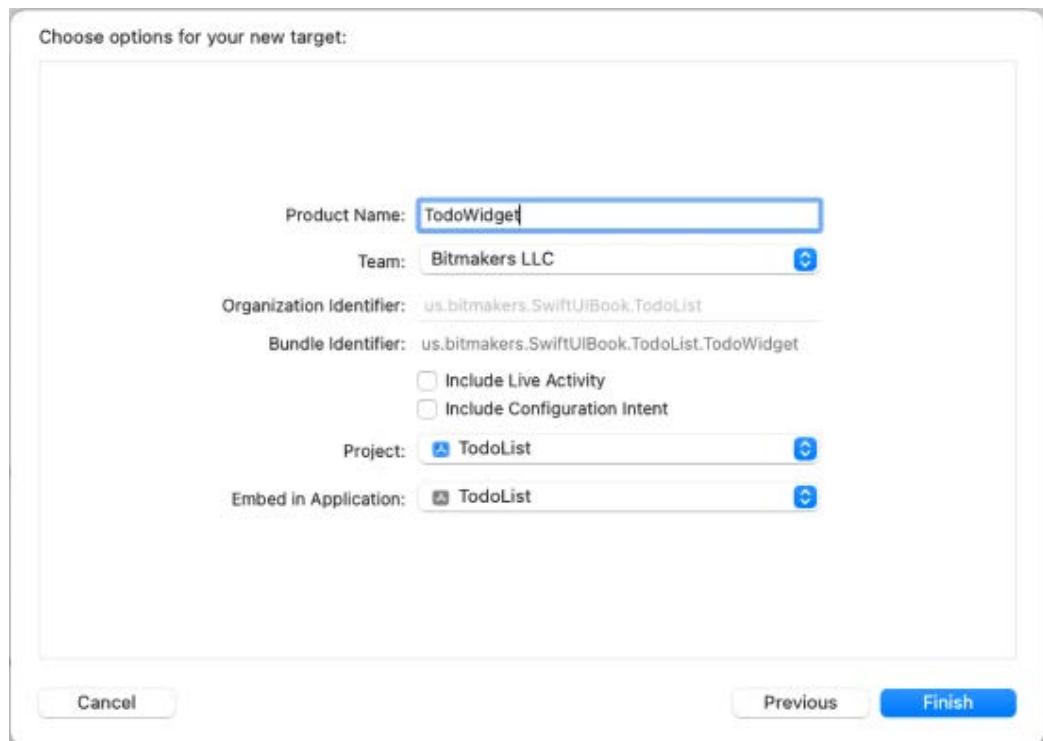


Figure 3.12: Setting the widget's name

6. Click **Activate** on the next pop-up alert.
7. Open the `TodoWidget.swift` file in the newly created `TodoWidget` folder. It contains some template code for a widget that updates the current time every hour.
8. Delete all the content of `TodoWidget.swift` except for the `import` statements.
9. 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
}
```

10. Xcode will display an error regarding the type `Task` of the `task` property of `TaskEntry`.
11. Resolve the error by making the content of `Tasks.swift` available to our `WidgetKit` extension.
12. 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:

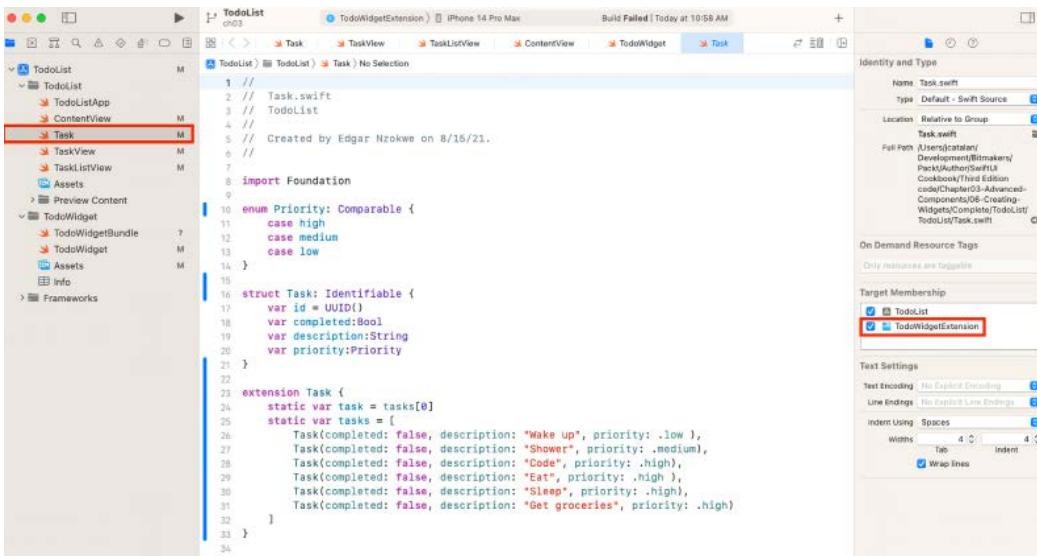


Figure 3.13: Adding `Task.swift` to our `TodoWidgetExtension` target

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

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

14. 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:

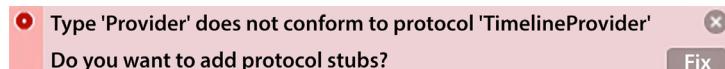


Figure 3.14: Xcode build error notification

15. 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
```

16. 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.
17. Now, update the `placeholder` and `getSnapshot` functions to display a sample task declared in the `Tasks.swift` file. We use Swift's powerful type inference to omit the type of the static property `Task.task`, just by typing `.task`. The `Provider` struct should look as follows:

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

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

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

typealias Entry = TaskEntry
```

18. Delete `typealias Entry = TaskEntry`, because Swift can infer `Entry` from the conformance to the `TimelineProvider` protocol.

19. 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 = Task.tasks.sorted(by: {$0.priority >  
        $1.priority})  
    for index in 0..        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)  
    }  
    let timeline = Timeline(entries: entries, policy: .atEnd)  
    completion(timeline)  
}
```

20. 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 {  
        VStack {  
            Image(systemName: imageName(forTask: entry.task))  
            Text(entry.task.description)  
        }.containerBackground(for: .widget) {  
            Color(.widgetBackground)  
        }  
    }  
    private func imageName(forTask task: Task) -> String {  
        switch task.priority {  
        case .high: "arrowshape.up.circle"  
        case .medium: "arrowshape.forward.circle"  
        case .low: "arrowshape.down.circle"  
        }  
    }  
}
```

The `WidgetBackground` color has not been defined yet and Xcode shows an error. Let's add the color now.

21. Click on `Assets.xcassets` | `WidgetBackground` | `Universal` and select `systemYellowColor` from the `Content` color picker:

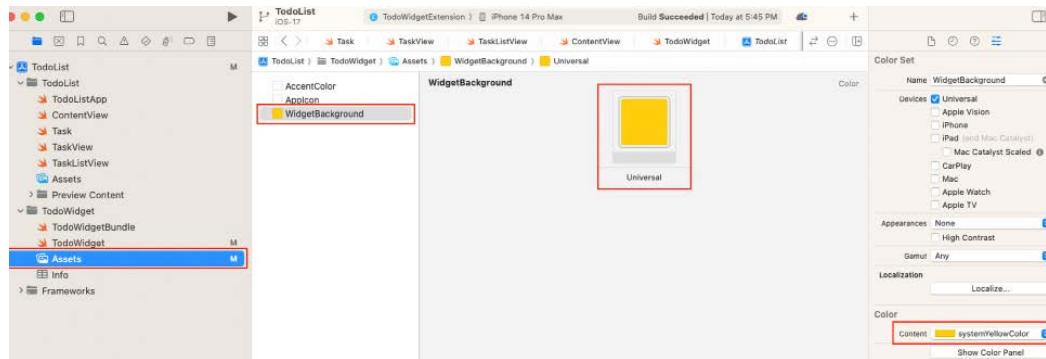


Figure 3.15: Setting up the color of `WidgetBackground`

22. Let's put everything together with a `TodoWidget` struct that conforms to the `Widget` protocol:

```
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")
            .supportedFamilies([.systemSmall, .systemMedium])
        )
    }
}
```

23. Now, let's add a couple of `#Preview` macros so that we can preview the widget timeline as well as the widget in two different sizes:

```
#Preview("small", as: .systemSmall) {
    TodoWidget()
} timelineProvider: {
    Provider()
}

#Preview("medium", as: .systemMedium) {
    TodoWidget()
} timeline: {
    TaskEntry(date: Date(), task: Task(completed: false, description:
"Wake up", priority: .low))
    TaskEntry(date: Date(), task: Task(completed: false, description:
"Shower", priority: .medium))
    TaskEntry(date: Date(), task: Task(completed: false, description:
"Code", priority: .high))
}
```

24. Preview the widget in the canvas. Switch between the small and medium previews. The result should look something like this:



Figure 3.16: The preview of the small TodoList widget



Figure 3.17: The preview of the medium TodoList widget

25. If you run the widget in the iOS simulator, you can 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.



Figure 3.18: The small TodoList widget running in the iOS simulator

Thanks to the preview macros and timeline preview available in Xcode 15, there is a faster way to observe the timeline of a widget. Use the first preview, named **small**, to interact with the timeline. Use the playback buttons at the bottom to play, advance, or go back on the timeline entries, or tap on individual entries to see how the widget changes.

Notice how the changes on the widget are animated by default.

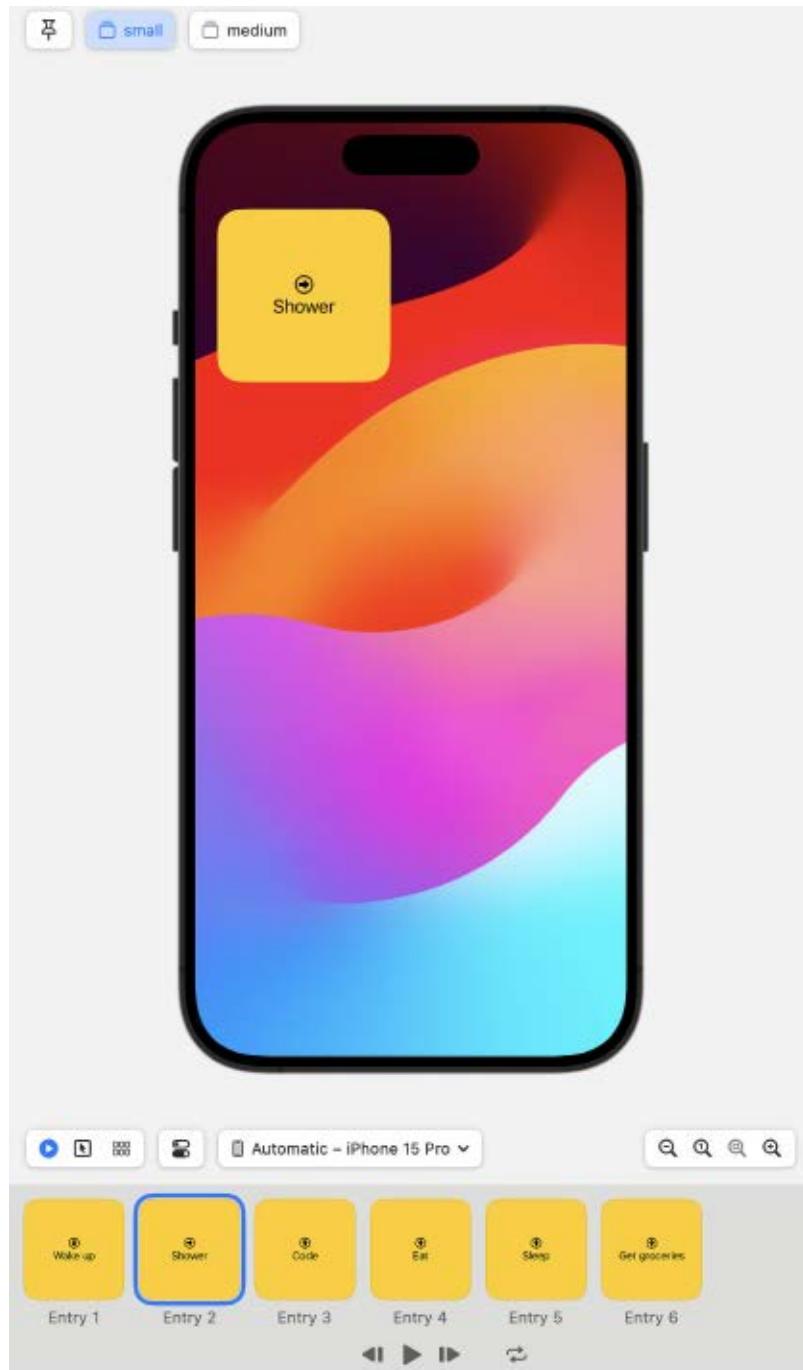


Figure 3.19: The timeline preview of TodoWidget

How it works...

In a widget, an entry is a struct that holds some data and a date property used by WidgetKit to update the widget view at certain time intervals. 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 fewer 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 WidgetKit: <https://developer.apple.com/documentation/widgetkit>

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

<https://packt.link/swiftUI>



4

Viewing while Building with SwiftUI Preview in Xcode 15

Developing an application requires several interactions between users and developers. For example, users 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. Xcode and SwiftUI solve this problem by introducing canvas previews. Previews allow for live viewing of UI changes without building the app and running it on the simulator.

Xcode 14 improved the canvas and live previews with a new user interface and faster updates. Xcode 15 with Swift 5.9 brought new preview macros and native previews for UIKit, AppKit, Widgets, and Live activities. The biggest improvement is that we can change the device settings from the Canvas and preview our SwiftUI views without having to resort to code changes. We can even see these variations side by side on the canvas with the click of a button.

In this chapter, we will learn how to make effective use of Xcode previews to speed up the UI development time. We will use a dual approach, using the Xcode canvas without any code changes, and using some code to create previews programmatically.

This chapter includes the following recipes:

- Using the live preview canvas in Xcode 15
- Previewing a view in a `NavigationStack`
- Previewing a view with different traits
- Previewing a view on different devices
- Using previews in `UIKit`
- Using mock data for previews

Technical requirements

The code in this chapter is based on Xcode 15.0 and iOS 17.0. You can download and install the latest version of Xcode from the App Store. You'll also need to be running macOS Ventura (13.5) or newer.

Simply search for Xcode in the App Store, and select and download the latest version. Launch Xcode and follow any additional installation instructions that your system may prompt you with. Once Xcode has fully launched, you're ready to go.

All the code examples for this chapter can be found on GitHub at <https://github.com/PacktPublishing/SwiftUI-Cookbook-3rd-Edition/tree/main/Chapter04-Viewing-while-building-with-SwiftUI-Preview/>.

Using the live preview canvas in Xcode 15

In Xcode 15, the canvas feels more responsive than ever, and when we change the SwiftUI code, the live updates happen much faster than in older versions of Xcode. In this recipe, we will cover the live preview canvas in Xcode. We will learn how to make the best use of previews to become more productive and increase the speed of our development.

Getting ready

Create a new SwiftUI app named `CanvasPreview`.

How to do it...

In this recipe, we will create a view that will change its appearance, depending on the three device settings supported by the canvas: color scheme, orientation, and dynamic type. We will also preview the different variants side by side. The steps are as follows:

1. 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 `PostView`.
2. In the `PostView` struct, create the following three properties:

```
var title: String  
var text: String  
var imageName: String
```

3. Replace the content of the body of the `PostView` struct with the following code:

```
GeometryReader { geometry in  
    ScrollView {  
        if geometry.size.height > geometry.size.width {  
            // View for portrait orientation  
            VStack(spacing: 20) {  
                Image(systemName: imageName)  
                    .symbolRenderingMode(.multicolor)  
                    .resizable()
```

```
        .aspectRatio(contentMode: .fit)
        .frame(width: geometry.size.width * 0.75)
    VStack(alignment: .leading) {
        Text(title)
            .font(.title2)
            .padding(.top)
        Text(text)
            .padding(.top)
    }
}
.padding()
} else {
    // View for landscape orientation
    HStack(alignment: .top, spacing: 20) {
        Image(systemName: imageName)
            .symbolRenderingMode(.multicolor)
            .resizable()
            .aspectRatio(contentMode: .fit)
            .frame(height: geometry.size.height * 0.55)
        VStack(alignment: .leading) {
            Text(title)
                .font(.title2)
                .padding(.top)
            Text(text)
                .padding(.top)
        }
    }
    .padding()
}
}
```

4. Adjust the preview to have a title and use some sample data. This will get rid of the error that Xcode is showing. Replace the preview code with:

```
#Preview("PostView") {
    PostView(title: "Weather forecast",
              text: "Scattered thunderstorms. Potential for heavy
rainfall. Low 68F. Winds light and variable. Chance of rain 60%",
              imageName: "cloud.bolt.rain")
}
```

The preview should look as follows:



Figure 4.1: PostView preview

5. Open the `ContentView.swift` file.
6. Above the `body` variable, add an `@Environment` variable that checks for the color scheme of the current device:

```
    @Environment(\.colorScheme) var deviceColorScheme
```

7. Replace the content of the body variable of the ContentView with a NavigationStack, which will conditionally include two different instances of PostView, depending on the deviceColorScheme variable. We will also have a different navigation title for each of the two views:

```
NavigationStack {  
    if deviceColorScheme == .light {  
        // View for light appearance color scheme  
        PostView(title: "Captivating force of light",  
                 text: "The captivating force of light lies in its  
ability to ignite our senses and draw us into a mesmerizing world of  
beauty and wonder. It dances across the landscape, casting intricate  
patterns and creating a symphony of colors. Whether it cascades through  
a stained-glass window, dapples through a canopy of trees, or reflects  
off a shimmering body of water, light possesses an inherent allure that  
captures our attention. It has the power to transform ordinary scenes  
into extraordinary ones, unveiling the hidden details and textures  
that might otherwise go unnoticed. Light creates a dynamic interplay  
between shadows and highlights, enhancing the depth and dimension of our  
surroundings, and evoking an emotional response that resonates within  
us.",  
                 imageName: "sun.max.fill")  
        .navigationTitle("Light colors")  
    } else {  
        // View for dark appearance color scheme  
        PostView(title: "Fascinating Dark Secrets",  
                 text: "The concept of fascinating darkness captivates  
the imagination, drawing us into a realm of mystery and intrigue. Within  
the folds of darkness, hidden wonders and secrets lie waiting to be  
discovered. It is in this absence of light that our senses heighten,  
and our perception expands beyond the ordinary. Fascinating darkness  
challenges our understanding of the world, inviting us to explore the  
depths of our own thoughts and emotions. It is a canvas upon which our  
imagination can paint vivid scenes and narratives, sparking creativity  
and contemplation. In the captivating embrace of darkness, we find  
solace, inspiration, and the opportunity for self-discovery.",  
                 imageName: "moon.stars.fill")  
        .navigationTitle("Dark colors")  
    }  
}
```

The resulting preview should look as follows:

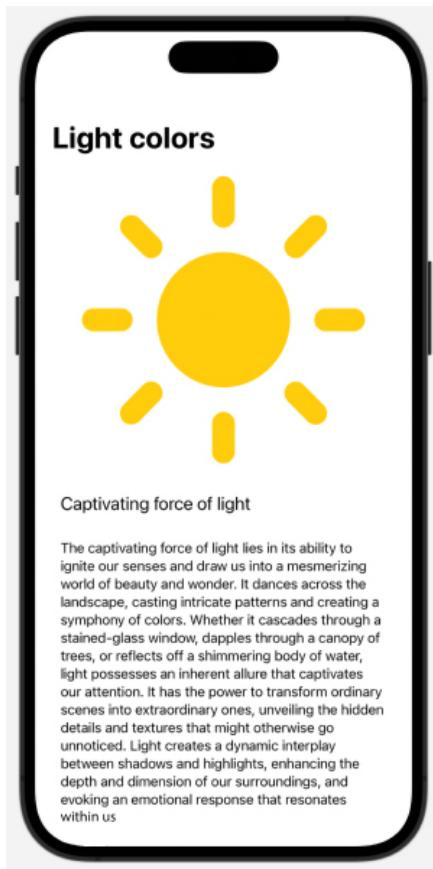


Figure 4.2: ContentView preview

8. Modify the #Preview macro for ContentView to include a title. The preview code should be the following:

```
#Preview("ContentView") {  
    ContentView()  
}
```

9. Make sure that the ContentView preview is displayed on the canvas, and take some time to look at the controls on the canvas. They should look like the following:

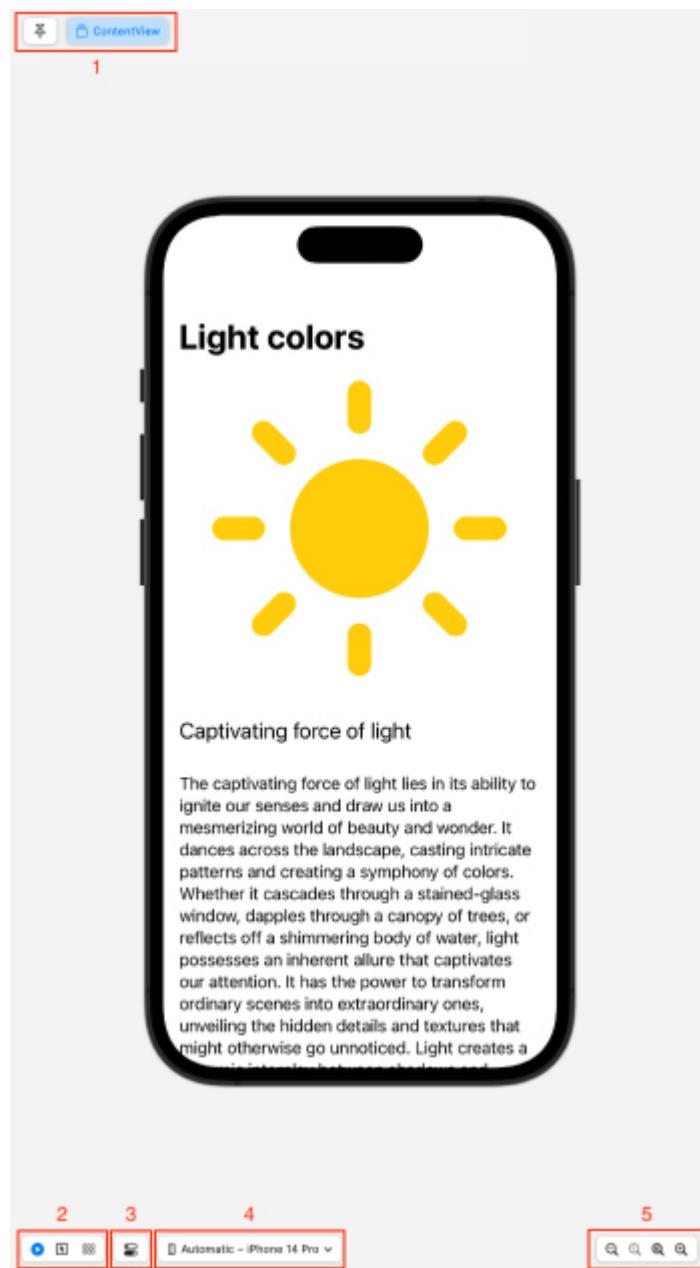


Figure 4.3: Xcode 15 canvas showing the live preview

The controls are grouped into five sections with the following functionality:

1. A **Pin** button to “pin” the preview so that when you move to a different file, the preview stays visible. To the trailing edge of the **Pin** button, we have one button that represents the preview shown, or several buttons that represent the different previews shown. The buttons display the custom titles used in the preview macros, or **Preview** if no title is passed to the preview macro.
2. Three buttons that control the preview mode. In trailing order, we can choose from the live preview, selected by default, selectable preview mode, or the device variants preview.
3. The device settings selector. Allows you to select the orientation, color scheme, and dynamic type text.
4. The preview device selector, which allows you to preview using a different iPhone or iPad than the default setting.
5. The zoom settings controls. In leading to trailing order, we find these controls: **Zoom out**, **Zoom to 100%**, **Zoom to fit**, and **Zoom in**.

We will cover the functionality of the canvas in detail with some examples, using the app we have just created in this recipe.

10. Switch to the **PostView** file, and choose the **selectable** preview from the canvas. Click on the text **Weather forecast** on the preview. Observe how Xcode highlights the SwiftUI code, in lines 26–28, which are responsible for creating the text view that displays **Weather forecast**. In the right pane, we see the attributes of the **Text** view, a **Title2** font, and padding at the top:

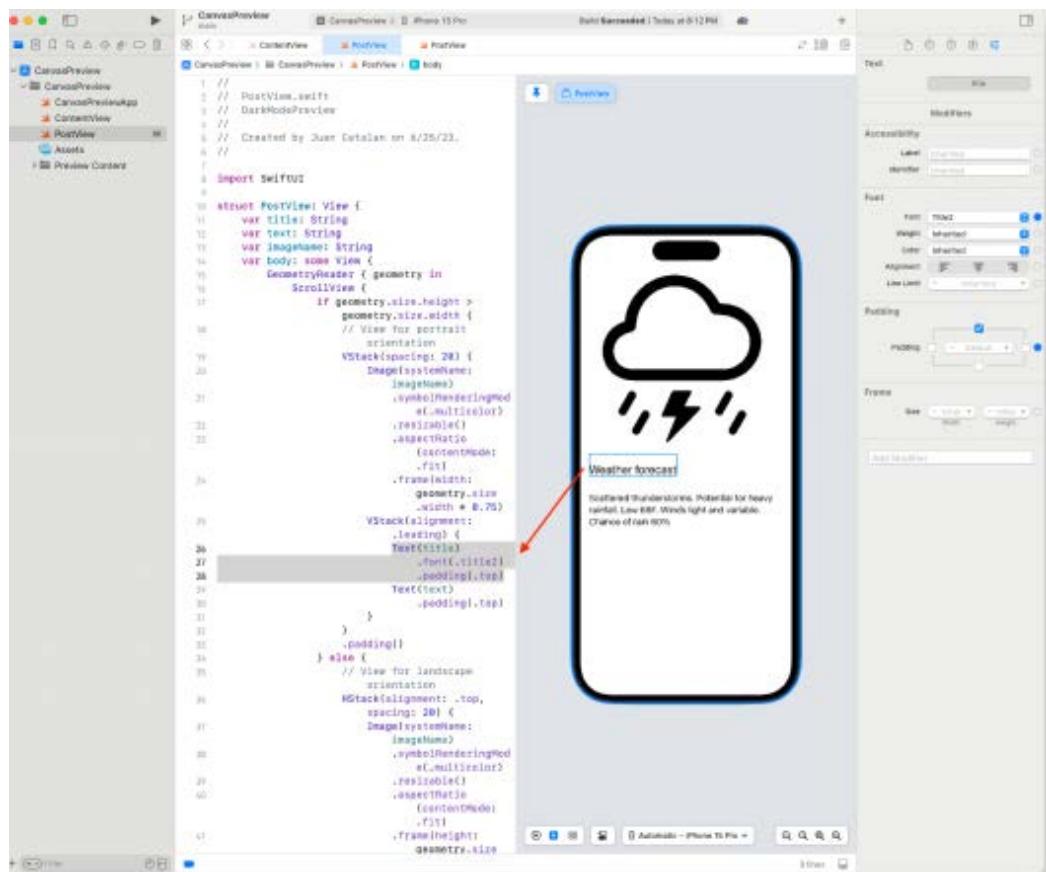


Figure 4.4: Xcode 15 showing the selectable preview

- Now, let's see how we can preview different device settings variants side by side. Switch back to the **ContentView** file, make sure that the preview is displayed on the canvas, and then click on the **variants** menu button to reveal a small pop-up menu with three options.

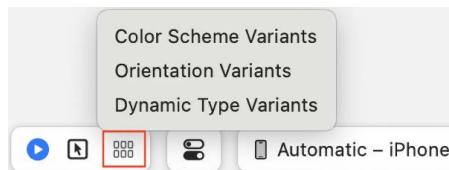


Figure 4.5: Variants button showing the pop-up menu

12. Play with the settings, and see how the canvas shows all the variations of the specific property selected side by side. If we click on one of the variants, Xcode will bring the variant to the canvas and hide the rest of the variants. With one variant displayed, a Back button appears at the top-left of the canvas to return to the side-by-side preview. Below are some of these side-by-side previews:



Figure 4.6: Color scheme variants side by side



Figure 4.7: Dynamic type variants side by side

13. The preview shown by default includes our device in **Portrait** orientation, with a **Light Landscape Left** color scheme and **Large** dynamic type. Let's play with these three settings and see how the canvas preview reacts to the changes. Make sure that the **ContentView** preview is displayed on the canvas, and then click on the **Device Settings** button to reveal a pop-up window with the controls we will use to change the device settings.

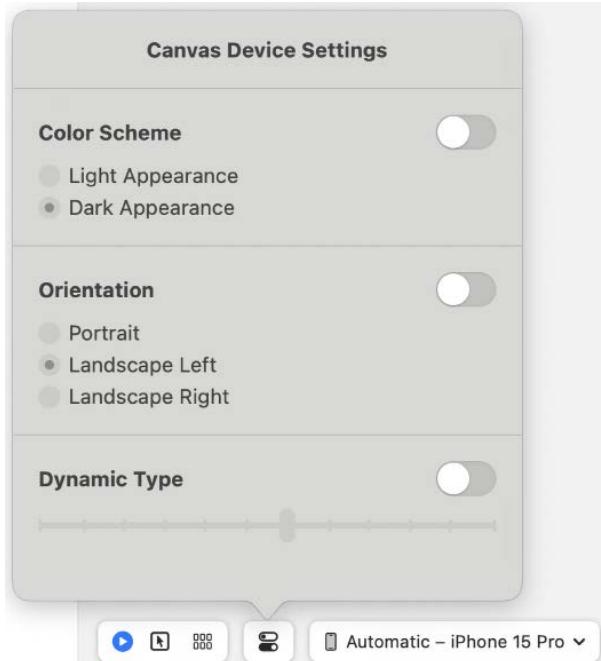


Figure 4.8: Device Settings button showing the pop-up menu

14. Play with these controls, and observe how the canvas changes and the simulated device shows a different UI. For example, if you switch the **Color Scheme** toggle on and switch the **Orientation** toggle on, the preview will show the app with a dark appearance (also known as dark mode) and in a **landscape-left** orientation. The resulting preview should look as follows:



Figure 4.9: Canvas preview in dark mode and landscape-left orientation

15. We have the option of previewing our views on a different type of device. We can accomplish this with the **Preview Device** menu:

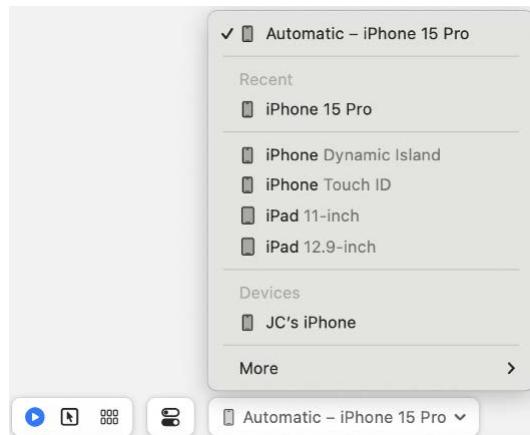


Figure 4.10: Preview Device menu options

16. Most of the time, you'll be working with the **Automatic** option, but you can choose a specific iOS simulator, a device family, or even a physical iPhone/iPad connected to your Mac. It is interesting to preview your view directly on your iPhone. Give it a try, and see how the preview on your iPhone updates in real time as you change the code.

In the following image, I have chosen an iPhone with Touch ID for the preview:

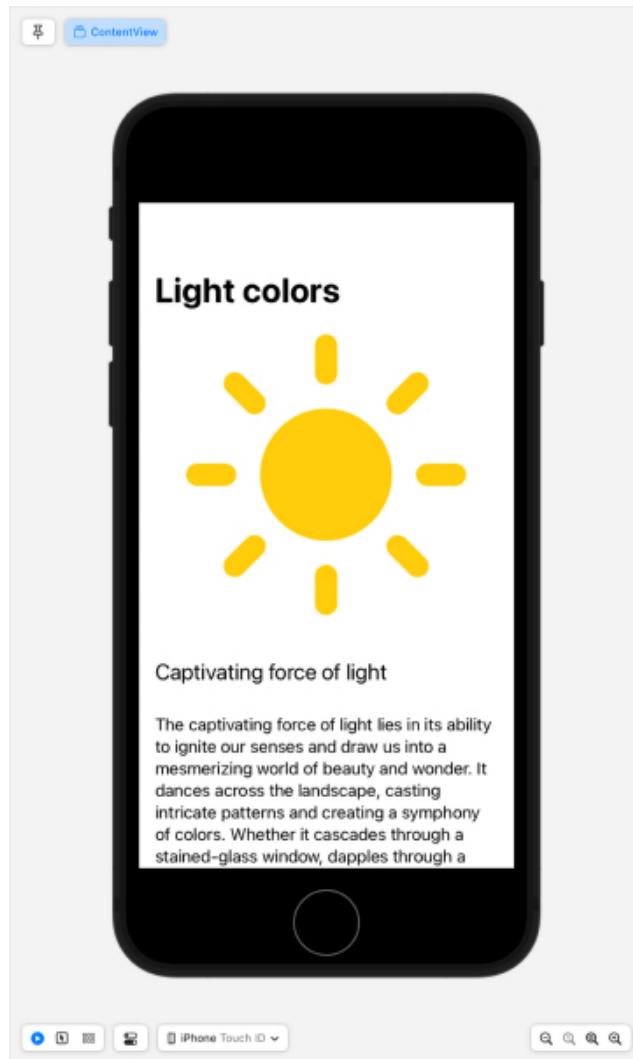


Figure 4.11: Preview on an iPhone with a Touch ID simulator

17. The zoom settings are self-explanatory, and I won't cover them. Spend some time getting used to them and you'll see they are very intuitive.
18. Finally, let's see how the Pin preview feature works and how the preview titles are useful. Switch back to Automatic Preview Device, make sure ContentView is selected and showing the preview, and click on the pin preview button in the top-left corner of the canvas. The pin icon should change from an outline icon to a solid icon like this:



Figure 4.12: Pin preview selected

19. Switch to the PostView file, and now the canvas should show two buttons to the right of the pin icon. By clicking on these buttons, you'll switch between the two different previews, the ContentView, which was pinned, and the PostView, which is the preview for the file we have selected. When we have more than one preview showing, it is useful to have titles on each preview.



Figure 4.13: Two previews side by side

How it works...

This recipe is all about using the canvas preview in Xcode to increase our productivity, as already explained in the previous section. However, a couple of code snippets are worth mentioning.

To make our UI react to color scheme changes, we added a `deviceColorScheme` variable, which is tied to the `colorScheme` environment value managed by SwiftUI. We can use a conditional statement to display a different view, depending on the color scheme of the device. In our case, we used two different `PostView` instances, created by passing different parameters to the initializer:

```
if deviceColorScheme == .light {  
    // View for light appearance color scheme  
    PostView(...)  
} else {  
    // View for dark appearance color scheme  
    PostView(...)  
}
```

Like the `colorScheme` environment variable, we have the `dynamicTypeSize` environment variable, which allows us to know which size was selected by the user, giving us the ability to modify our UI programmatically, as we did for the color scheme.

To detect the orientation of the device and use a conditional statement to display different views, we used a `GeometryReader` in `PostView`. The initializer of the `GeometryReader` uses a closure with a `GeometryProxy` as an input parameter. The `GeometryProxy` allows us to access the size and coordinate space of the `PostView` view. If the height of the view is greater than the width, we can infer that the view is showing in portrait orientation. Our `PostView` is basically a custom view with three embedded views, two `Text` instances, and one `Image` instance. We use a `VStack` to group the two `Text` instances. In portrait orientation, we use another `VStack` to arrange the image vertically on top of the `VStack` with the two `Text` instances. In landscape orientation, we use an `HStack` to group the `Image` view on the side of the `VStack` with the two `Text` instances:

```
if geometry.size.height > geometry.size.width {  
    // View for portrait orientation  
} else {  
    // View for landscape orientation  
}
```

See also

- A list of all possible environment values: <https://developer.apple.com/documentation/swiftui/environmentvalues>
- `GeometryReader` in detail: <https://developer.apple.com/documentation/swiftui/geometryreader>
- More dynamic type sizes: <https://developer.apple.com/documentation/swiftui/dynamictypesize>

Previewing a view in a NavigationStack

Some views are designed to be embedded in a `NavigationStack`; however, the `NavigationStack` is usually declared in the parent view and not our view. Following best coding practices, our view is a separate struct usually declared in a separate file. When we preview our view, we would like to see how it would look embedded in a `NavigationStack` instead of as an isolated view. 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 changes live without rebuilding the app.

In this recipe, we will create an app with a view that is part of the `NaigationStack`, previewing it in a `NavigationStack` instead of as an isolated view.

Getting ready

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

How to do it...

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

1. Replace the content of the `body` variable in `ContentView` with a `NavigationStack`, containing a `NavLink` that directs us to `SecondView`:

```
struct ContentView: View {  
    var body: some View {  
        NavigationStack {  
            NavLink(destination: SecondView(someText: "Text passed  
from ContentView")) {
```

```
        Text("Go to second view")
            .foregroundStyle(.white)
            .padding()
            .background(.black)
            .cornerRadius(25)
        }
        .navigationTitle("Previews")
        .navigationBarTitleDisplayMode(.inline)
    }
}
```

2. Pass a title to the preview macro:

```
#Preview("ContentView") {
    ContentView()
}
```

3. Create a new SwiftUI view called SecondView:

- Press **Command (⌘) + N**.
- Select **SwiftUI View**.
- Enter **SecondView** in the **Save As** field.
- Click **Create**.
- Replace the content of the **SecondView.swift** file with the following:

```
struct SecondView: View {
    var someText: String
    var body: some View {
        Text(someText)
            .navigationTitle("Second View")
            .navigationBarTitleDisplayMode(.inline)
    }
}

#Preview("SecondView") {
    SecondView(someText: "This is the second view")
}
```

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



Figure 4.14: SecondView with no navigation bar

4. Open the `ContentView.swift` file.
5. Choose live mode by clicking the **Live** button on the canvas.
6. Click on the **Go to second view** button to navigate to `SecondView`. The `SecondView` live preview should look as follows:

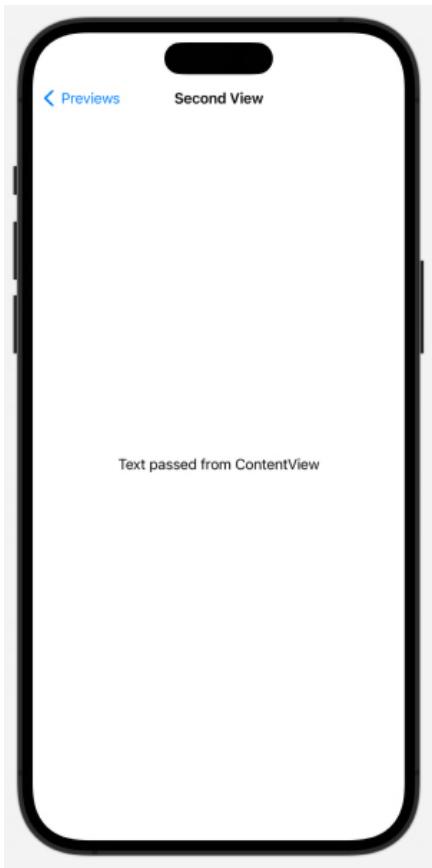


Figure 4.15: SecondView UI within a NavigationStack

7. Now, go back to our `SecondView.swift` file, and embed the `SecondView` declaration in a `NavigationStack`. After the live preview updates, the canvas should show the navigation title:

```
#Preview("SecondView") {
    NavigationStack {
        SecondView(someText: "This is the second view")
    }
}
```

Now, after this change, the preview should look as follows:

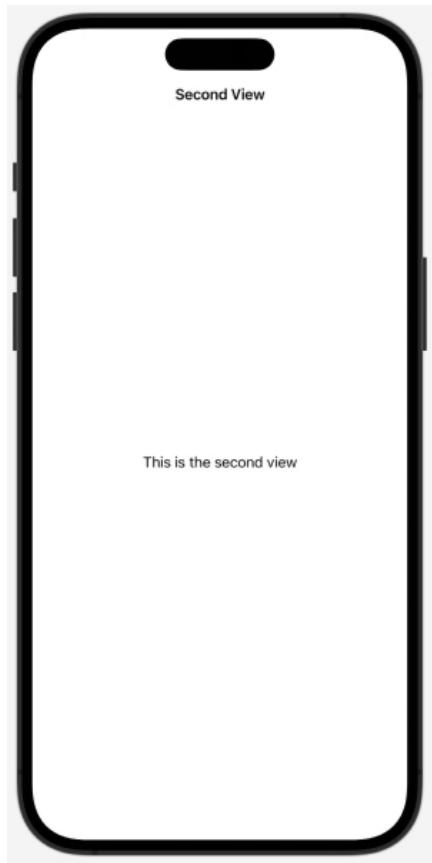


Figure 4.16: SecondView UI within a NavigationStack

How it works...

Views containing a `.navigationTitle()` modifier but no `NavigationStack` are meant to be displayed as part of a navigation stack. Enclosing the preview of such views in a `NavigationStack` component provides a quick way of accurately previewing the intended UI.

Using a `NavigationStack` in the `#Preview` macro allows us to visualize updates to views that are part of a navigation stack, without first running the app. This approach can be generalized to other types of container views. For example, if we have an app that displays a list of custom views representing elements in a collection, we might want to embed the preview for the custom view in a list to reflect our use case more accurately.

Previewing a view with different traits

SwiftUI's preview macro allows us to preview views with different traits. In this recipe, we will create a simple app that displays an image and some text, and we will preview it with different traits. This will allow us to have different variants of the view in the same canvas, switching among them by clicking on the preview buttons generated by Xcode.

Getting ready

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

How to do it...

We will modify the `ContentView` struct generated by Xcode and then add multiple preview macros to show the content with different traits. The steps are as follows:

1. Open the `ContentView.swift` file and modify the body of the `ContentView` struct:

```
struct ContentView: View {
    var body: some View {
        ZStack {
            Color.teal
                .frame(width:200, height: 150)
            VStack {
                Image(systemName: "globe")
                    .imageScale(.large)
                Text("Hello, world!")
                    .font(.title2)
            }
            .foregroundStyle(.primary)
            .padding(.all, 20)
            .background()
            .border(.tertiary, width: 10)
        }
    }
}
```

2. Let's add a title to the preview macro:

```
#Preview("Portrait") {
    ContentView()
}
```

3. Now, let's add three more preview macros. Below the `Portrait` preview macro, add the following:

```
#Preview("Landscape", traits: .landscapeLeft) {
    ContentView()
}

#Preview("Upside down", traits: .portraitUpsideDown) {
    ContentView()
}

#Preview("Size that fits", traits: .sizeThatFitsLayout) {
    ContentView()
}

#Preview("Fixed size", traits: .fixedLayout(width: 500, height: 200)) {
    ContentView()
}
```

4. Select the **iPhone 14 Pro** iOS simulator from the top toolbar, and make sure the canvas is shown and that the preview device shows **Automatic**. Switch to **selectable** preview mode. If everything went well, there should be five different previews. Spend some time playing with the different previews to understand how they work. Make sure to choose the selectable preview mode; otherwise, the last two previews that have custom sizes won't show as intended.



Figure 4.17: Canvas with several previews

How it works...

The preview macro accepts an optional `String`, used as a title for the preview, an optional `traits` parameter, and a content closure with the view we want to preview. The `traits` parameter accepts several values to choose the orientation of the device, or even a custom frame to preview our content. The `.sizeThatFitsLayout` is useful to see the minimum frame that contains our view.

Previewing a view on different devices

Xcode 15 allows us to preview our designs on multiple screen sizes and device types simultaneously, using the new device selector. In this recipe, we will create a simple app that displays an image and some text and preview it in multiple devices, without using any additional code.

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. Download the `friendship.jpg` image from this chapter's resources file at <https://github.com/PacktPublishing/SwiftUI-Cookbook-3rd-Edition/blob/main/Resources/Chapter04/recipe3/friendship.jpg>.
2. Drag and drop the downloaded image file into the `Assets` folder in Xcode.
3. Open the `ContentView.swift` file, and replace its content with the following code to display an image and some text:

```
import SwiftUI

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()
        }
    }
}

#Preview("Friendship") {
    ContentView()
}
```

Using the **Preview Device Selector** in the canvas preview area, we can choose among different device types. We even have the option of choosing generic devices like the iPhone Dynamic Island:

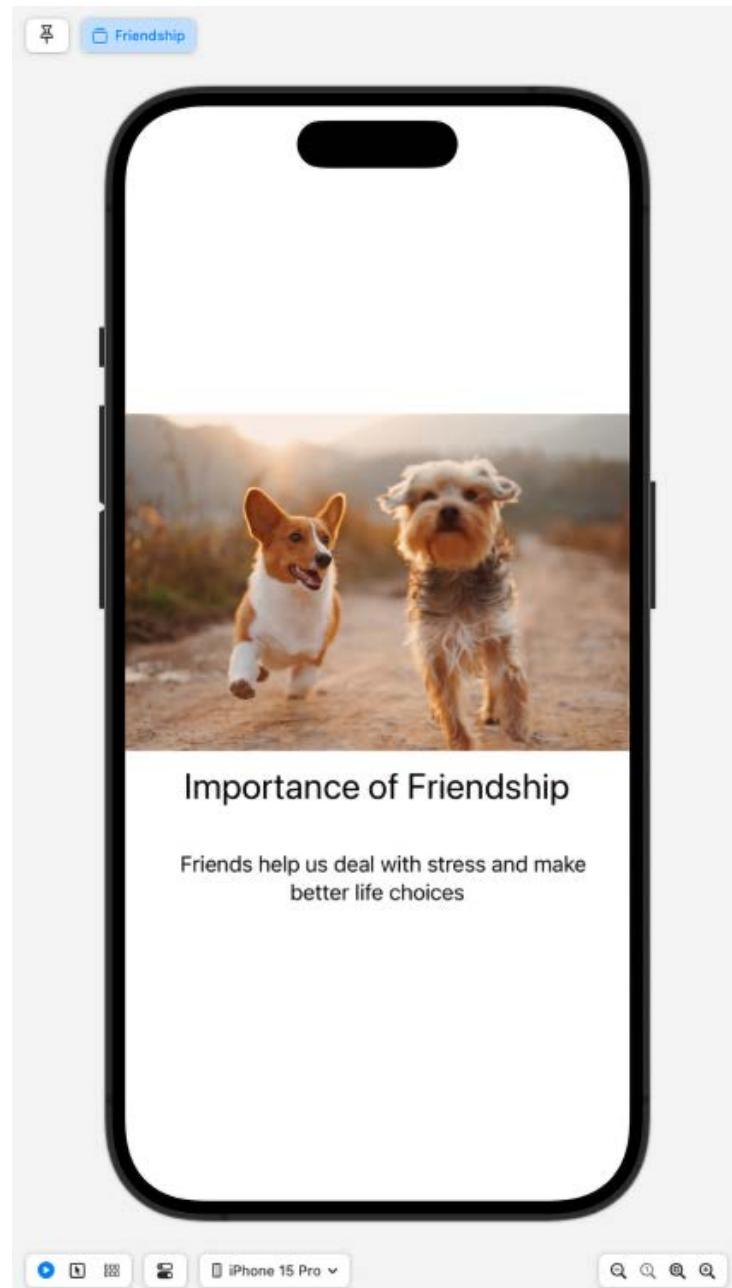


Figure 4.18: Canvas preview with the iPhone 15 Pro selected

How it works...

Xcode 15 takes preview functionality to a different level. Before, we had to use different modifiers to preview on different devices, but now, we can do it with just a click on the Preview Device Selector, as shown in the following screenshot:

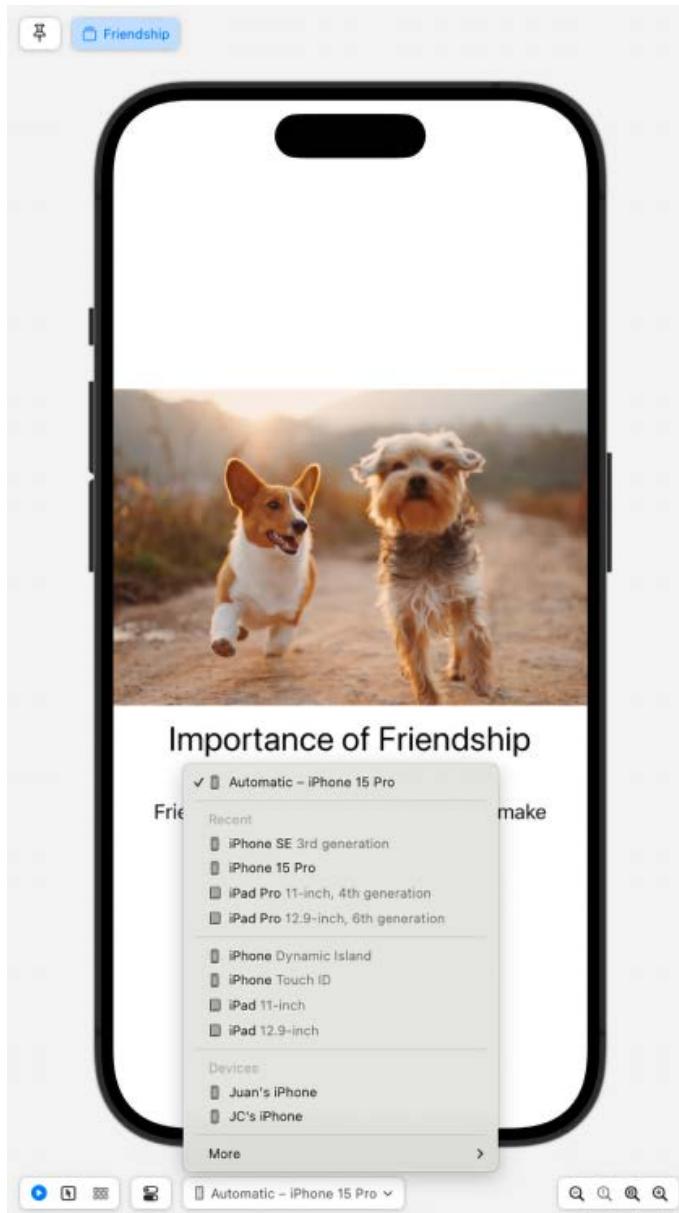


Figure 4.19: List of available devices in the Device Preview selector

See also

- Previews in Xcode: <https://developer.apple.com/documentation/swiftui/previews-in-xcode>
- Preview macros on Xcode 15: <https://developer.apple.com/videos/play/wwdc2023/10252/>

Using previews in UIKit

If you love the ease with which you can preview UI changes, but you are still working on UIKit projects, rest easy, as you can also use the canvas preview feature while building UIKit apps. With Xcode 15, the preview macros work with UIKit out of the box. Under the hood, Xcode adds the boilerplate code to wrap UIKit views and view controllers in SwiftUI views, so they can be previewed in the canvas.

In this recipe, we will learn how to preview a `UIViewController` and a `UIView` on the Xcode canvas.

Getting ready

Since this book is about SwiftUI and we are going to add previews to a UIKit app, we have already provided the `UIKit` app as a starter project, so you can focus on the preview functionality. It is very important that you clone or download the code for this chapter from GitHub, as explained in the *Technical requirements* section.

How to do it...

We will use the `UIKit` app provided and then add a couple of preview macros. The steps are as follows:

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

- Click on the **UIKitPreviews** project file in the Xcode navigation pane, select **Build Settings**, scroll down to the **Deployment** section, and make sure **iOS Deployment Target** has a value of **iOS 17 or higher**:

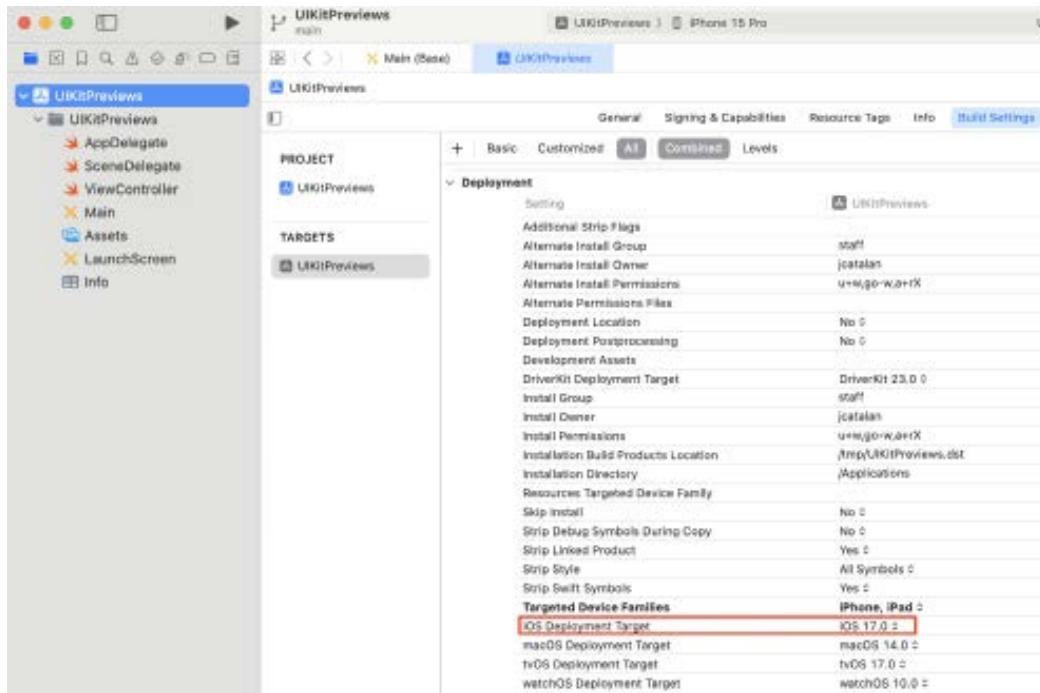


Figure 4.20: Xcode build settings with the deployment targets highlighted

- Run the app on your simulator of choice to get familiarized with it. You'll see it is a simple app with a `main` view and a button, which, when tapped, pushes another view on a navigation stack. Clicking on the back button on the navigation bar of the detail view takes us back to the `main` view.
- Open the `Main` storyboard to see the UI for the whole app.
- Open the `ViewController.swift` file. You'll see that there is not much code there because all the UI elements are defined in the storyboard.
- Now, let's take steps that will allow us to preview the app using the preview macros and the Xcode canvas. Let's add this code at the end of the `ViewController` file:

```
#Preview {
    let sb = UIStoryboard(name: "Main", bundle: nil)
    let vc = sb.instantiateInitialViewController()!
    return vc
}
```

- You'll see that as soon as you add the `#Preview` macro, the Xcode canvas appears. If everything went well, the canvas preview should look as follows:

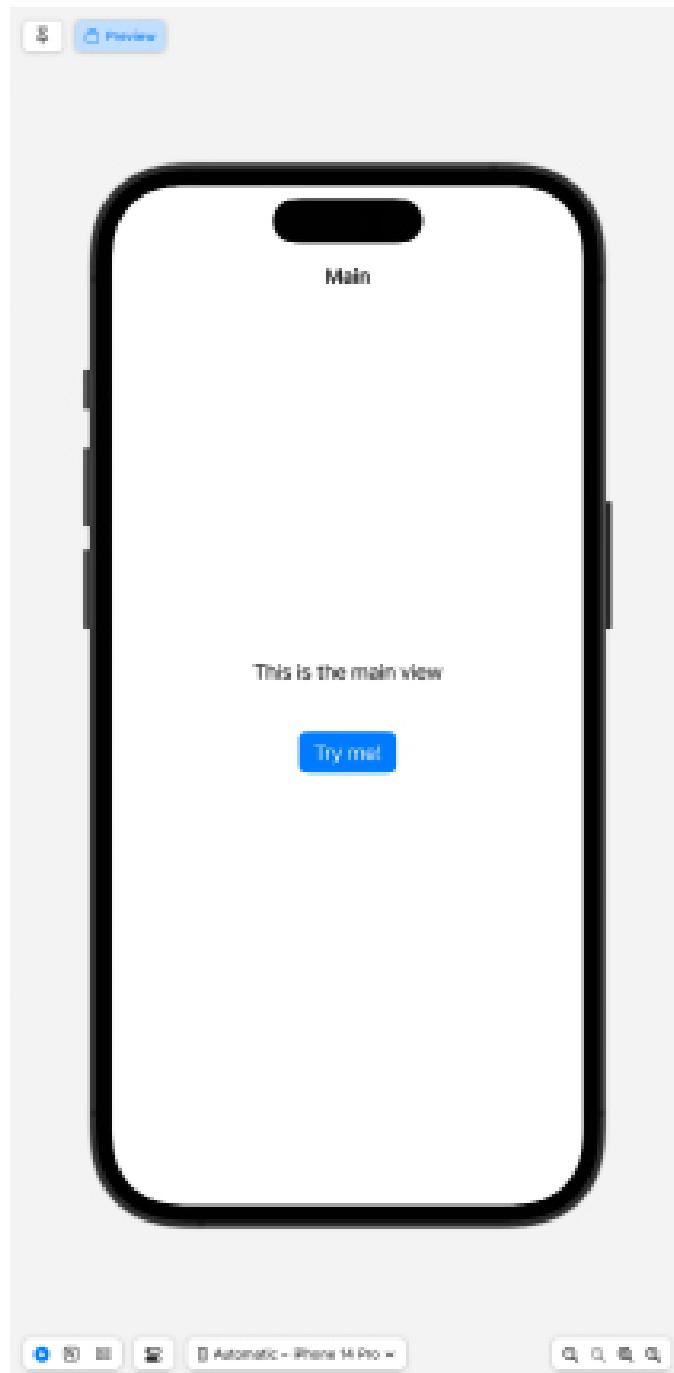


Figure 4.21: The canvas showing the live preview of a UIKit app

Interact with the live preview, and you'll see that the app behaves as expected. You can show the detail view by tapping on the button and going back to the main view.

8. Let's add this other code to the end of the `ViewController` file, showing you how to preview a `UILabel`:

```
#Preview("UILabel") {  
    let label = UILabel()  
    label.text = "Previewing a UILabel"  
    label.font = UIFont.preferredFont(forTextStyle: .title3)  
    return label  
}
```

9. Go to the canvas preview and click on the `UILabel` button at the top. The result should look as follows:



Figure 4.22: The canvas showing the live preview of a `UILabel`

To see the power of the live previews for `UIKit`, change the title of the label to a different text or change the font, and then you'll see how the preview updates with the changes. The preview code can be found in the `ViewController.swift` file located in the `Complete` folder of this recipe.

How it works...

The preview macros in Xcode 15 are straightforward to use. Just instantiate a `UIViewController` or a `UIView`, configure it, and return it in the preview closure. Xcode will preview what you return in the closure.

In our first preview macro, we instantiate the initial view controller of the app using the `Main` storyboard, and we return the view controller in the last statement of the closure. Using the initial view controller in a `UIKit` app allows us to use the live preview to interact with the app on the canvas.

Our second preview macro instantiates a `UILabel`, configures some properties on the label, and then returns the label at the end of the closure. The preview shows the label as configured.

Using mock data for previews

So far, we've built apps using our own data. However, when dealing with large projects, data is usually obtained by making API calls. However, that can be time-consuming and quickly become a bottleneck for our previews. The better option is to make some mock data available for previews only. We can tell Xcode not to bundle the data and resources used for previews with our app when we submit the app to the App Store.

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 a lightweight format used to store and transfer 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-3rd-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:

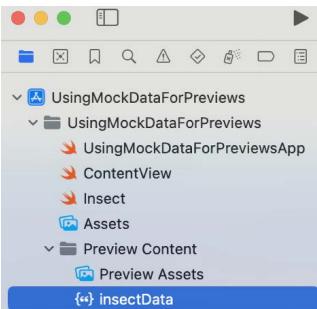


Figure 4.23: Mock data in the Preview Content folder

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

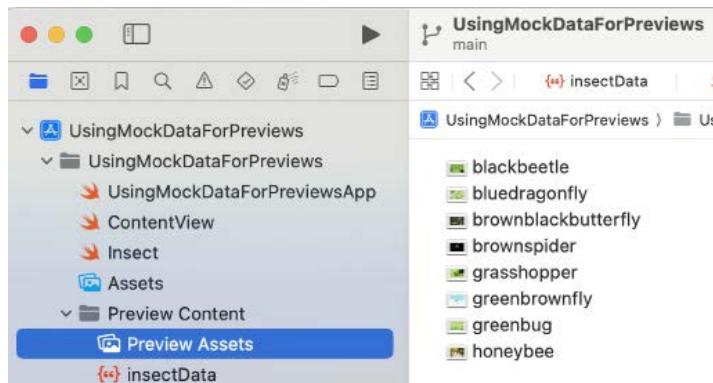


Figure 4.24: Adding the insect images to the Preview Content folder

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

6. Open the `ContentView.swift` file. Replace the content with the following code to display a list of insects:

```
struct ContentView: View {
    var insects: [Insect] = []
    var body: some View {
        NavigationStack {
            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 the preview macro to read and display content from our JSON file with the mock data:

```
#Preview {
    let testInsect = Insect(id: 1, imageName: "grasshopper", name: "grass", habitat: "rocks", description: "none")
    let 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]
}()
return ContentView(insects: testInsects)
}
```

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

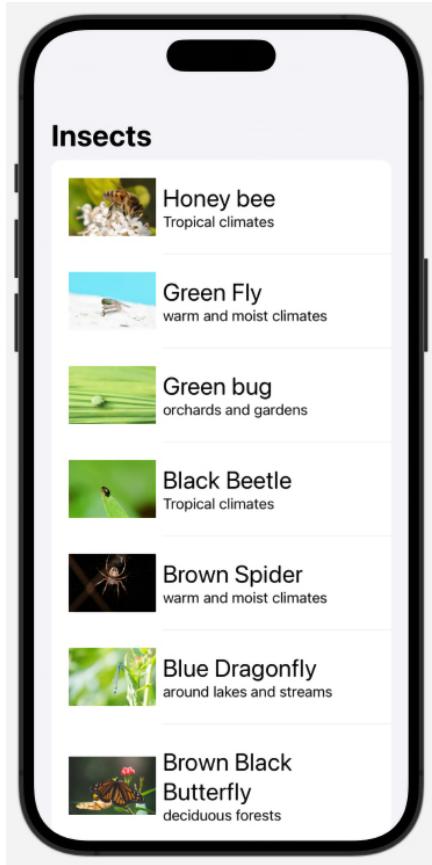


Figure 4.25: Using mock data in the project preview

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

- Finally, let's make sure Xcode does not bundle our preview images and JSON file with the mock data for release builds. Select the project file, go to the **Build Settings** tab, and type **Development Assets** in the filter box at the top right. You'll see a field named **Development Assets** with a value of **UsingMockDataForPreviews/Preview Content**, like in the figure below:

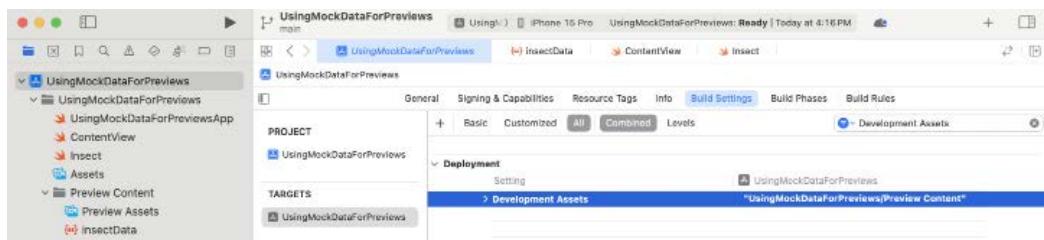


Figure 4.26: Excluding development assets from builds

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 the preview macro, we perform several steps to fetch and convert our data into an array of `Insect` instances. We first fetch the data from the file:

```
guard let url = Bundle.main.url(forResource: "insectData",
withExtension: "json"),
      let data = try? Data(contentsOf: url)
else {
    return []
}
```

Then, using a `JSONDecoder` instance, 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.

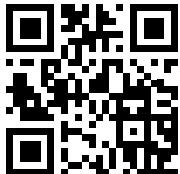
Try breaking down the code into `InsectView` and `InsectListView` SwiftUI views. The `InsectView` should focus on displaying data regarding one insect. The `InsectListView` takes an array of insects and displays them in a list, using our `InsectView` SwiftUI view.

Finally, our `ContentView.swift` file should do nothing else but call our `InsectListView`, while passing an array of `Insect` structs.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

<https://packt.link/swiftUI>



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 negative UX and low retention rates. We used forms and containers in the earlier chapters, but here we will learn about additional ways of using forms.

We will also learn more about how to display information in two-dimensional grids, using new components like **Grid** views, and how to display editable information using **Table** views.

In this chapter, we will focus on grouping views using **Form**, **Table**, and **Grid** container 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
- Creating multi-column lists with Table
- Using Grid, a powerful two-dimensional layout

Technical requirements

The code in this chapter is based on Xcode 15.0 and iOS 17.0. You can download and install the latest version of Xcode from the App Store. You'll also need to be running macOS Ventura 13.5 or newer.

Simply search for Xcode in the App Store and select and download the latest version. Launch Xcode and follow any additional installation instructions that your system may prompt you with. Once Xcode has fully launched, you're ready to go.

All the code examples for this chapter can be found on GitHub at <https://github.com/PacktPublishing/SwiftUI-Cookbook-3rd-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, named `Previous 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 **Create**.
2. In `SignUpView.swift`, above the `SignUpView` struct declaration, define the new struct `Address` used to hold some data that will be used in the form:

```
struct Address {  
    var street: String = ""  
    var city: String = ""  
    var postalCode: String = ""  
}
```

3. In `SignUpView`, declare and initialize the `@State` variables that will be used in the form:

```
struct SignUpView: View {  
    @State private var firstName = ""  
    @State private var lastName = ""  
    @State private var address = Address()
```

```
@State private var address2 = Address()  
@State private var lessThanTwo = false  
@State private var username = ""  
@State private var password = ""  
// More code here
```

4. Replace the content of the `body` variable with a `NavigationStack`, a `Form` view, and a `.navigationTitle` modifier:

```
var body: some View {  
    NavigationStack {  
        Form{  
            // More code here  
        }  
        .navigationTitle("Sign Up")  
    }  
}
```

5. Inside the `Form`, add a `Section` with two `TextField` views. The first will bind the `text` argument to the `firstName` variable with the label of "First Name", and the second `text` argument will be bound to the `lastName` variable with the label of "Last Name". This will take care of the storage of the user's first and last names:

```
Section("Names") {  
    TextField("First Name", text: $firstName)  
    TextField("Last Name", text: $lastName)  
}
```

6. Just below the `Section` defined in the previous step, add another `Section` with fields for the user's current address:

```
Section("Current Address") {  
    TextField("Street Address", text: $address.street)  
    TextField("City", text: $address.city)  
    TextField("Postal Code", text: $address.postalCode)  
}
```

7. Add a `Toggle` view to the preceding section to inquire whether the user has been at their current address for at least 2 years. Place the `Toggle` view below the `TextField` where we request the user's postal code:

```
Toggle(isOn: $lessThanTwo) {  
    Text("Have you lived here for 2+ years")  
}
```

8. 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: $address2.street)  
        TextField("City", text: $address2.city)  
        TextField("Postal Code", text: $address2.postalCode)  
    }  
}
```

9. Next, add a `Section` to the form with a `TextField` view for the username and a `SecureField` view for the password:

```
Section {  
    TextField("Create Username", text: $username)  
    SecureField("Create Password", text: $password)  
}
```

10. 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 like this:

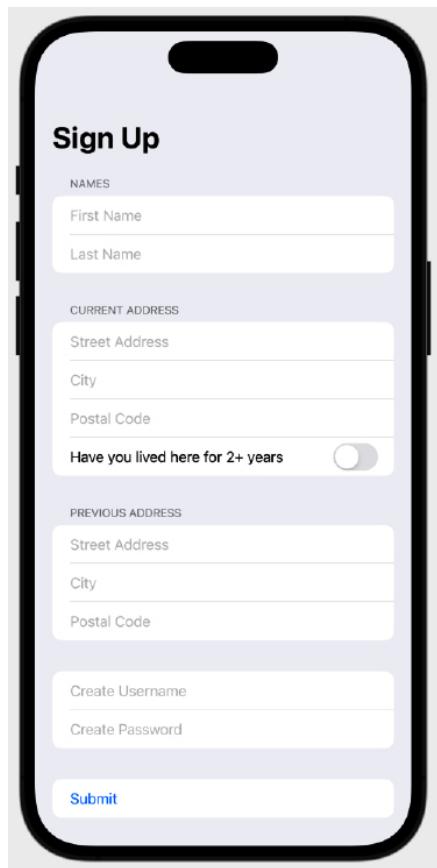


Figure 5.1: Signup form

How it works...

A basic Form view with a TextField view is created by wrapping the TextField view with a Form view, then wrapping the Form view with a NavigationStack view, as follows:

```
struct signUp: View {  
    @State var firstName: String = ""  
  
    var body: some View {  
        NavigationStack {  
            Form {  
                TextField("firstName", text: $username)  
            }  
            .navigationTitle("Settings")  
        }  
    }  
}
```

We create a `@State` variable to hold the user's input and a `TextField` to request the input. We can add more variables and more fields, but the overall structure of the code is like the code of the simple code block above.

Furthermore, `Section` views are used to provide a visible separation between related fields. Additionally, we can add headers to `Section` views, such as the `NAMES` section, or provide no header, such as the section with fields for `username` and `password`.

Finally, 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 `false`.

There's more...

If you use the live preview on the canvas with `ContentView` and click the `Submit` button, you may notice that no `print` statement gets displayed. If that's the case, it means that the `Debug Area` of Xcode is hidden. To see `print` statements, you need to make sure the `Debug Area` is visible (`View | Debug Area | Show Debug Area`), as shown in the following figure:

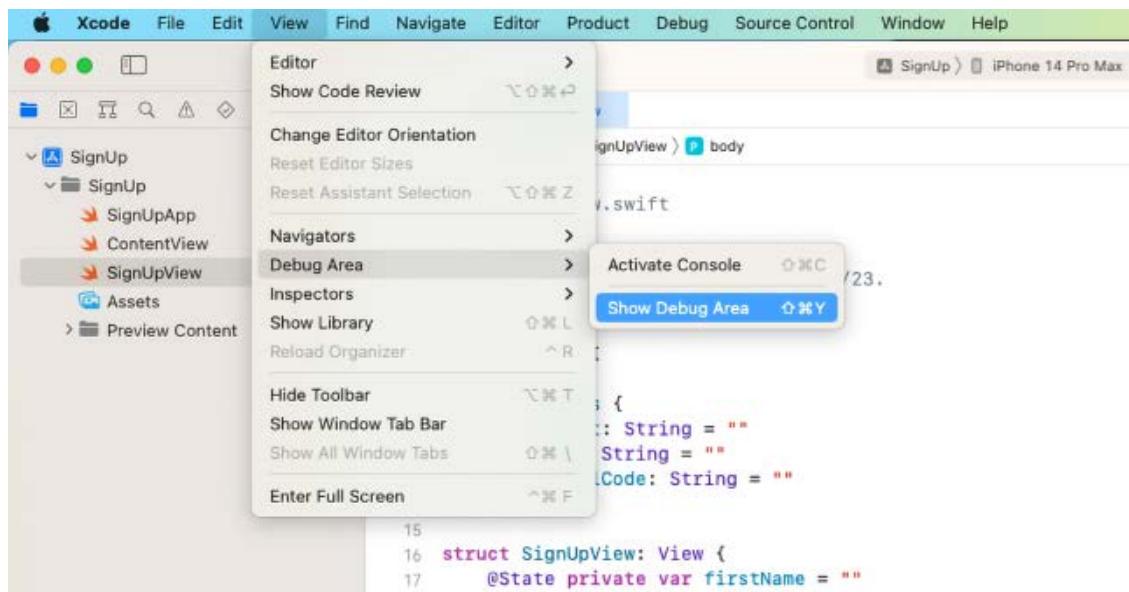


Figure 5.2: Show Debug Area option

If the **Debug Area** is visible, you will see print statements being displayed in the **Debug Area** each time the **Submit** button is clicked. In the **Debug Area**, make sure to select the **Previews** or **Executable** tab, depending on whether you are using the live preview or running the app in a simulator or device.

See also

Apple's 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 until 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 **Create**.
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, replace the contents of the `body` variable with 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)  
            .foregroundStyle(.blue)  
            .font(.largeTitle)  
            .padding(.bottom, 30)  
    }  
    .padding()  
    // ...  
}
```

4. Add a placeholder below the Text view for the user's profile picture:

```
Image(systemName: "person.circle")
    .font(.system(size: 150))
    .foregroundStyle(.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 {
    print("Login tapped")
} label: {
    Text("Login")
}
.padding()
.background((username.isEmpty || password.isEmpty) ? .gray : .blue)
.foregroundStyle(.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:

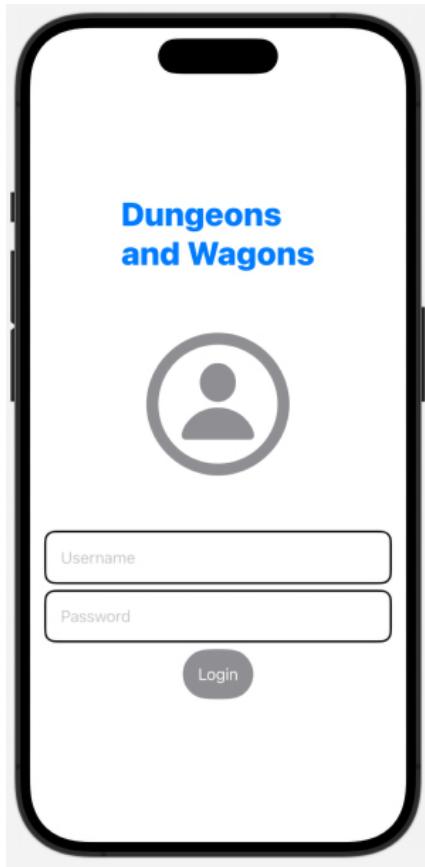


Figure 5.3: `FormFieldDisable` preview

Use the live preview in the canvas and notice the `Login` button only turns blue when both the `Username` and `Password` fields are filled in. 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. This modifier takes a boolean parameter and disables the button when the boolean parameter's value is set to `true`.

In this recipe, we checked whether our `@State` variables for the username and password fields were not empty. The `Login` button stayed disabled as long as one or both fields were empty.

We also applied 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. Before Swift 5.9, `Group` was also used to overcome a limitation on the `ViewBuilder` in `HStack`, `VStack`, and `ZStack`, which could only hold up to 10 embedded views.

This limitation no longer exists thanks to the introduction of Parameter Packs in Swift 5.9.

Filling out forms easily using Focus and Submit

Filling out forms can be tedious if the user must 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` declaration, 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. Replace the content of the body variable with 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)
    }
    .padding()
}
```

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 a simulator. Make sure to disable the hardware keyboard from the I/O menu of the simulator or the keyboard will not show. The result should be like this:



Figure 5.4: FocusAndSubmit iOS simulator screenshot

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 code you'd like on the `onSubmit` closure.

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 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 the **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. In our recipe, we used an `enum` instead of a `boolean` for our `@FocusState` variable. In that case, to remove the focus, we should set the variable to `nil`.

See also

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

Creating multi-column lists with Table

macOS has supported multi-column tables for a long time. SwiftUI support for multi-column tables was added in macOS 12 with the `Table` struct, which, one year later, was added to iOS with the release of iOS 16.

Table is a container that presents rows of data arranged in columns and provides the ability to sort the data by column and to select one or multiple rows of data. Table only supports the multi-column layout on the iPad, and it falls back to a one-column list on the iPhone, displaying, by default, the first column. In a similar way, column sorting and multiple-row selection are only available on the iPad.

In this recipe, we will create an app that displays the top 20 US cities by population on the iPad screen using Table. We will incorporate sorting and multiple-row selection, and finally, create a custom layout to display the information on the iPhone.

Getting ready

Create a new SwiftUI project named `MultiColumnTable`.

Check the project's **Build Settings** and make sure the iOS target version is set to **17.0** or higher.

How to do it...

We declare a custom struct, `City`, with a few properties to represent relevant attributes of a city. Then we populate an array of `City` elements with the top 20 US cities by population:

1. Let's define a new struct to model our data to represent a city. Create a new Swift file by selecting **File | New | File | Swift file** from the Xcode menu or by using the *Command* (**⌘**) + *N* key combination. Name the file `City`.
2. In the file, create a `City` struct with the following properties:

```
struct City: Identifiable {
    var name: String
    var state: String
    var population: Int
    var area: Measurement<UnitArea>
}
```

3. Since instances of the `City` struct will be displayed in a `Table`, we need `City` to conform to the `Identifiable` protocol. This protocol just has one requirement, which is that each instance needs to implement an `id` property that will uniquely identify the instance among a collection of instances. For a city in the US, a combination of the city name and the state will give us this uniqueness. Right after the `area` property, let's add a computed variable to fulfill the protocol conformance and remove Xcode's error:

```
var id: String { "\((name), \((state))" }
```

4. Now we need a collection of cities to populate our `Table`. Let's create a static variable of type `[City]` and initialize it with a collection of cities, representing the top 20 cities in the US by population. We use an extension to define this collection:

```
extension City {
    static var top20USCities: [City] = [
```

```
City(name: "Austin", state: "Texas", population: 964177, area:  
Measurement(value: 828.5, unit: UnitArea.squareKilometers)),  
    City(name: "Charlotte", state: "North Carolina",  
population: 879709, area: Measurement(value: 798.5, unit: UnitArea.  
squareKilometers)),  
    City(name: "Chicago", state: "Illinois", population: 2696555,  
area: Measurement(value: 589.7, unit: UnitArea.squareKilometers)),  
    City(name: "Columbus", state: "Ohio", population: 906528, area:  
Measurement(value: 569.8, unit: UnitArea.squareKilometers)),  
    City(name: "Dallas", state: "Texas", population: 1288457, area:  
Measurement(value: 879.6, unit: UnitArea.squareKilometers)),  
    City(name: "Denver", state: "Colorado", population: 711463, area:  
Measurement(value: 396.5, unit: UnitArea.squareKilometers)),  
    City(name: "Fort Worth", state: "Texas", population: 935508,  
area: Measurement(value: 899.5, unit: UnitArea.squareKilometers)),  
    City(name: "Houston", state: "Texas", population: 2288250, area:  
Measurement(value: 1658.6, unit: UnitArea.squareKilometers)),  
    City(name: "Indianapolis", state: "Indiana", population: 882039,  
area: Measurement(value: 936.5, unit: UnitArea.squareKilometers)),  
    City(name: "Jacksonville", state: "Florida", population: 954614,  
area: Measurement(value: 1935.5, unit: UnitArea.squareKilometers)),  
    City(name: "Los Angeles", state: "California", population:  
3849297, area: Measurement(value: 1216, unit: UnitArea.  
squareKilometers)),  
    City(name: "New York", state: "New York", population: 8467513,  
area: Measurement(value: 778.3, unit: UnitArea.squareKilometers)),  
    City(name: "Oklahoma City", state: "Oklahoma", population:  
687725, area: Measurement(value: 1570.1, unit: UnitArea.  
squareKilometers)),  
    City(name: "Philadelphia", state: "Pennsylvania",  
population: 1576251, area: Measurement(value: 348.1, unit: UnitArea.  
squareKilometers)),  
    City(name: "Phoenix", state: "Arizona", population: 1624569,  
area: Measurement(value: 1341.6, unit: UnitArea.squareKilometers)),  
    City(name: "San Antonio", state: "Texas", population: 1451853,  
area: Measurement(value: 1291.9, unit: UnitArea.squareKilometers)),  
    City(name: "San Diego", state: "California", population: 1381611,  
area: Measurement(value: 844.1, unit: UnitArea.squareKilometers)),  
    City(name: "San Francisco", state: "California",  
population: 815201, area: Measurement(value: 121.5, unit: UnitArea.  
squareKilometers)),  
    City(name: "San Jose", state: "California", population: 983489,  
area: Measurement(value: 461.8, unit: UnitArea.squareKilometers)),
```

```
        City(name: "Seattle", state: "Washington", population: 733919,  
area: Measurement(value: 217, unit: UnitArea.squareKilometers))  
    ]  
}
```

- Now that we have all the data we need, let's create our table of cities. We want to sort the cities by population in descending order so that our table displays the most populated city first and the least populated last. To sort the data, we use the `sorted(using:)` function of `Sequence`, which takes an instance of a `SortComparator` to specify the sort order. Open the `ContentView.swift` file and declare two variables right after the struct declaration. One variable defines a `KeyPathComparator`, a struct that conforms to the `SortComparator` protocol, and the other variable is a `@State` variable, which holds the data to be displayed:

```
static let populationComparator = KeyPathComparator(\City.population,  
order: .reverse)  
@State private var cities: [City] = City.top20USCities.sorted(using:  
populationComparator)
```

- A multi-column table is meant to be used in an iPad app. From the Xcode toolbar in *Figure 5.5*, select the iPad simulator of your choice:



Figure 5.5: Xcode toolbar with the iPad mini simulator selected

- Now we are finally ready to create our Table. We wrap the Table in a `NavigationStack` and provide a title with the `.navigationTitle` modifier. Replace the content of the `body` variable with the following:

```
NavigationStack {  
    Table(cities) {  
        TableColumn("Name", value: \.name)  
        TableColumn("State", value: \.state)  
        TableColumn("Population") { city in  
            Text("\(city.population)")  
        }  
        TableColumn("Area") { city in  
            Text("\(city.area.value)")  
        }  
    }  
    .navigationTitle("Top 20 US cities")  
}
```

If everything went well, the preview (in landscape left orientation) should look like this:

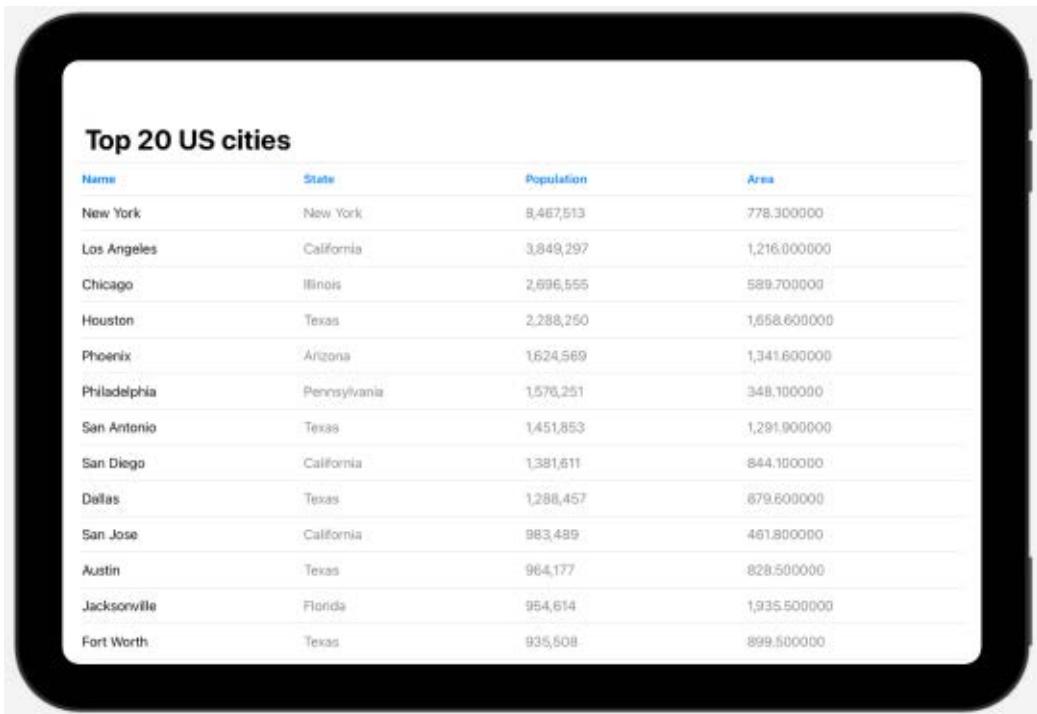


Figure 5.6: Preview of our table

The table of cities looks good except for the data formatting and presentation style of the **Population** and **Area** columns. We would like an appropriate numeric format and to align the values on the right since they represent numeric values. For the **Area** column, we would like to display the unit used to measure the area of the city with one decimal digit.

- Following best coding practices, we will separate the data formatting from the presentation style. The data formatting will be in our model code and the presentation in our view code. Let's switch to our **City** struct and add two computed properties that will transform the numeric values to nicely formatted string representations. Declare the two properties after the **id** property in the **City** struct:

```
var formattedArea: String {
    let formatter = MeasurementFormatter()
    formatter.numberFormatter.minimumFractionDigits = 1
    return formatter.string(from: area)
}

var formattedPopulation: String {
    let formatter = NumberFormatter()
    formatter.numberStyle = .decimal
    formatter.usesGroupingSeparator = true
    formatter.maximumFractionDigits = 0
}
```

```
        return formatter.string(from: NSNumber(integerLiteral: population))!
    }
```

9. Let's take care of the presentation style and use the new formatted data. Switch to the `ContentView` struct and replace the `TableColumn` declarations for `Population` and `Area` with the following:

```
 TableColumn("Population") { city in
    Text(city.formattedPopulation)
        .font(.body.monospacedDigit())
        .frame(minWidth: 100, alignment: .trailing)
}
.width(max: 150)
TableColumn("Area") { city in
    Text(city.formattedArea)
        .font(.body.monospacedDigit())
        .frame(minWidth: 100, alignment: .trailing)
}
.width(max: 150)
```

The preview should look like this:

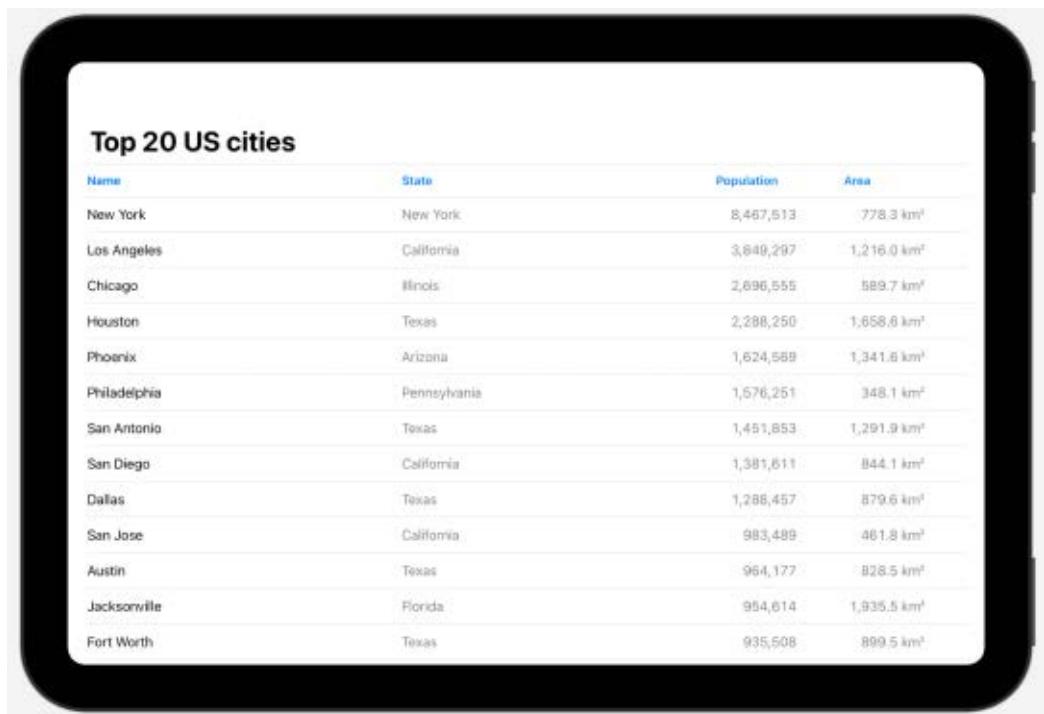


Figure 5.7: Preview of our Table with improved formatting and presentation styles

- Now we will implement sorting for each column. The `Table` struct makes this process very easy as it only needs binding to an array of `SortComparator` instances. Let's define a `KeyPathComparator` for each column of the table and declare a `@State` variable that will hold the array of the four comparators. Since the comparator for the `Population` column was already defined, we just need to define three more comparators, one for each of the remaining columns of the table:

```
@State private var sortOrder = [populationComparator, KeyPathComparator(\City.state), KeyPathComparator(\City.name), KeyPathComparator(\City.area)]
```

- We need to make three more modifications to the `Table` instance to implement the column-sorting functionality. First, we need to modify the declaration of the `table` to use the binding to the `sortOrder` variable:

```
Table(cities, sortOrder: $sortOrder)
```

- Then we need to modify the declaration of the `Population` and `Area` columns. Replace these corresponding `TableColumn` declarations with the following:

```
 TableColumn("Population", value: \.population) { ... }  
 TableColumn("Area", value: \.area) { ... }
```

- Finally, make the `Table` react to changes in the sort order so it will dynamically change the order of the elements. Add the `.onChange(of:initial: _ :)` modifier to the table, right after the `.navigationTitle` modifier:

```
.onChange(of: sortOrder) { _, newOrder in  
    cities.sort(using: newOrder)  
}
```

If all the steps were completed correctly, the preview should look as follows:

Name	State	Population	Area
New York	New York	8,467,513	778.3 km ²
Los Angeles	California	3,849,297	1,216.0 km ²
Chicago	Illinois	2,696,555	589.7 km ²
Houston	Texas	2,288,250	1,658.6 km ²
Phoenix	Arizona	1,624,569	1,341.6 km ²
Philadelphia	Pennsylvania	1,576,251	348.1 km ²
San Antonio	Texas	1,451,853	1,291.9 km ²
San Diego	California	1,381,611	844.1 km ²
Dallas	Texas	1,288,457	879.6 km ²
San Jose	California	983,489	461.8 km ²
Austin	Texas	964,177	828.5 km ²
Jacksonville	Florida	954,614	1,935.5 km ²
Fort Worth	Texas	935,508	899.5 km ²

Figure 5.8: Preview of our Table with column sorting by population in descending order

The Table reflects the sorted state through its column headers. Click on each column header and observe how the order of the rows changes. Additionally, notice how there is an arrow next to the column name. The arrow points up or down, reflecting ascending or descending sort order.

14. Let's enhance our table and implement a multiple-row selection. This is very easy to implement because Table supports this functionality by taking a binding to a selection variable. The variable is a Set of City.ID elements. Recall that ID is the associated type of the Identifiable protocol, which City conforms to. Declare a @State variable to hold the Set of ID instances and initialize it to an empty set:

```
@State private var selection = Set<City.ID>()
```

15. A couple of modifications are needed for the table to implement the multi-row selection. Modify the declaration of `Table` with the following:

```
Table(cities, selection: $selection, sortOrder: $sortOrder) {...}
```

16. And then add the `.onChange(of:initial:_:)` modifier at the end of the table, right after the `..onChange(of:initial:_:)` modifier used for the sort order:

```
.onChange(of: selection) {
    print("Selected = { \(Array(selection).sorted().joined(separator: " | "
)) }")
}
```

17. Let's run the app in the iPad simulator this time and click on a row. The row is highlighted, and the name of the city and its state is printed in the Xcode console. Select multiple cities by using **Option** + click or by using *Shift* + left-click on the keyboard. If everything went well, the simulator screenshot will look like the following:

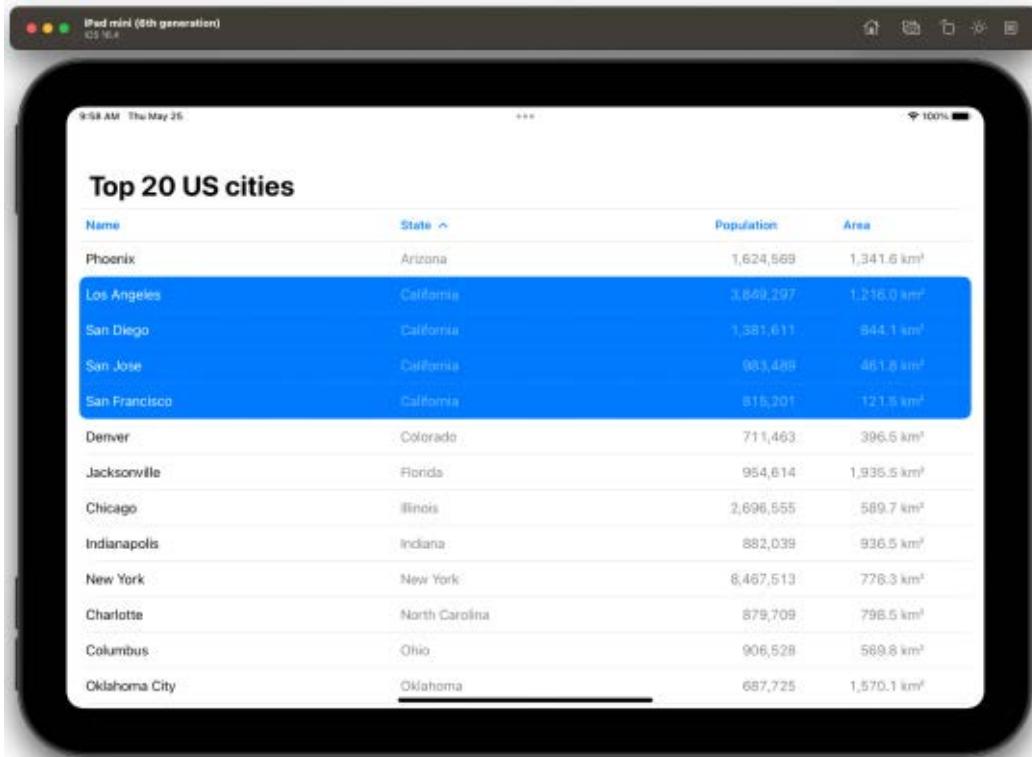


Figure 5.9: Preview of our Table with four cities selected

18. Look at the **Debug** area of the console of Xcode. You will see print statements each time we select a city or cities. It should look like the following:



```
Selected = { Los Angeles, California }
Selected = { Los Angeles, California | San Diego, California | San Francisco, California | San Jose, California }
```

Figure 5.10: Xcode console showing the four cities selected

19. Multi-row selection is only available if an external keyboard is connected to the iPad. In the case of the simulator, make sure you have the external keyboard feature enabled. From the simulator menu choose **I/O | Keyboard | Connect Hardware Keyboard**:



Figure 5.11: Selection of external keyboard on the iOS simulator

20. Let's now preview the app on an iPhone. From the Xcode toolbar, select the iPhone simulator of your choice, similar to what you did in step 6 for the iPad simulator.

Once the Xcode canvas updates the preview, it should look like the following screenshot:

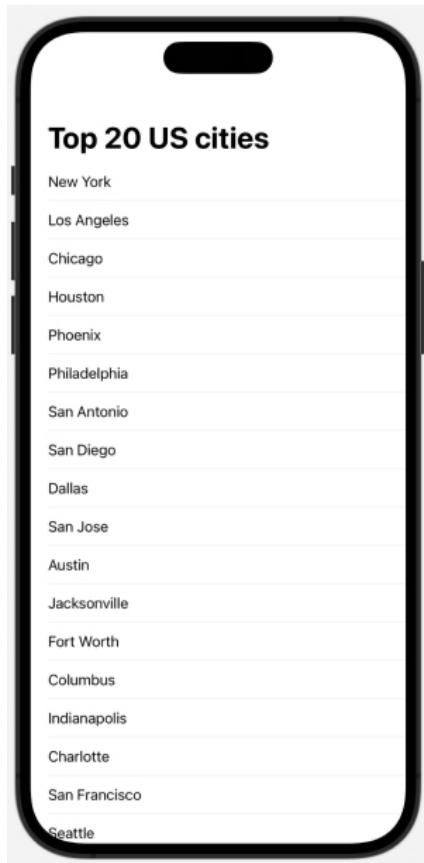


Figure 5.12: Preview of the app on the iPhone 14 Pro

What is happening here? According to Apple's documentation, in a compact horizontal size class environment, typical on iPhones (or iPads in certain modes, like **Split View** or **Slide Over**), the table has limited space to display its columns. To conserve space, the table automatically hides headers and all columns after the first when it detects this condition. Additionally, on the iPhone, multi-row selection is not available.

21. We want to provide a better user experience for the compact horizontal size class environment, which affects our iPhone users, or our iPad users using **Split View** or **Slide Over**. To do it, we will check if the horizontal size class is `UserInterfaceSizeClass.compact` and display a custom view with the four attributes of our city, or just display the city name in a regular size class environment. Go to the `ContentView` view and add these two variables:

```
@Environment(\.horizontalSizeClass) var horizontalSizeClass  
private var isCompact: Bool { horizontalSizeClass == .compact }
```

22. Now, modify the first `TableColumn` declaration in our table with:

```
 TableColumn("Name", value: \.name) { city in
    if isCompact {
        // compact view will go here
    } else {
        Text(city.name)
    }
}
```

23. Let's create a new custom view for the compact-size environment. Let's define a new struct to model our data to represent a city. Create a new Swift file by selecting **File | New | File | SwiftUI View** from the Xcode menu or by using the *Command* (⌘) + *N* key combination. Name the file `CityRowView`.
24. In the file, replace everything below the `import` statement with:

```
struct CityRowView: View {
    var city: City
    var body: some View {
        VStack(alignment: .leading) {
            LabeledContent {
                Text(city.state)
            } label: {
                Text(city.name)
                    .font(.title)
            }
            LabeledContent("Population") {
                Text(city.formattedPopulation)
                    .font(.body.monospacedDigit())
            }
            LabeledContent("Area") {
                Text(city.formattedArea)
                    .font(.body.monospacedDigit())
            }
        }
    }
}

#Preview {
    CityRowView(city: .top20USCities.first!)
        .padding()
}
```

We essentially create a custom view to display the four attributes of a city and adjust the presentation style. As for the preview, we feed sample data and force the device to an iPhone so we can see the preview in a compact horizontal size class in the Xcode canvas. If everything goes well, the preview for this custom view should look like the following:

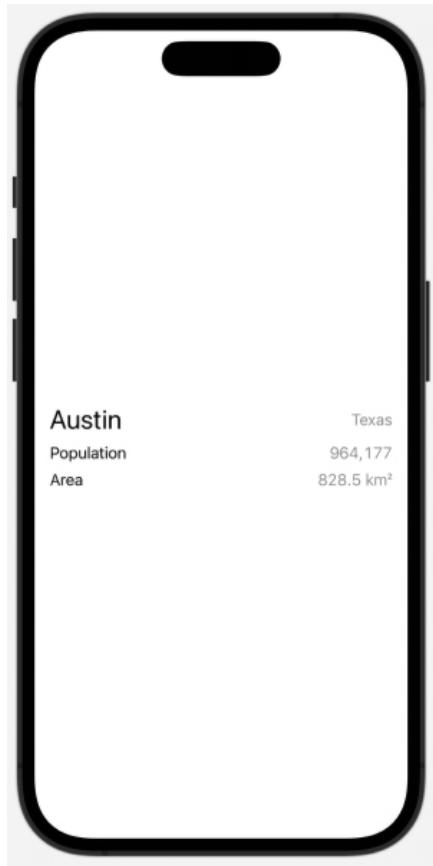


Figure 5.13: Preview of the CityRowView on the iPhone 14 Pro

25. Finally let's switch to our `ContentView` and use `CityRowView` in the first table column, which should look like this:

```
 TableColumn("Name", value: \.name) { city in
    if isCompact {
        CityRowView(city: city)
    } else {
        Text(city.name)
    }
}
```

If everything goes well, the preview on an iPhone should look like the following:

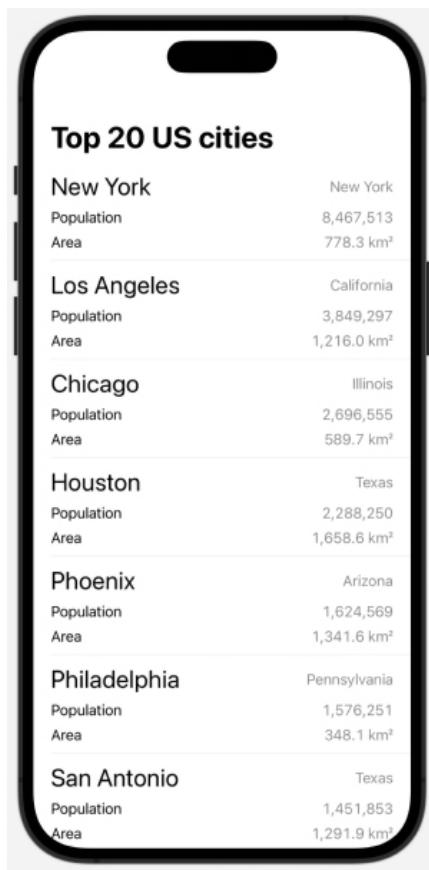


Figure 5.14: Preview of the app on the iPhone 14 Pro

If you're curious about how the app behaves on the iPad when the horizontal size class changes from regular to compact, run the app in the iPad simulator and use **Slide Over** or **Split View**.

The result should be like the following three screenshots:

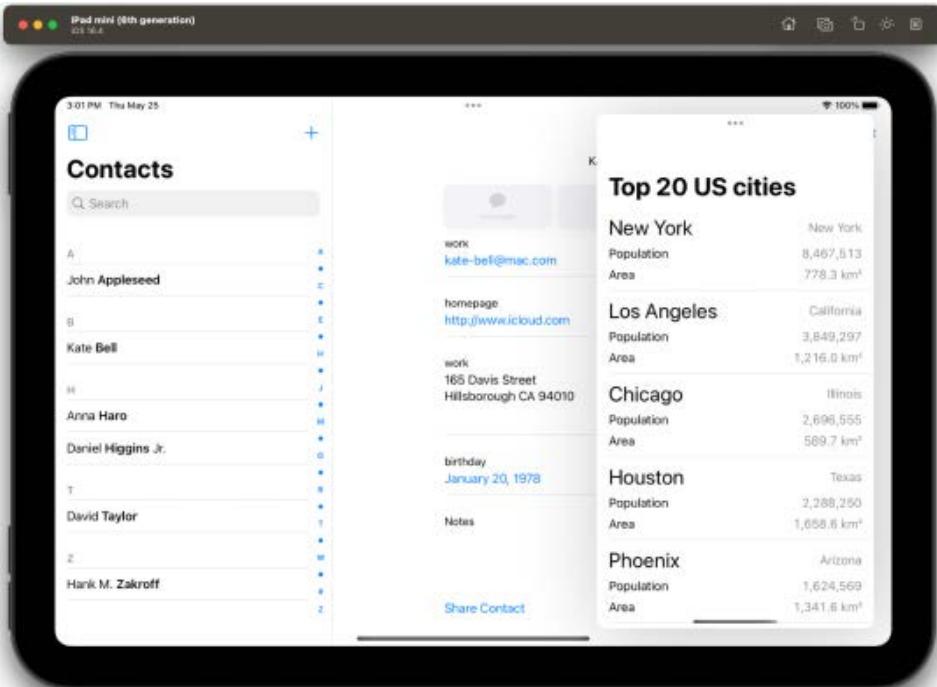


Figure 5.15: Screenshot of the app in Slide Over mode

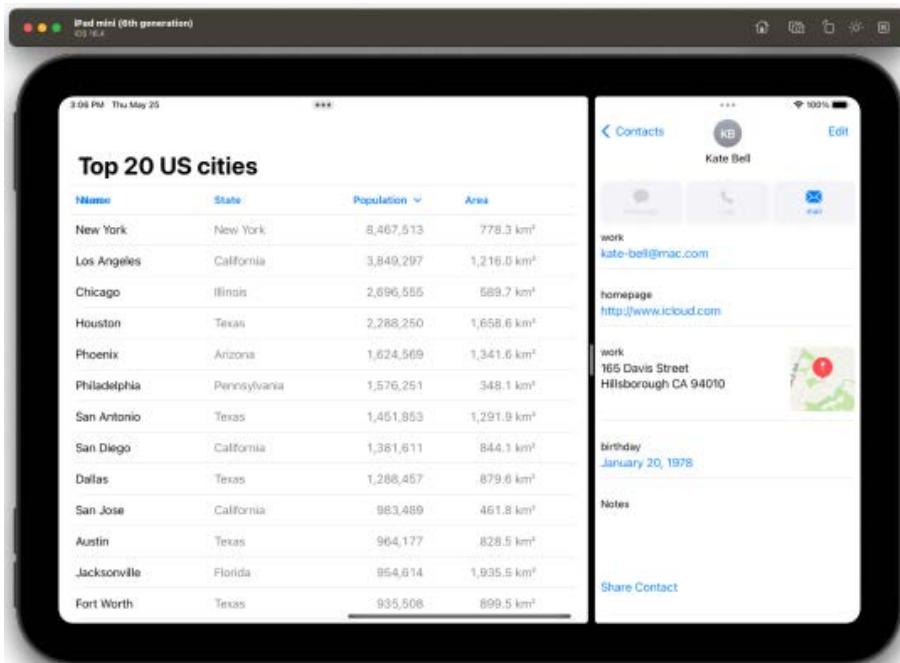


Figure 5.16: Screenshot of the app in Split View mode (two-thirds)

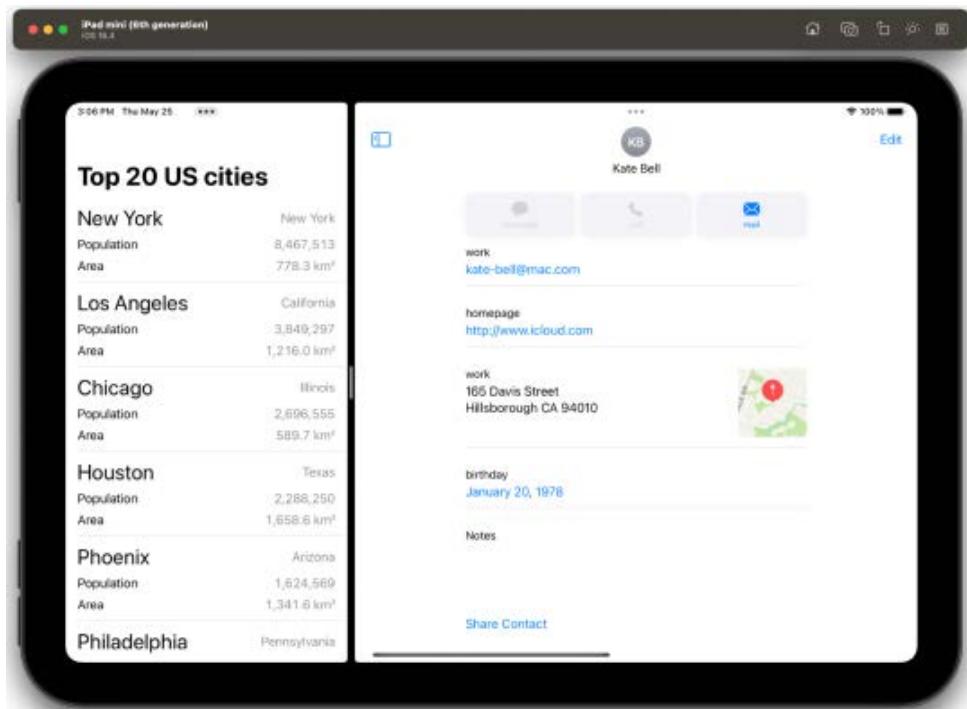


Figure 5.17: Screenshot of the app in Split View mode (one-third)

How it works...

The `Table` struct is a container view used to display selectable and sortable data arranged in rows and columns. When we create a table, we provide a collection of elements and then configure the rows and columns with the data we want. Usually, the data would be retrieved from a remote server via a network call or maybe retrieved from a local database like Core Data, Swift Data, or SQLite. In our case, and for simplicity, the data is hardcoded.

In our example app, we defined a `City` struct to model the data representing a city in the US. Although it is a simple struct, it reflects a real-world scenario because the properties have different types: the `name` and `state` properties are of the type `String`, the `population` property is of the type `Int`, and the `area` property is of the type `Measurement<UnitArea>`. Any type needs to be transformed to `String` to be displayed in a view such as `Text`, and this is why we created two computed properties: `formattedArea`, which returns the string representation of the `area` property, and `formattedPopulation`, which returns the string representation of the `population` property. We used the `NumberFormatter` and `MeasurementFormatter` classes from the Foundation framework to transform numeric values and measurement values into a `String` representation. Using these classes ensures we will have the correct formatting for different localization settings. To learn more about these classes of the Foundation framework, refer to the links provided at the end of this section.

Our app's main view, `ContentView`, defined a `Table` from an array of `City` instances, which displayed data for the top 20 cities in the US by population, ordered from the most populated city to the least populated city. We had to conform `City` to the `Identifiable` protocol to use this type to populate the table. The table was embedded in a `NavigationStack` to display a title above the column titles of the table.

In the table, we defined the four columns using the `TableColumn` struct, which takes a label, a content view, and an optional keypath. For columns displaying plain text with standard style, such as the `state` column, we used the `TableColumn` convenience initializer:

```
 TableColumn("State", value: \.state)
```

For more complex columns displaying numeric values with style and formatting, we used the full-fledged initializer:

```
 TableColumn("Area", value: \.area) { city in
    Text(city.formattedArea)
        .font(.body.monospacedDigit())
        .frame(minWidth: 100, alignment: .trailing)
}
.width(max: 150)
```

Notice how we provided the keypath `\.population`, which is different than `\.formattedPopulation`. The reason behind this choice is that we needed to sort by population, which is a numeric value, but display the `formattedPopulation` value, which is the string representation of the numeric value. Inside the trailing closure, we provide a full-fledged `Text` view with style, but we could have used a more complex view if needed.

The next task we performed was implementing the sorting functionality to allow the user to sort the table by any of the columns, in ascending or descending order. We passed a binding to an array of `KeyPathComparator` instances to the `Table`, one for each sortable column, then declared a keypath for each column to define which property should be used for sorting, and then finally, applied the `.onChange(of:)` modifier to the table to react to the change of sort order and effectively sorted the array of `City` instances with the new sort order.

We implemented row selection by passing a binding to a set of `City.ID` instances to the `Table` and then applied the `.onChange(of:)` modifier to react to selection changes. For simplicity, we printed the cities selected to the console, but in a real-world example, we would perform some action with the cities selected.

Our final step was to run the app on a compact horizontal size class, such as on an iPhone or an iPad in `Split View` or `Slide Over` modes. We saw how, when the screen size was constrained, the `Table` hid all the columns and titles and just displayed the first column. Taking this behavior into account, we created a custom view to display the four properties of our `City` instances in one column.

There's more...

Table can also support static rows or a mix of static and dynamic rows. In this case, we would use the `TableRow` struct and, using the `Table init(of:columns:rows:)` initializer, pass a trailing closure with the rows.

See also

- Apple's documentation on **Table**: <https://developer.apple.com/documentation/swiftui/tables>
- Apple's documentation on **NumberFormatter**: <https://developer.apple.com/documentation/foundation/numberformatter>
- Apple's documentation on **MeasurementFormatter**: <https://developer.apple.com/documentation/foundation/measurementformatter>
- Apple's documentation on **KeyPathComparator**: <https://developer.apple.com/documentation/foundation/keypathcomparator>

Using Grid, a powerful two-dimensional layout

In *Chapter 3, Exploring Advanced Components*, we learned about `LazyHGrid` and `LazyVGrid`. These two container views provide an efficient way of drawing views in a two-dimensional layout. They are very efficient for a large set of embedded views because they only draw the displayed or about-to-be-displayed views. This efficiency comes with the limitation of having to specify the layout in one of the two dimensions explicitly. For example, in a `LazyVGrid`, we need to specify the layout for the columns. Once SwiftUI calculates the column width depending on the horizontal space available, the number of columns to be displayed gets fixed, and the embedded views get positioned in a grid with fixed columns and an unbounded number of rows.

But what about if we wanted a view to span two columns? What about if we have a small set of views that we want to display in a two-dimensional layout, but we want full control of the alignment and spacing among them? This is where `Grid` comes into play. `Grid` was introduced with iOS 16, and it is the preferred way to arrange views in a two-dimensional layout. Apple recommends using `Grid` by default, and if the app exhibits bad performance when it first displays a large `Grid` inside a `ScrollView`, then consider switching to `LazyVGrid` or `LazyHGrid`.

In this recipe, we will create an app to showcase all the features of `Grid`. We will display a series of squares positioned in a two-dimensional 7x3 grid, with a title for each row and column, and a big title for the whole grid.

Getting ready

Create a new SwiftUI project named `TwoDimensionalLayout`.

Check the project's build settings and make sure the iOS target version is set to `17.0` or higher.

How to do it...

We'll start by creating a simple grid and then start improving the presentation style. We will define the titles for the columns and the rows first, then create a title for the whole grid, and finally, populate the grid with different square views using the `Color` view with different modifiers. The steps are as follows:

1. We are going to create a grid of 3 rows and 6 columns to place our squares. We also want an extra row for the column titles and an extra column for the row titles, so we will end up for now with 4 rows and 7 columns. Let's replace the `body` property of `ContentView` with the following code, which defines our grid:

```
struct ContentView: View {  
    var body: some View {  
        Grid {  
            GridRow {  
                Color.clear.frame(width: 0, height: 0)  
                ForEach(1..<7) { index in  
                    Text("C\((index)")  
                }  
            }  
            .frame(width: 45)  
            GridRow {  
                Text("Row 1")  
            }  
            GridRow(alignment: .top) {  
                Text("Row 2")  
            }  
            GridRow(alignment: .bottom) {  
                Text("Row 3")  
            }  
        }  
        .padding(5)  
        .border(.gray)  
    }  
}
```

If everything goes well, the grid should look as follows:

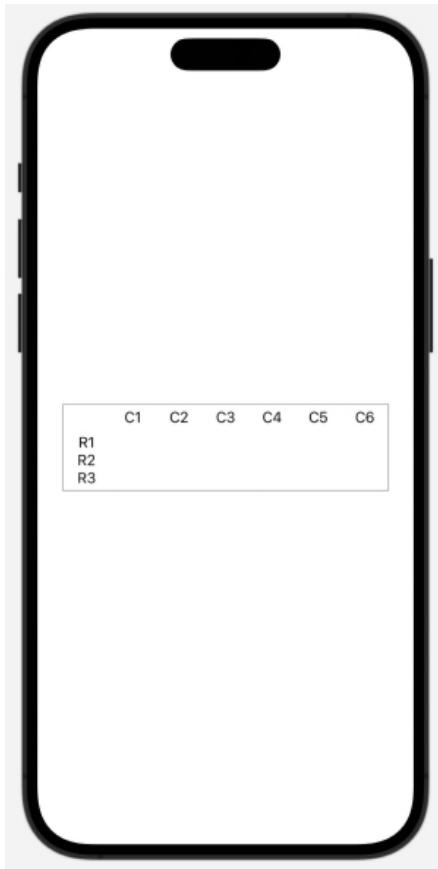


Figure 5.18: Screenshot of the initial grid

2. Now let's add the title for the grid and a divider to separate the grid title from the column titles. Right after the `Grid` declaration and before the `GridRow` declaration for the column titles, add the following code:

```
GridRow {  
    Text("My Awesome Grid")  
        .font(.largeTitle)
```

```
        .gridCellColumns(8)
    }
    Divider()
        .gridCellUnsizedAxes(.horizontal)
```

3. Let's also tell the grid how much space to use in between cells. Let's provide values for the two optional parameters of the `Grid` initializer to achieve this. The `Grid` declaration should read like this:

```
    Grid(horizontalSpacing: 5, verticalSpacing: 5)
```

4. And finally, let's lay out several squares using the `Color` view and the frame modifier. Replace the last three `GridRow` declarations with the following code:

```
GridRow {
    Text("R1")
    Color.clear.frame(width: 7.5, height: 7.5)
        .border(.gray)
    Color.black.frame(width: 15, height: 15)
        .gridColumnAlignment(.leading)
    Color.red.frame(width: 90, height: 90)
    Color.yellow.frame(width: 30, height: 30)
        .gridCellAnchor(.bottomTrailing)
}
GridRow(alignment: .top) {
    Text("R2")
    ForEach(0..<5) { _ in Color.green.frame(width: 45, height: 45) }
    Color.green.frame(width: 15, height: 15)
        .gridCellAnchor(.bottomLeading)
}
GridRow(alignment: .bottom) {
    Text("R3")
    Color.gray.frame(width: 15, height: 15)
        .gridCellAnchor(.topLeading)
    ForEach(0..<2) { _ in Color.blue.frame(width: 30, height: 30) }
    Color.brown.frame(width: 45, height: 45)
        .gridCellColumns(2)
    Color.gray.frame(width: 15, height: 15)
}
```

If everything was completed successfully, the preview should look as follows:



Figure 5.19: Screenshot of the final grid on an iPhone 15 Pro Max

How it works...

The `Grid` struct is a container view that arranges other views in a two-dimensional layout. It is common to call the arranged views, **cells**. We create this two-dimensional layout by initializing a `Grid` with a collection of `GridRow` instances, each of which represents a row in the grid. The views embedded in a `GridRow` define the columns in the grid. The first view in each row will be placed in the first column, the second view in the second column, and so on.

In our example app, in the `ContentView`, we defined a grid with five rows. The first row is for the title of the grid, the second row is for the column titles, and the last three rows are for placing square views. Each row has a variable number of cells, and the `Grid` view calculates the columns by looking at the row with the largest number of views. In our case, the row with the column titles and the row labeled R2 both have seven views, and this is why the grid has seven columns. The grid adds empty cells to the trailing edge of the rows with fewer columns.

The grid sizes its columns using the widest of its cells and sizes the rows using the tallest of its cells. For example, the height of row R1 is determined by the height of the tallest cell, which is the red cell. The width of column C3 is the widest in the grid due to the red cell too:



Figure 5.20: Row height and column width

But what happens to the title row that has a `Text` view? It is clearly the widest view, but why is the column with the row titles not as wide as the `Text` view? Because we used the modifier `.gridCellColumns(_:)` to indicate that the `Text` view spans several columns, in our case, the seven columns:

```
Text("My Awesome Grid")
    .font(.largeTitle)
    .gridCellColumns(7)
```

For the title row, we could have achieved the same layout by placing the `Text` view outside a `GridRow` and not using the modifier. Views in a grid but outside a `GridRow` struct get placed in between rows, span the maximum width available, and use the grid's `alignment` property for horizontal alignment, which, by default, is centered. This example illustrates the power of SwiftUI's declarative syntax, where a layout can be achieved in multiple different ways.

Following our explanation of views in a grid but outside the rows, we used a `Divider` between the first row and the second row. Because the divider takes the maximum width available, we used the `.gridCellUnsizedAxes(_:)` modifier of the divider to limit its width to the grid's width:

```
Divider()
    .gridCellUnsizedAxes(.horizontal)
```

When we declared our `Grid`, we defined the vertical and horizontal spacing between cells. But the `Grid` initializer also allows for an `alignment` parameter, which we didn't specify, and the `Grid` used the default `.center` alignment. This alignment lays out each view in the center of a cell. A good example is cell C1R1 with a small square view with a gray border. We can override the grid's vertical alignment for the cells in a row by passing a `VerticalAlignment` parameter to the `GridRow` initializer. Examples of this vertical alignment override are rows R2 and R3.

In a similar way, we can override the the horizontal alignment for the cells in a column by adding the view modifier `gridColumnAlignment(_:)` to exactly one of the cells in the column, and specifying a value of type `HorizontalAlignment` for the unnamed parameter. A good example of this is column C2. For total control over the alignment of a single cell, we can specify a custom alignment anchor by using the `gridCellAnchor(_:)` modifier on the cell's view. Examples of this are cells C4R1, C6R2, and C1R3.

The last thing to mention about grids is how to introduce a blank cell in a row. We did this in the row for the column titles. We introduced a blank cell before the six column titles using a `Color.clear` view. For a blank cell, we could use different options, but I would recommend a `Color.clear` view with a zero frame or with a `gridCellUnsizedAxes(_:)` modifier. Different options could be:

```
Color.clear.frame(width:0, height:0)  
Color.clear.gridCellUnsizedAxes([.horizontal, .vertical])
```

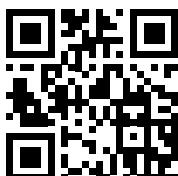
See also

- Apple's documentation on `Grid`: <https://developer.apple.com/documentation/swiftui/grid>
- Apple's documentation on `GridRow`: <https://developer.apple.com/documentation/swiftui/gridrow>
- Apple's documentation on `Alignment`: <https://developer.apple.com/documentation/swiftui/alignment>

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

<https://packt.link/swiftUI>



6

Presenting Views Modally

When we want to draw the user's attention to an important task and offer focused interaction, we present a view modally. Modality is a design technique that presents content in a separate view that prevents interaction with the parent view and requires a specific action to be dismissed. For example, suppose the users of your app 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 of whether 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 content modally 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 important information to the user modally, using alerts, confirmation dialogs, sheets, full-screen covers, popovers, and context menus.

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 and full-screen covers
- Displaying popovers
- Creating context menus

Technical requirements

The code in this chapter is based on Xcode 15.0 and iOS 17.0. You can download and install the latest version of Xcode from the App Store. You'll also need to be running macOS Ventura 13.5 or newer.

Simply search for Xcode in the App Store and select and download the latest version. Launch Xcode and follow any additional installation instructions that your system may prompt you with. Once Xcode has fully launched, you're ready to go.

All the code examples for this chapter can be found in the Chapter06 folder of this book's GitHub repository: <https://github.com/PacktPublishing/SwiftUI-Cookbook-3rd-Edition/tree/main/Chapter06-Presenting-Views-Modally>.

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 in `ContentView.swift`.

Getting ready

Create a new SwiftUI project called `PresentingAlerts`.

Make sure that the project's iOS deployment target is at least 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 content of the `body` variable 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
    }
```

- Finally, add an `.alert()` modifier that displays our alert when `showSubmitAlert` is true:

```
.alert("Confirm Actions", isPresented: $showSubmitAlert) {  
    Button("OK") {}  
} message: {  
    Text("Are you sure you want to proceed?")  
}
```

The iOS 13 and 14 steps are as follows:

- Create a new SwiftUI view called `OldAlerts.swift`.
- Create a `@State` variable that will be used to trigger the presentation of our Alert view:

```
@State private var showSubmitAlert = false
```

- 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: {  
    showSubmitAlert = true  
}) {  
    Text("Show alert")  
    .padding()  
    .background(Color.blue)  
    .foregroundColor(.white)  
    .clipShape(Capsule())  
}
```

- Finally, add an `.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 and later

We used the `alert(_:isPresented:actions:message:)` view modifier, which has 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.

iOS 13 and 14

Displaying alerts is a three-step process. First, we create a `@State` variable whose value is used to trigger the showing or hiding of the alert. Then, we add an `.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's 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 on a button. When triggered, the alert displays a message asking the user if they want to change the text currently being displayed. A tap on **OK** changes the button title, while a tap on **Cancel** leaves the text unchanged.

The steps for iOS 15 are as follows:

1. Create a `@State` variable of type `Bool` that holds the on/off state of our alert. Also, add a variable for the button title. 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 contents of the `body` variable 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 an `.alert()` modifier to the button containing two action buttons, one for changing the displayed text and another that does nothing:

```
    Button(displayedText) {  
        changeText = true  
    }.alert("Changing Text", isPresented: $changeText) {  
        Button("Yes", role: .destructive) {  
            displayedText = if (displayedText == "Stay Foolish") {  
                "Stay Hungry" } else {  
                "Stay Foolish"  
            }  
        }  
        Button("Do Nothing", role: .cancel) {}  
    } message: {  
        Text("Do you want to change the text?")  
    }
```

4. 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 tap on the **Yes** alert button toggles the displayed text between **Stay Foolish** and **Stay Hungry**. The **Do Nothing** button does nothing but dismisses 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**. Note that we assigned roles to the buttons and iOS takes care of displaying the button in the correct position and foreground color according to the role.

If we didn't provide a button with the `cancel` role, SwiftUI would add a button with the `Cancel` title to the alert. Try for yourself; comment out or delete the line of code for the **Do Nothing** button, and you'll get a standard `Cancel` button automatically.



Figure 6.3: *AlertsWithActions* preview (default Cancel button)

Presenting confirmation dialogs

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 a confirmation dialog by adding the `.confirmationDialog(_:_:isPresented:titleVisibility:actions:)` view modifier to a button with an action that changes the value of a variable. This variable will be used to determine whether the confirmation 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  
private 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("Save") {  
        print("Save action")  
    }  
    Button("Print") {  
        print("Print action")  
    }  
    Button("Update", role: .destructive) {  
        print("Update action")  
    }  
} message: {  
    Text("Use Confirmation Dialogs to present several actions")  
}
```

- Now use the live preview in the canvas to interact with the `ContentView`. Click on the `Present Confirmation Dialog` button. The result should look as follows:

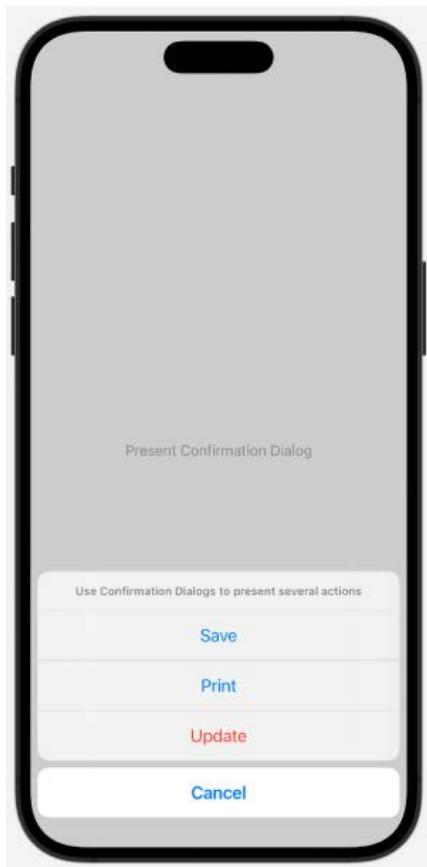


Figure 6.4: `PresentingConfirmationDialog` preview

You can click on `Cancel` to dismiss the dialog. Clicking the other three buttons displays a message in the console. Make sure the `debug` area is shown in Xcode, and the `Preview` tab is selected so you can see the `print` statements in the console.

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 also a `ViewBuilder` 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. If there is no button with the role `.cancel`, like in our code, SwiftUI adds a `Cancel` button automatically.

See also

Apple's documentation on confirmation dialogs: [https://developer.apple.com/documentation/swiftui/view/confirmationdialog\(_:_:ispresented:titlevisibility:presenting:actions:message:\)-8y541](https://developer.apple.com/documentation/swiftui/view/confirmationdialog(_:_:ispresented:titlevisibility:presenting:actions:message:)-8y541)

Presenting new views using sheets and full-screen covers

Sheets and **full-screen covers** are another way to present modals in our apps. **Sheets** are used to present views modally over existing ones while allowing the user to dismiss the sheet by dragging it down. A **full-screen cover**, or **cover** for short, is another modal view that takes as much of the screen as possible.

In this recipe, we will learn how to present several sheets to the user, how to control the height of a sheet, how to add a navigation bar with a **Dismiss** button to a sheet, and how to present a full-screen cover.

Getting ready

Create a new SwiftUI project named `PresentingSheets`.

How to do it...

We will create two SwiftUI views named `SheetView` and `SheetWithNavBar` 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 three `@State` variables of type `Bool`. The state variables will be used to trigger the presentation of different modal views:

```
@State private var isSheetPresented = false  
@State private var isSheetWithNavPresented = false  
@State private var isFullScreenCoverPresented = false
```

2. Replace the contents of the `body` variable with a `VStack` and a `Button` that changes the value of our `isSheetPresented` variable to `true` when tapped. Add a `.sheet()` modifier to the button that gets displayed when `isSheetPresented` is set to `true`:

```
VStack(spacing: 20) {  
    Button("Show sheet with drag") {  
        isSheetPresented = true
```

```
    }.sheet(isPresented: $isSheetPresented) {
        SheetView()
            .presentationDragIndicator(.visible)
    }
}
```

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`, replace the `body` property with the following:

```
var body: some View {
    ZStack {
        Color.cyan
            .ignoresSafeArea()
        Text("This is the sheet we want to display")
    }
}
```

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 6.5: Sheet with drag indicator preview

Notice the visible drag indicator. Swipe down to dismiss the sheet.

6. Now let's add a `Button` struct and `.sheet()` modifier to display a modal view with a navigation bar and a **Dismiss** button. We will also control the height of the sheet and allow interaction with the parent view:

```
Button("Show sheet with navigation bar") {
    isSheetWithNavPresented = true
}.sheet(isPresented: $isSheetWithNavPresented) {
    print("Sheet dismissed")
} content: {
    SheetWithNavBar(text: "Sheet with navigation bar and presentation
detents")
        .presentationDetents([.medium, .large])
        .presentationBackgroundInteraction(.enabled)
}
```

7. Let's create the `SheetWithNavBar` SwiftUI view used in the `Button` content:

- Create a new SwiftUI view by pressing the *Command* (⌘) + *N* keys.
- Select **SwiftUI View** and click **Next**.
- In the **Save As** field, enter `SheetWithNavBar`.
- Click **Create**.

8. Replace the `SheetWithNavBar` struct with the following:

```
struct SheetWithNavBar: View {
    @Environment(\.dismiss) private var dismiss
    var text: String = "Sheet with navigation bar"
    var body: some View {
        NavigationStack {
            ZStack {
                Color(uiColor: UIColor(white: 0.9, alpha: 1.0))
                    .ignoresSafeArea()
                Text(text)
            }
            .navigationTitle(Text("Sheet title"))
            .navigationBarTitleDisplayMode(.inline)
            .toolbar {
                Button("Dismiss") {
                    dismiss()
                }
            }
        }
    }
}
```

```
        }
        .toolbarBackground(.gray, for: .navigationBar)
        .toolbarBackground(.visible, for: .navigationBar)
    }
    .tint(.white)
}
}
```

9. Switch back to the `ContentView` view and add a new button to the top of the `VStack`, immediately before the `Show sheet with drag` button:

```
Button("Try me!") {
    print("Try me!")
}.padding(.bottom, 200)
```

10. Finally, in the Live Preview canvas, a tap on the `Show sheet with navigation bar` button should display the following:

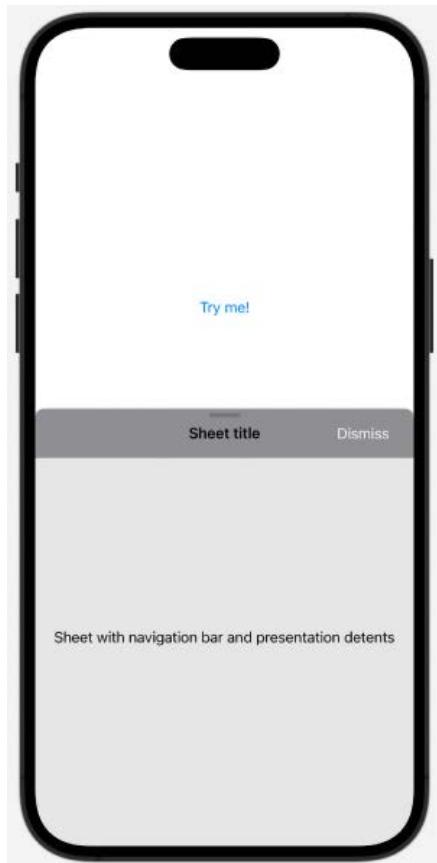


Figure 6.6: Medium height sheet with navigation bar preview



Figure 6.7: Large height sheet with navigation bar preview

Notice how the sheet occupies half the screen and the parent view allows user interactions. Touch the **Try me!** button and the action triggered will print a text in the console. Swipe up and the sheet will occupy most of the screen, disabling the user interaction with the parent view. Swipe halfway down to go back to half-screen height. To dismiss the sheet, swipe all the way down or touch the **Dismiss** button.

11. Now let's present the previous sheet using a full-screen cover. Add the following to the end of the `VStack` in `ContentView`:

```
Button("Show full screen cover") {  
    isFullScreenCoverPresented = true  
}.fullScreenCover(isPresented: $isFullScreenCoverPresented) {  
    SheetWithNavBar(text: "Full screen cover with navigation bar")  
}
```

12. Finally, in the Live Preview canvas, tap the Show full screen cover button to display the following:

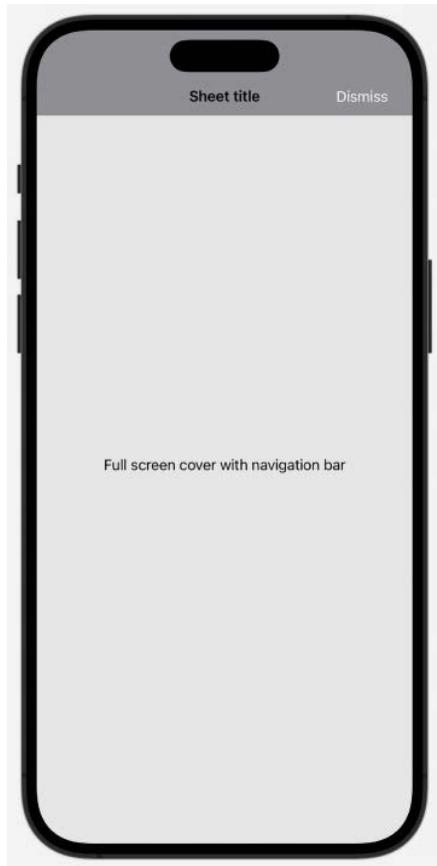


Figure 6.8: Full screen cover preview

How it works...

Our ContentView is basically a VStack that includes four different buttons. The Try me! button, which prints text to the console, is used to show how a view can still be interactive when it is partially covered by another modal view like a sheet. Each of the other buttons is used to display a sheet in different ways, using the `.sheet(isPresented:onDismiss:content:)` view modifier, which displays a sheet modally. The content of the sheet is passed as a parameter in the trailing content closure, and the sheet is displayed depending on whether the value of the `isPresented` parameter is true or false. We can also use the optional `onDismiss` closure to trigger an action when the sheet is dismissed.

Our first modal view, SheetView, is simple; it includes a Color view and a Text view inside a ZStack. The Color view is used to set the background color and extends to the whole screen thanks to the `.ignoresSafeArea()` modifier.

The second view, `SheetWithNavBar`, is presented modally as a sheet and as a full-screen cover. This view has a `NavigationStack` that includes a navigation bar with a title and a button on the trailing edge. The view has a property of type `String` with a default value, which is used in a `Text` view at the center of the content. This allows us to pass a `String` parameter when the view is instantiated. The button is used to dismiss the view using the `dismiss` environment variable of the `DismissAction` struct introduced in iOS 15. By calling the variable as a function with `dismiss()`, SwiftUI dismisses the view when presented modally.

Back to `ContentView`, the **Show sheet with drag** button displays `SheetView` modally and using the `.presentationDragIndicator(_:)` modifier displays a drag indicator at the top of the sheet.

The **Show sheet with navigation bar** button displays `SheetWithNavBar` modally with the optional `onDismiss` closure to print a message to the console once the sheet is dismissed. We apply two modifiers to the sheet. The `.presentationDetents(_:)` modifier sets two detents, `medium` and `large`, which allows the user to choose the height of the sheet by dragging the top of the sheet or touching the drag indicator. The drag indicator is automatically displayed when the sheet uses more than one detent. The second modifier, `.presentationBackgroundInteraction(_:)`, controls whether the user can interact with a view behind the modal sheet. We set the interaction to `enabled` so that when the sheet is displayed, the user can interact with the `Try me!` button in the parent view.

Finally, the **Show full screen cover** button displays a `SheetWithNavBar` modally using the `.fullScreenCover(isPresented:onDismiss:content:)` modifier, which displays the sheet covering as much of the screen as possible.

See also

- Modal presentations in SwiftUI: <https://developer.apple.com/documentation/swiftui/modal-presentations>
- Apple's human interface guidelines regarding modality: <https://developer.apple.com/design/human-interface-guidelines/modality>

Displaying popovers

A popover is a view that can be displayed onscreen to provide more information about a particular item. It includes an arrow pointing to the location where it originates from. You can tap on any other screen area to dismiss the popover. Popovers are typically used on larger screens such as on the iPad.

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 will first create a `@State` variable whose value triggers the displaying or hiding of a popover. Then we will add a `.popover()` modifier that displays the popover when the `@State` variable is true. The steps are as follows:

- 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
```

- Within the body variable, replace the contents with a `Button` that changes the value of `showPopover` to true when clicked:

```
Button("Show popover") {
    showPopover = true
}
.font(.system(size: 40))
```

- Add the `.popover()` modifier to the end of the `Button`:

```
.popover(isPresented: $showPopover) {
    Text("Popover content displayed here")
        .font(.system(size: 20))
        .padding()
}
```

Since popovers are used on the iPad, let's change the canvas preview to display on an iPad. Click on the preview device selector located in the canvas area, which shows **iPhone 15 Pro** in the following screenshot:



Figure 6.9: Set the preview with the active scheme button on Xcode

- Select any of the iPads in the list (in this case, we choose the generic **iPad 11-inch**):

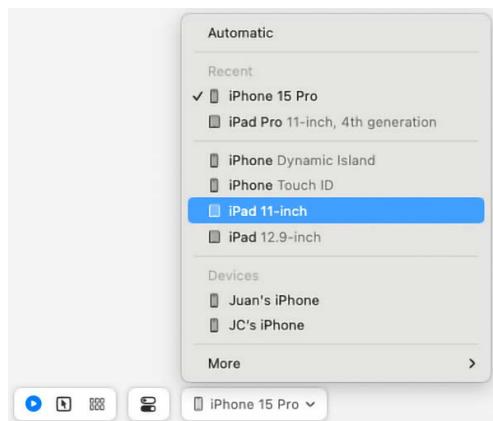


Figure 6.10: List of Xcode simulators

Run the code in the Live Preview canvas and click on the **Show popover** button to display the popover. The resulting view should be as follows:

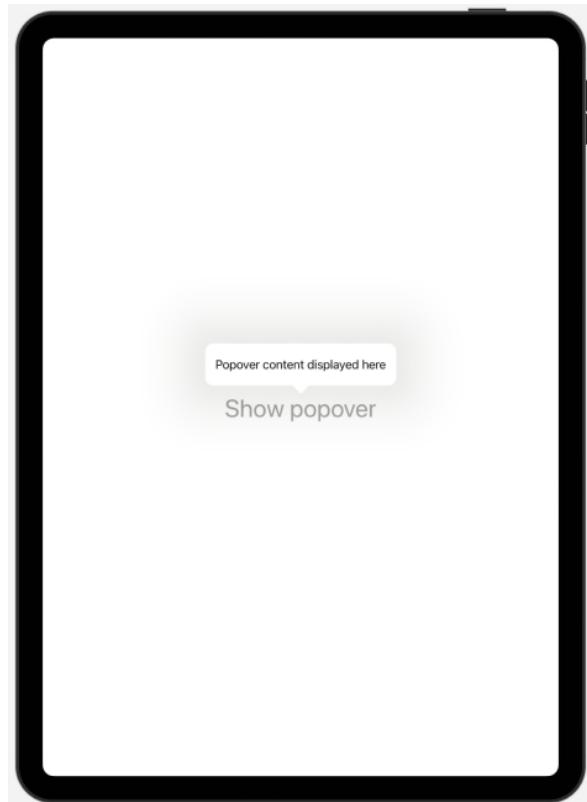


Figure 6.11: DisplayingPopovers preview

Touch any other area of the screen to dismiss the popover.

How it works...

A popover is very similar to a sheet and full-screen cover because it presents a view modally. The `.popover()` modifier takes four possible arguments, although only two were used here. Possible arguments are `isPresented`, `attachmentAnchor`, `arrowEdge`, and `content`.

The `isPresented` argument takes a binding to the `showPopover` variable that is used to trigger the display of the popover.

We did not use an `attachmentAnchor` in this recipe. An `attachedAnchor` 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`. This argument is only used in macOS, and if used in iOS, it gets ignored and has no effect.

The `content` argument is a closure that returns the content of the popover. In this recipe, a `Text` view 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, as we did for the recipe about sheets.



Popovers are only displayed in a regular-size class, such as the iPad in **full-screen** mode. In a compact view, such as the iPhone or the iPad in **slide over** or **split view** modes, SwiftUI displays the content as a sheet.

See also

Apple's human interface guidelines regarding popovers: <https://developer.apple.com/design/human-interface-guidelines/ios/views/popovers/>

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 the touch-and-hold gesture in iOS and iPadOS and a control click on macOS. Context menus consist of a collection of buttons displayed horizontally in an implicit `VStack`.

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 a `@State` variable to hold the current color of the bulb and change its value within the context menu. The steps are as follows:

- Just above the body variable in `ContentView.swift`, add a `@State` variable to hold the color of the bulb. Initialize it to red:

```
@State private var bulbColor = Color.red
```

- Within the body variable, change the `Text` struct to an `Image` struct that displays a light bulb from SF Symbols:

```
Image(systemName: "lightbulb.fill")
```

- Add `.font()` and `.foregroundStyle()` modifiers to the image. Change the font size to 60 and the foreground color to our `bulbColor` variable:

```
.font(.system(size: 60))  
.foregroundStyle(bulbColor)
```

4. Add a `.contextMenu()` modifier with three buttons. Each Button changes the value of our `bulbColor` state variable to a new color:

```
.contextMenu {  
    Button("Yellow Bulb") {  
        bulbColor = .yellow  
    }  
    Button("Blue Bulb") {  
        bulbColor = .blue  
    }  
    Button("Green Bulb") {  
        bulbColor = .green  
    }  
}
```

Run the live preview and long-press on the light bulb icon. A context menu is displayed, as seen in the following figure:



Figure 6.12: DisplayingContextMenus live preview

You can choose any of the options in the context menu to change the light bulb color.

How it works...

The `.contextMenu(menuItems:)` modifier can be attached to any view. In this recipe, we attached it to an `Image` view such that a touch-and-hold gesture on the `image` view displays the context menu. The `.contextMenu()` modifier contains buttons whose action closures perform a particular function. In our case, each `Button` changes our `@State` `bulbColor` variable to a new color, thereby updating our `image` view.

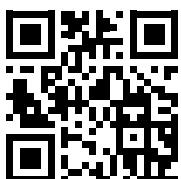
See also

Apple's documentation on the context menu: [https://developer.apple.com/documentation/swiftui/view/contextmenu\(menuitems:\)](https://developer.apple.com/documentation/swiftui/view/contextmenu(menuitems:))

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

<https://packt.link/swiftUI>



7

Navigation Containers

When we refer to navigation, we are referring to the different design techniques that provide the user experience of our app. Apps with good navigation are intuitive and easy to use, and users can focus on the content and experience. Our goal when implementing navigation is to allow people to discover the content and interact with it, without even paying attention to how to do it.

In the previous chapter, we learned about how to present views modally, to draw the attention of the user to an important task, and to offer focused interaction. In this chapter, we will learn about showing our content on different screens, and how to navigate among these. We show content on different screens to better organize our app and separate different user experiences.

Apple provides several components to implement navigation. Some components are meant to be used at the top level of the navigation hierarchy, providing a logical grouping and separation of different experiences or flows. Some other components are part of one action or flow, or provide visual feedback to the user. **Navigation containers** provide structure to the app's user interface and enable users to move among different parts of the app.

In this chapter, we will cover the following recipes:

- Simple navigation with `NavigationStack`
- Typed data-driven navigation with `NavigationStack`
- Untyped data-driven navigation with `NavigationStack`
- Multi-column navigation with `NavigationSplitView`
- Navigating between multiple views with `TabView`
- Programmatically switching tabs on a `TabView`

Technical requirements

The code in this chapter is based on Xcode 15.0 and iOS 17.0. You can download and install the latest version of Xcode from the App Store. You'll also need to be running macOS Ventura (13.5) or newer.

Simply search for Xcode in the App Store and select and download the latest version. Launch Xcode and follow any additional installation instructions that your system may prompt you with. Once Xcode has fully launched, you're ready to go.

All the code examples for this chapter can be found on GitHub at <https://github.com/PacktPublishing/SwiftUI-Cookbook-3rd-Edition/tree/main/Chapter07-App-Navigation>.

Simple navigation with NavigationStack

Throughout the previous chapters, we have seen different apps using `NavigationStack` and `NavLink` to provide simple navigation. So, let's learn about `NavigationStack` in detail. A navigation stack is ideal for displaying hierarchical content because it allows users to go from a more general view to a more detailed view of the content. The user goes down the content hierarchy by performing an action in the top-level view, usually a tap on a button. When the action occurs, a new view replaces the original view. We call this action pushing the view onto the navigation stack. At the top of this new view is a navigation bar with a title and a back button. The user can go back to the previous view by tapping on the back button, and we call this action popping the view from the navigation stack. This is the most common navigation in iOS apps, present since the beginning of iOS more than a decade ago, and everyone is familiar with it.

In this recipe, we will implement an app with several custom views to illustrate how a navigation stack works.

Getting ready

Create a new SwiftUI iOS app named `SimpleNavigation`.

How to do it...

In our app, from the main view, we will push the custom views onto a navigation stack. The main view will be the root view of our navigation stack. Let's start creating the custom views first. Here are the steps to do so.

1. Create a new SwiftUI view named `ChildAView`, and replace the body and preview macro with the following:

```
struct ChildAView: View {
    @State private var title = "Child A"
    var body: some View {
        VStack {
            Image(systemName: "a.square")
                .font(.system(size: 80))
                .foregroundStyle(.yellow)
            Text("This is the Child A View")
                .foregroundStyle(.primary)
```

```
        .padding()
    }
    .navigationTitle($title)
    .navigationBarTitleDisplayMode(.inline)
}
}

#Preview {
    NavigationStack {
        ChildAView()
    }
}
```

The preview should look like this:

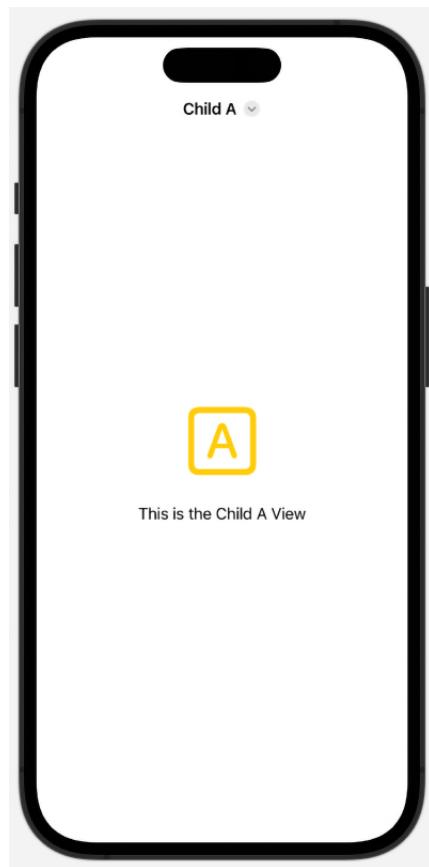


Figure 7.1: ChildAView preview

2. Create another SwiftUI view named `ChildBView`. This view is going to be practically identical to the previous one:

```
struct ChildBView: View {  
    var body: some View {  
        VStack {  
            Image(systemName: "b.square")  
                .font(.system(size: 80))  
                .foregroundStyle(.brown)  
            Text("This is the Child B View")  
                .foregroundStyle(.primary)  
                .padding()  
        }  
        .navigationTitle("Child B")  
        .navigationBarTitleDisplayMode(.inline)  
    }  
}  
#Preview {  
    NavigationStack {  
        ChildBView()  
    }  
}
```

And the preview should look like this:



Figure 7.2: ChildBView preview

3. Create a new SwiftUI view named `SheetView`. The content should be as follows:

```
struct SheetView: View {
    var body: some View {
        NavigationStack {
            ZStack {
                Color(uiColor: UIColor(red: 0, green: 0.9, blue: 1,
alpha: 1))
                    .ignoresSafeArea()
                Text("Sheet with navigation bar")
            }
            .navigationTitle("Sheet title")
            .navigationBarTitleDisplayMode(.large)
            .toolbarBackground(.teal, for: .navigationBar)
            .toolbarBackground(.visible, for: .navigationBar)
        }
    }
}
```

Now, let's create four new SwiftUI views: `FirstView`, `SecondView`, `ThirdView`, and `FourthView`. These four views are meant to be in a navigation stack, so we will adjust each preview macro to use a `NavigationStack`.

4. Switch to `FirstView` and replace the content with the following:

```
struct FirstView: View {
    var body: some View {
        VStack {
            Image(systemName: "circle")
                .font(.system(size: 80))
                .foregroundStyle(.cyan)
            Text("This is the First View")
                .foregroundStyle(.primary)
                .padding()
        VStack(spacing: 20) {
            NavigationLink("Show Child A") {
                ChildAView()
            }
            NavigationLink("Show Child B") {
                ChildBView()
            }
        }
    }
}
```

```
        .padding()
        .navigationTitle("First")
    }
}
#Preview {
    NavigationStack {
        FirstView()
    }
}
```

The preview should look like this:

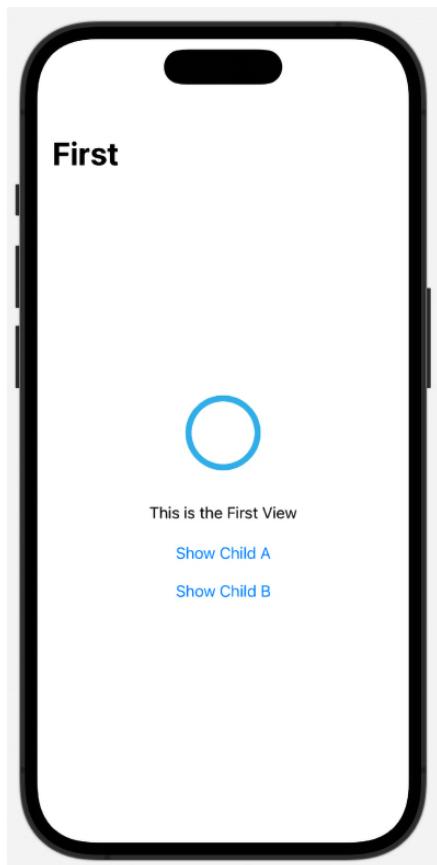


Figure 7.3: FirstView preview

5. SecondView should be as follows:

```
struct SecondView: View {
    var body: some View {
        VStack {
            Image(systemName: "square")
                .font(.system(size: 80))
                .foregroundStyle(.blue)
            Text("This is the Second View")
                .foregroundStyle(.primary)
                .padding()
            List {
                NavigationLink("Show Child A") {
                    ChildAView()
                }
                NavigationLink("Show Child B") {
                    ChildBView()
                }
            }
            .listStyle(.inset)
            .frame(height: 160)
        }
        .padding()
        .navigationTitle("Second")
    }
}
#Preview {
    NavigationStack {
        SecondView()
    }
}
```

The preview should look like this:

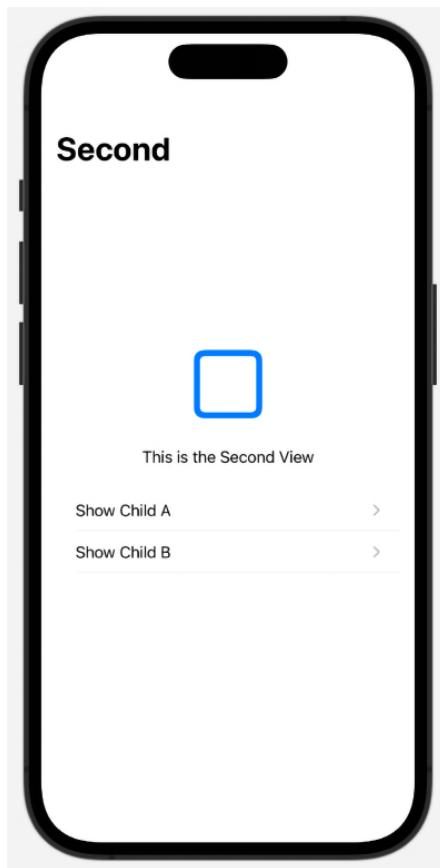


Figure 7.4: SecondView preview

6. ThirdView should be as follows:

```
struct ThirdView: View {
    @Environment(\.dismiss) private var dismiss
    var body: some View {
        ScrollView {
            Image(systemName: "triangle")
                .font(.system(size: 80))
                .foregroundStyle(.teal)
                .padding(.vertical, 50)
            Text("This is the Third View")
                .foregroundStyle(.primary)
                .padding()
        }
        .padding()
        .navigationTitle("Third")
        .toolbarBackground(.quaternary, for: .navigationBar)
        .navigationBarBackButtonHidden()
        .toolbar {
            Button("Dismiss") {
                dismiss()
            }
        }
    }
}
#Preview {
    NavigationStack {
        ThirdView()
    }
}
```

The preview should look like this:

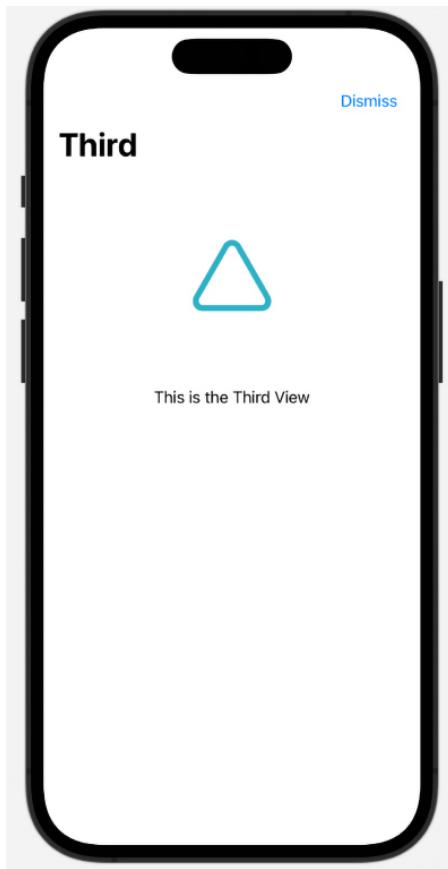


Figure 7.5: *ThirdView* preview

7. `FourthView` should be as follows:

```
struct FourthView: View {
    @State private var isSheetPresented = false
    var body: some View {
        ScrollView {
            Image(systemName: "capsule")
                .font(.system(size: 80))
                .foregroundStyle(.red)
                .padding(.vertical, 50)
            Text("This is the Fourth View")
                .foregroundStyle(.primary)
                .padding()
            Button("Show sheet") {
                isSheetPresented = true
            }.sheet(isPresented: $isSheetPresented) {
                SheetView()
                    .presentationDetents([.fraction(0.75)])
                    .presentationDragIndicator(.visible)
            }
        }
        .navigationTitle("Fourth")
        .toolbarBackground(.quaternary, for: .navigationBar)
        .toolbarBackground(.visible, for: .navigationBar)
    }
}
#Preview {
    NavigationStack {
        FourthView()
    }
}
```

The preview should look like this:

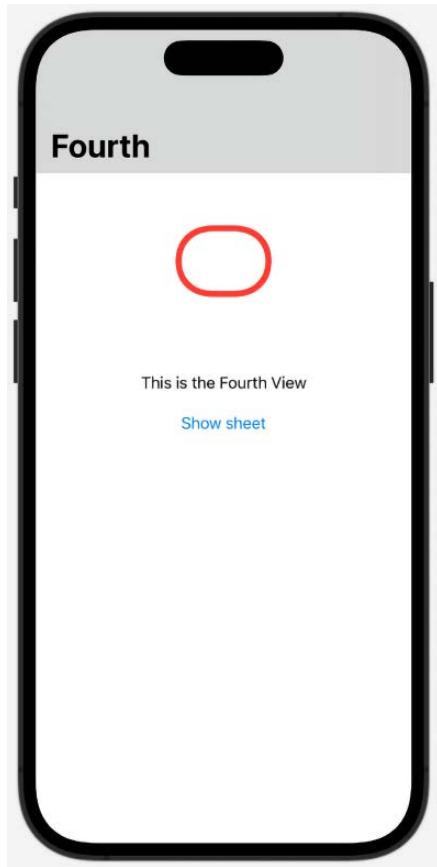


Figure 7.6: FourthView preview

8. Now, let's finally go to the ContentView and create the navigation stack:

```
struct ContentView: View {
    var body: some View {
        NavigationStack {
            List {
                NavigationLink("Show First View") {
                    FirstView()
                }
                NavigationLink("Show Second View") {
                    SecondView()
                }
            }
        }
    }
}
```

```
        NavigationLink("Show Thrid View") {
            ThirdView()
        }
        NavigationLink("Show Fourth View") {
            FourthView()
        }
    }
    .navigationTitle("Top Level")
}
.tint(.teal)
}
```

9. Go ahead and use the live preview feature with `ContentView` to interact with the app. Play around for a few minutes with the app before you read the details on how it works.

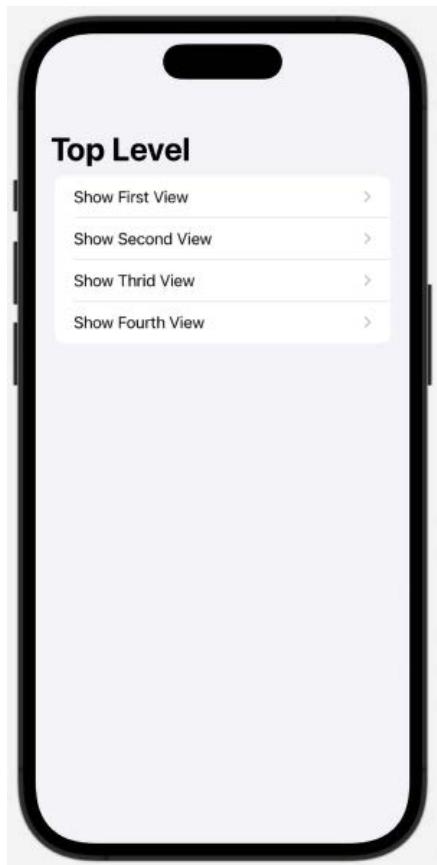


Figure 7.7: `ContentView` preview

How it works...

We use the `NavigationStack` to present a stack of views over a root view. We can add views to the stack by adding `NavLink` controls inside the `NavigationStack`. `NavLink` is an interactive control, and when the user taps on it, a new view is pushed onto the navigation stack. The view pushed onto the stack can be dismissed with platform-appropriate controls, like the back button in the leading edge of a navigation bar provided by default in iOS. The root view is the one that contains the `NavigationStack`, and it cannot be dismissed. In our example app, `ContentView` is the root view, containing a `List` of four `NavLink` instances. Each of these navigation links pushes a different view onto the stack. If `NavLink` is outside a `NavigationStack`, the link is disabled, so it appears grayed out and not interactive.

When a view is inside a navigation stack and we apply the `navigationTitle(_:)` view modifier to the view, the title is displayed in the navigation bar. In the simplest form of this modifier, we pass a string to be the title, but we can pass a binding or a custom view too. If we pass a binding, the navigation bar will have a button to edit the title. Try for yourself and see how it works on `ChildAView`. Note that this view modifier needs to be applied to the views inside a navigation stack and not to the navigation stack.

We also use the `navigationBarTitleDisplayMode` modifier to configure the appearance of the title in the navigation bar: `large`, `inline`, or `automatic`.

`FirstView` and `SecondView` are practically identical with the difference that in `FirstView` we use a `VStack` to enclose the two `NavLink` instances to `ChildAView` and `ChildBView`, and in `SecondView`, we use a `List`. See how SwiftUI gives a different appearance to the navigation links. Choose the approach that works best for your user experience.

`ThirdView` and `FourthView` control the appearance of the navigation bar using the `toolbarBackground` modifier.

`ThirdView` hides the back button of the navigation bar, using the `navigationBarBackButtonHidden` modifier, and adds a dismiss button to the navigation bar, using the `toolbar(content:)` modifier and providing a `ToolBarItem` with the `topBarTrailing` placement. To add views to a toolbar, we can either enclose each view in a `ToolBarItem`, as we did with the `Button` with the `dismiss()` action, or use a `ToolBarItemGroup` to enclose one or more views.

`FourthView` displays a `SheetView` as a sheet. In order to understand the inner workings of the sheet, you can refer to the recipe *Presenting new views using sheets and full-screen covers* in Chapter 6.

See also

- Navigation bars: <https://github.com/PacktPublishing/SwiftUI-Cookbook-3rd-Edition/tree/main/Resources/Chapter07/recipe3>
- Configure your app navigation titles: <https://developer.apple.com/design/human-interface-guidelines/navigation-bars>
- Toolbars: <https://developer.apple.com/documentation/swiftui/toolbars>

Typed data-driven navigation with NavigationStack

When `NavigationStack` was introduced with iOS 16, `NavigationView` was deprecated, and `NavLink` was updated with new functionality. If you use old code with `NavigationView`, it can be directly replaced with `NavigationStack` in its simplest form, as we explained in the previous recipe. For more complex cases, Apple has a document with recommendations on how to migrate to `NavigationStack` and the new `NavLink` functions.

Two of the most exciting features of `NavigationStack` are: 1) navigation is driven by data, and 2) we separate the view code from the navigation code. Since the navigation is driven by data, it is possible to implement arbitrary navigation to any of the views in a navigation stack by manipulating the data. This opens the possibility for flexible navigation patterns like the one needed for deep links.

In this recipe, we will create an app to keep track of our favorite foods to illustrate how a typed data-driven navigation stack works.

Getting ready

Create a new SwiftUI iOS app named `ModernNavigation`.

How to do it...

Our app is going to display in its root view a list of different types of food grouped by category, such as fruit, vegetable, or meat. Each entry on the list will be a view, with the name and a small thumbnail picture of the food. Once we tap on an entry, we will navigate to a detailed view of the selected food, which will contain the name, category, and picture of the food, occupying the top half of the screen. In the bottom half of the screen, we will display another list of navigation links representing the last three most recently viewed foods. Finally, we will add a button to the navigation bar to go back to the root view, using typed data-driven navigation. These are the steps:

1. Download the project images from the GitHub link at <https://github.com/PacktPublishing/SwiftUI-Cookbook-3rd-Edition/tree/main/Resources/Chapter07/recipe3>. These images were obtained royalty-free from Pexels: <https://www.pexels.com>. You can check out the website and get free pictures for your own projects.
2. Drag and drop the downloaded images for this recipe into the project's `Assets.xcassets` directory or the `Assets` folder in Xcode, as shown in the following screenshot:

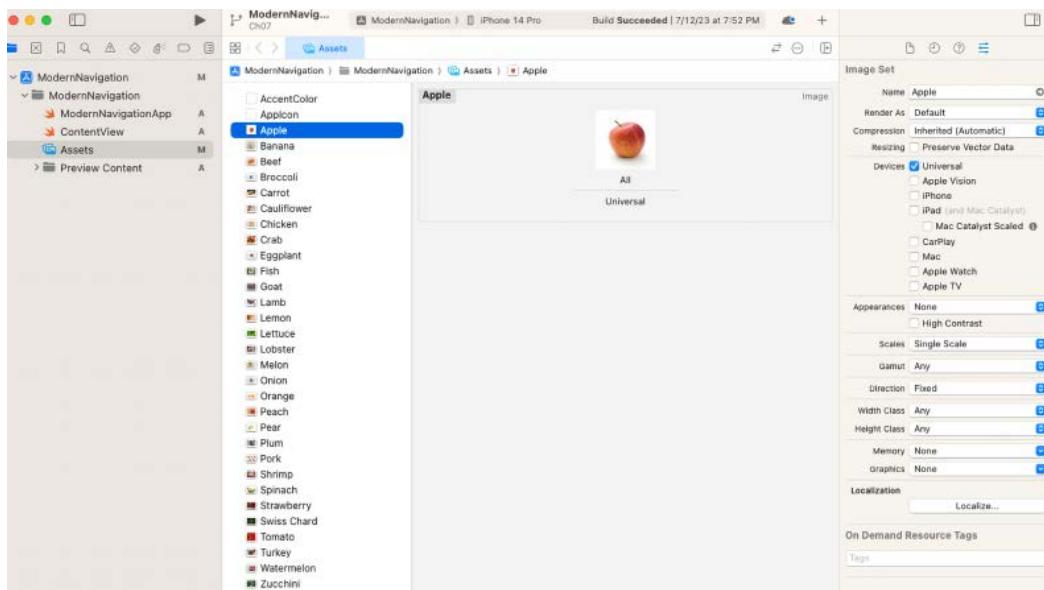


Figure 7.8: Assets folder in Xcode

3. Let's start by creating our model classes to define our food. Create a new file Swift file named **Category.swift**. Then, create the following enum type to model our food categories:

```
enum Category: Int, Identifiable, CaseIterable, Hashable {
    case meat
    case vegetable
    case fruit
    var id: Int { rawValue }

    var name: String {
        switch self {
        case .meat: "Meat"
        case .vegetable: "Vegetable"
        case .fruit: "Fruit"
        }
    }
}
```

4. Now, let's model our food. Create a new file Swift file named `Food.swift`. Then, create the following `struct` type:

```
struct Food: Identifiable, Hashable {  
    let id = UUID()  
    var name: String  
    var category: Category  
}
```

5. In the same file, let's extend the `Food` struct to have an array of the food instances we will use in the app:

```
extension Food {  
    static let samples: [Food] = [  
        Food(name: "Apple", category: .fruit),  
        Food(name: "Pear", category: .fruit),  
        Food(name: "Orange", category: .fruit),  
        Food(name: "Lemon", category: .fruit),  
        Food(name: "Strawberry", category: .fruit),  
        Food(name: "Plum", category: .fruit),  
        Food(name: "Banana", category: .fruit),  
        Food(name: "Melon", category: .fruit),  
        Food(name: "Watermelon", category: .fruit),  
        Food(name: "Peach", category: .fruit),  
        Food(name: "Pork", category: .meat),  
        Food(name: "Beef", category: .meat),  
        Food(name: "Lamb", category: .meat),  
        Food(name: "Goat", category: .meat),  
        Food(name: "Chicken", category: .meat),  
        Food(name: "Turkey", category: .meat),  
        Food(name: "Fish", category: .meat),  
        Food(name: "Crab", category: .meat),  
        Food(name: "Lobster", category: .meat),  
        Food(name: "Shrimp", category: .meat),  
        Food(name: "Carrot", category: .vegetable),  
        Food(name: "Lettuce", category: .vegetable),  
        Food(name: "Tomato", category: .vegetable),  
        Food(name: "Onion", category: .vegetable),  
        Food(name: "Broccoli", category: .vegetable),  
        Food(name: "Cauliflower", category: .vegetable),  
        Food(name: "Eggplant", category: .vegetable),  
        Food(name: "Swiss Chard", category: .vegetable),  
        Food(name: "Spinach", category: .vegetable),  
    ]
```

```
        Food(name: "Zucchini", category: .vegetable)
    ]
}
```

6. We will define a new class, `Storage`, to store our food samples and the most recently viewed food. This new type needs to be a class because we will make it conform to `ObservableObject` and declare several `@Published` properties. Our views will be able to subscribe to any published property and react to value updates. Our class will have methods to initialize the object, get the types of food in a category, and add a food to our recent list of food. Create a new Swift file named `Storage.swift` and define the following class:

```
final class Storage: ObservableObject {
    @Published var food: [Food]
    @Published var recents: [Food]

    init(food: [Food], recents: [Food] = []) {
        self.food = food
        self.recents = recents
    }
    func food(in category: Category) -> [Food] {
        food
            .filter { $0.category == category }
            .sorted { $0.name < $1.name }
    }

    func addRecent(_ food: Food) {
        if !recents.contains(food) {
            recents = [food] + Array(recents.prefix(2))
        }
    }
}
```

7. To finalize our model, we will create another class named `Navigation` to store our navigation data. Our views will use this class, which conforms to `ObservableObject`, has one `@Published` property to keep track of the navigation path, and a method to navigate back to the root view. Create a new Swift file named `Navigation.swift` and define the following class:

```
final class Navigation: ObservableObject {
    @Published var path = [Food]()

    func popToRoot () {
        path = .init()
    }
}
```

8. Now, let's work on our views. Create a new SwiftUI View file named `CategoryView.swift`. This view will be very simple, displaying the fruit category name in a capsule with a colored background. The file contents should be as follows:

```
struct CategoryView: View {
    var category: Category
    var color: Color {
        switch category {
        case .fruit: .yellow
        case .meat: .red
        case .vegetable: .green
        }
    }
    var body: some View {
        Text(category.name)
            .foregroundStyle(.white)
            .padding(.all, 8)
            .background(color)
            .clipShape(Capsule())
    }
}
#Preview {
    VStack(spacing: 10) {
        ForEach(Category.allCases) { category in
            CategoryView(category: category)
        }
    }
}
```

If everything went well, the preview should include three capsules, one capsule for every category. It should look like the following screenshot:



Figure 7.9: CategoryView preview showing the three categories

9. Our main screen will contain a list of our food with a vertical scroll. We will create a row view to display our food as an entry in the scrolling list. We will include the name of the food and a small thumbnail picture for the row view. Create a new SwiftUI View file named `FoodRowView.swift`. The contents of the file should be as follows:

```
struct FoodRowView: View {
    var food: Food
    var body: some View {
        LabeledContent {
            Image(food.name)
                .resizable()
                .aspectRatio(contentMode: .fit)
                .frame(height: 40)
        } label: {
            Text(food.name)
        }
    }
}

#Preview {
    let fruits = Array(Food.samples.prefix(4))
    return NavigationStack {
        List(fruits) { fruit in
            NavigationLink(value: fruit) {
                FoodRowView(food: fruit)
            }
        }
        .navigationTitle(fruits.first!.category.name)
    }
}
```

The preview should include four rows and look like the following:

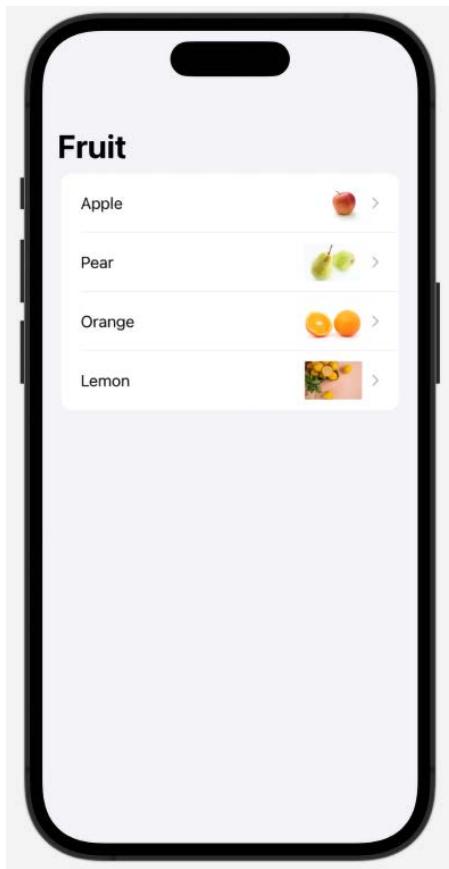


Figure 7.10: FoodRowView preview showing four rows

10. When the user taps on a row representing a type of food, we will show a detailed view of the food. The detailed view will contain the name, the category, and a big picture of the food instead of a thumbnail. Let's create a new SwiftUI file named `FoodView.swift`. The contents of the file should be as follows:

```
struct FoodView: View {  
    var food: Food  
    var body: some View {  
        VStack(alignment: .leading) {  
            CategoryView(category: food.category)  
                .padding(.leading)  
            Image(food.name)  
                .resizable()  
                .aspectRatio(contentMode: .fit)  
                .frame(maxWidth: .infinity, maxHeight: 300)  
            Spacer()  
        }  
        .navigationTitle(food.name)  
    }  
}  
  
#Preview {  
    let fruits = Array(Food.samples.prefix(4))  
    return NavigationStack {  
        FoodView(food: fruits.first!)  
    }  
}
```

The preview should look like this:



Figure 7.11: FoodView preview showing the main content

11. Let's improve the detail view to track the most recently viewed food types. At the top of the `FoodView` struct, right above the variable `food`, add another variable:

```
@EnvironmentObject private var storage: Storage
```

Then, in the body property, replace the `Spacer()` with the following code:

```
if storage.recents.isEmpty {
    Spacer()
} else {
    List {
        Section("Recents") {
            ForEach(storage.recents) { recent in
                NavigationLink(value: recent) {
                    FoodRowView(food: recent)
                }
                .disabled(recent == food)
            }
        }
    }
    .listStyle(.grouped)
}
```

12. To make the preview work again, we need to inject an environment object to `FoodView`. Add the following `environmentObject` modifier to the preview, which should be as follows:

```
#Preview {
    let fruits = Array(Food.samples.prefix(3))
    return NavigationStack {
        FoodView(food: fruits.first!)
            .environmentObject(Storage(food: Food.samples, recents:
        fruits))
    }
}
```

If everything went well, the preview should show three rows for the recently viewed types of food. It should look like the following:

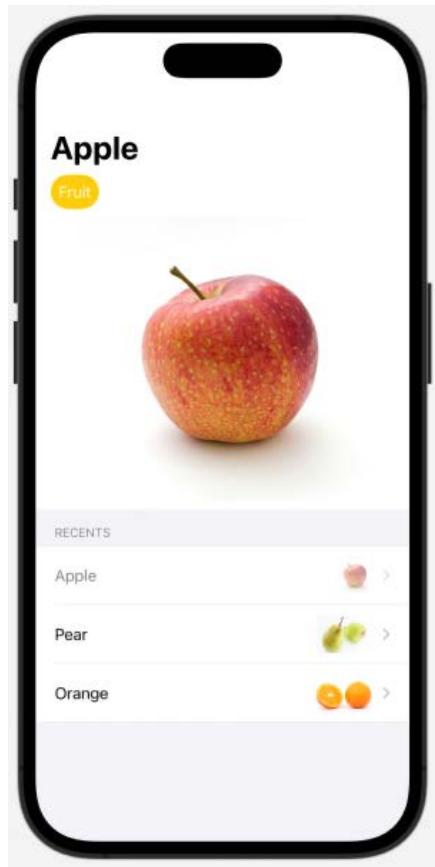


Figure 7.12: FoodView preview showing the main content and the most recently viewed food

13. The final touch to our detailed view is to add a button in the navigation bar to go back to the root view. This is necessary because, as we would see when we implement the root view, tapping on one of the most recently viewed food types at the bottom of the detail view will navigate to a new detail view. With this interactivity, we can keep navigating to different detail views, adding more and more views to the navigation stack. At one point the navigation stack can have several views, and the best way to go back to the root view is with a convenience button to perform the action. This is possible because of the data-driven navigation that keeps track of our navigation path. Let's first add at the top of the FoodView struct another environment object:

```
@EnvironmentObject private var navigation: Navigation
```

14. At the end of the body, right after the `navigationTitle` modifier, add the following code, which creates, in the trailing edge of the navigation bar, a button to navigate back to the root view:

```
.toolbar {
    Button {
        navigation.popToRoot()
    } label: {
        Image(systemName: "list.bullet")
    }
}
```

15. Right after the code for the button, add the following code, which will add the food to the recently viewed food, when we navigate away from the detail view:

```
.onDisappear {
    storage.addRecent(food)
}
```

16. Finally, let's inject a `Navigation` object into the preview environment so that it does not crash. Add another modifier to `FoodView` in the preview, which should look like this:

```
#Preview {
    let fruits = Array(Food.samples.prefix(3))
    return NavigationStack {
        FoodView(food: fruits.first!)
            .environmentObject(Navigation())
            .environmentObject(Storage(food: Food.samples, recents:
fruits))
    }
}
```

If everything went well, the preview in the Xcode canvas should look like this:

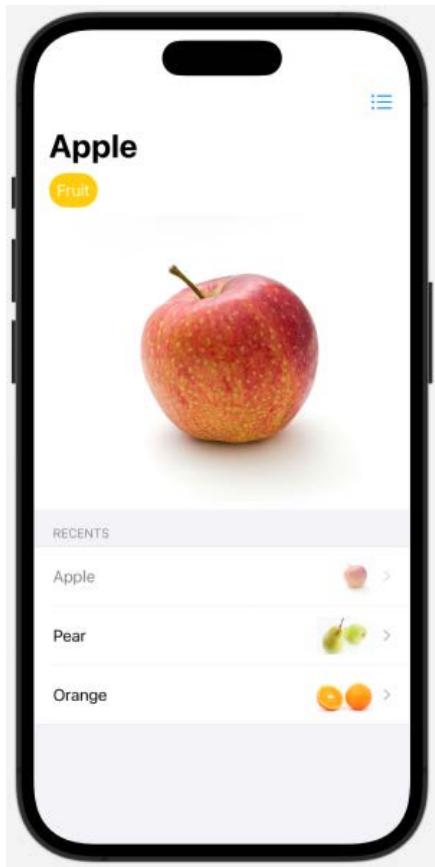


Figure 7.13: FoodView preview showing the pop-to-root button in the navigation bar

17. Now, we are finally ready to work on the root view of the app. Go to the `ContentView` and add the two `@StateObject` variables at the top, right after the struct declaration:

```
@StateObject private var navigation = Navigation()  
@StateObject private var storage = Storage(food: Food.samples)
```

18. And now, we can finally create our list of food types. Our list will have a section for each food category. Replace the `body` property with the following:

```
var body: some View {  
    NavigationStack(path: $navigation.path) {  
        List(Category.allCases) { category in  
            Section(category.name) {  
                ForEach(storage.food(in: category)) { food in  
                    NavigationLink(value: food) {
```

```
        FoodRowView(food: food)
    }
}
}
.navigationTitle("My Food")
.navigationDestination(for: Food.self) { food in
    FoodView(food: food)
        .environmentObject(navigation)
        .environmentObject(storage)
}
}
}
```

The preview should look like the following screenshot:

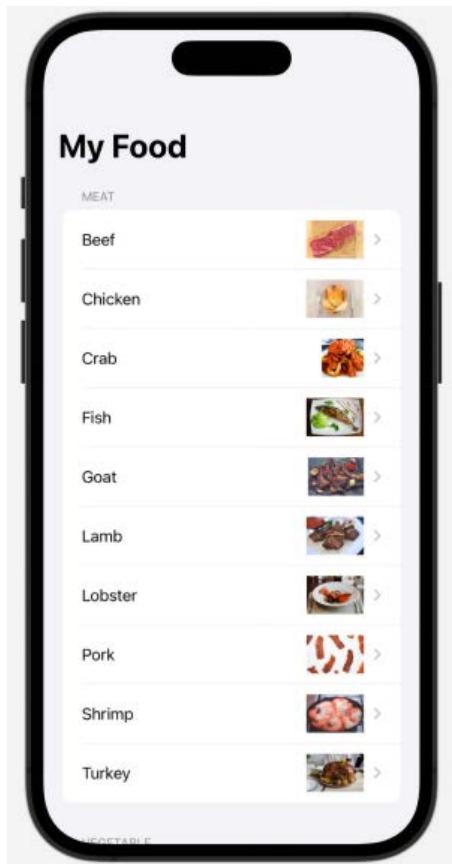


Figure 7.14: ContentView preview

Spend some time interacting with the app and viewing different food types. Notice how, as we use the app, new food types are added to the recently viewed list, at the bottom of any of the detail views. Interact with these entries and navigate to other food types. Finally, test the go-to-root button at the top right of the navigation bar in the detail view.

How it works...

In order to implement data-driven navigation, we need three things: 1) an instance of a `NavigationStack`, which takes a binding to a collection of data that will represent the path or state of the navigation stack, 2) instances of `NavLink`, which will add values to the path, and 3) the use of the `navigationDestination(for:destination:)` modifier to associate a value with a specific view, used as the destination of the navigation value.

In our app, `ContentView` is the root view of our navigation stack, and we define two properties that will be shared across all our views:

```
@StateObject private var navigation = Navigation()  
@StateObject private var storage = Storage(food: Food.samples)
```

The `storage` property of type `Storage` is used to store all our food types and our recently viewed food.

The `navigation` property of type `Navigation` is used to store the state of our navigation stack. Since our app shows different types of food, the state of the navigation stack is defined by the data used to model a food type. In our case, we created the `Food` struct to define a food type, so an array of `Food` instances is a good choice to model our navigation path because it is an ordered collection of elements.

In `Navigation`, we define the property `path` to hold an array of food types that represent the navigation path:

```
@Published var path: [Food] = .init()
```

In the `path` array, the order of the `Food` instances represents the order in which we viewed the food types. Since the navigation stack is driven by the data stored in the `path` variable, to go back to the previously viewed food, we could just drop the last element of the array, and the navigation stack will react to the change and pop to the previous view in the stack. To navigate to the root view, we could simply drop all the elements of the array or initialize the array to an empty array, and the navigation stack will pop to the root view.

We initialized our `NavigationStack` with a binding to the navigation path `$navigation.path` and, as the root view, a `List` view of all our food types. Our list is organized by food category, so we use the `Section` container view to include all the food in each category. Within the section, we use a `ForEach` to iterate through all the different food types in a category, and for each food, we create a `NavLink`, which uses the food instance as a value to add to the navigation path and a `FoodRowView` as the view for the entry in the list.

Finally, we use the `navigationDestination(for:destination:)` modifier to navigate, for values of type `Food`, to views of type `FoodView`, supplying `navigation` and `storage` as shared objects:

```
.navigationDestination(for: Food.self) { food in
    FoodView(food: food)
        .environmentObject(navigation)
        .environmentObject(storage)
}
```

Our `FoodView` object is the detail view for the selected food. The `navigation` object is used to pop to the root view by calling the `popToRoot()` method, triggered by the navigation bar button with the list icon. The `storage` object is used to obtain the recently viewed food array and to add the current food to the list of recent once the view disappears.

See also

- `NavigationStack`: <https://developer.apple.com/documentation/swiftui/navigationstack>
- `NavLink`: <https://developer.apple.com/documentation/swiftui/navigationlink>

Untyped data-driven navigation with NavigationStack

When using a `NavigationStack` with data-driven navigation, we may need to navigate to views driven by different data types. One way to handle this situation would be to implement an `enum` type with associated values of different data types. Then, in the destination closure of the `navigationDestination(for:destination:)` modifier, we could have a `switch` statement over the different associated values, instantiating different types of views, one for each associated value. However, Apple already thought of this situation, and provides `NavigationPath`, a type that holds an array of type-erased data to represent the content of the navigation stack.

Getting ready

Our starting point will be the app from the previous recipe. Either duplicate the folder of the app you created in the previous recipe or download the complete code for the previous recipe from the GitHub repository.

How to do it...

To illustrate the untyped data-drive navigation, we will modify the root view of the app to display two different types of data in a list with two sections. The section at the top of the list will display the different food categories. When we tap on a category, we will navigate to a new view, showing a list of all the food types in that category. The second section of the list will display a list of our favorite food. When we tap on a favorite food, we will navigate to the detailed view of that food. The top section is driven by the `Category` enumeration, and the bottom section is driven by the `Food` struct. The completed app should look like the following screenshot:

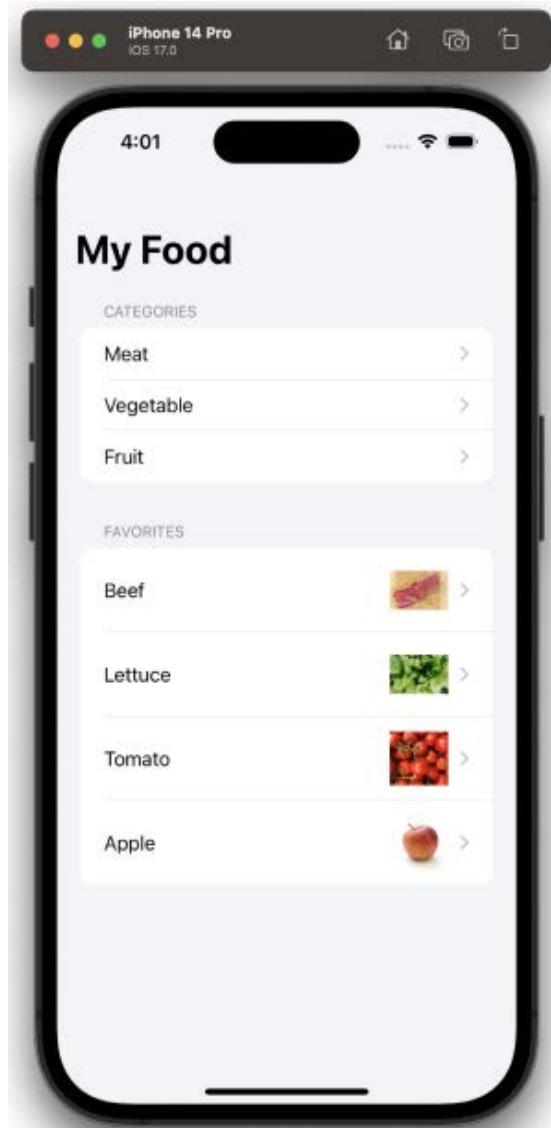


Figure 7.15: Root view of the app running on the iOS simulator

Let's start building the new app starting from the app of the previous recipe. These are the steps to do so:

1. Modify the `Storage` class to have a new array to store the favorite food types. Add the property at the top of the class, under the other two properties, and modify its initializer to include the new property. The final code should be as follows:

```
final class Storage: ObservableObject {  
    @Published var food: [Food]  
    @Published var recents: [Food]
```

```
@Published var favorites: [Food]
init(food: [Food], recents: [Food] = [], favorites: [Food] = []) {
    self.food = food
    self.recents = recents
    self.favorites = favorites
}
...
}
```

- Now, add a couple of functions to manage the favorite food feature. Add these two functions to the class:

```
func isFavorite(_ food: Food) -> Bool {
    favorites.contains(food)
}
func toggleFavorite(_ food: Food) {
    if isFavorite(food) {
        favorites.removeAll { $0.id == food.id }
    } else {
        favorites.append(food)
    }
}
```

The first function will check if a food is a favorite, and the second function will toggle the favorite status of a food.

- Now, let's add a new view to display a list of all the food types in a food category. Go ahead and create a new SwiftUI view named `FoodCategoryView.swift` with the following content:

```
struct FoodCategoryView: View {
    @EnvironmentObject private var storage: Storage
    var category: Category
    var body: some View {
        List(storage.food(in: category)) { food in
            NavigationLink(value: food) {
                FoodRowView(food: food)
            }
        }
        .navigationTitle(category.name)
    }
}
```

```
#Preview {
    NavigationStack {
        FoodCategoryView(category: .fruit)
            .environmentObject(Storage(food: Food.samples))
    }
}
```

If everything went well, the preview should look like this:

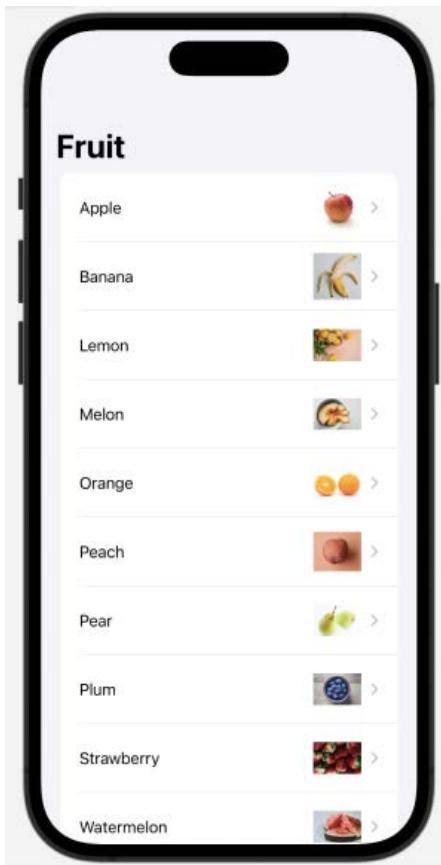


Figure 7.16: Root view of the app running on the iOS simulator

- Now, let's add the functionality to add/remove a favorite food. Go to `FoodView.swift` and add a new button to the toolbar to perform the action. The code for the toolbar should be as follows:

```
.toolbar {
    Button {
        storage.toggleFavorite(food)
```

```
        } label: {
            if storage.isFavorite(food) {
                Image(systemName: "minus.square")
            } else {
                Image(systemName: "plus.square")
            }
        }
    Button {
        navigation.popToRoot()
    } label: {
        Image(systemName: "list.bullet")
    }
}
```

The preview should look like this:

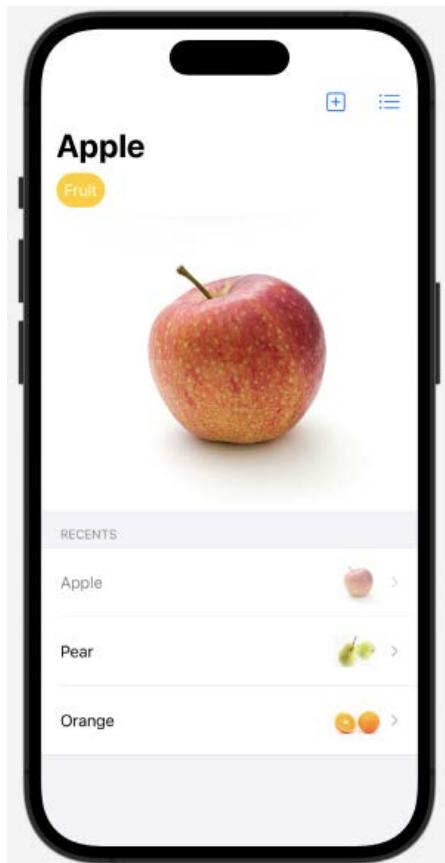


Figure 7.17: FoodView preview with the toggle favorite button

- Now, let's go to `Navigation.swift` and modify the type of the only property in the `Navigation` class:

```
@Published var path: NavigationPath = .init()
```

- Finally, let's modify the root view with our new list with two sections. Go to `ContentView.swift` and replace the `body` variable with the following code:

```
var body: some View {
    NavigationStack(path: $navigation.path) {
        List {
            Section("Categories") {
                ForEach(Category.allCases) { category in
                    NavigationLink(category.name, value: category)
                }
            }
            Section("Favorites") {
                if storage.favorites.isEmpty {
                    Text("No favorites added")
                } else {
                    ForEach(storage.favorites) { food in
                        NavigationLink(value: food) {
                            FoodRowView(food: food)
                        }
                    }
                }
            }
        }
        .navigationTitle("My Food")
        .navigationDestination(for: Category.self) { category in
            FoodCategoryView(category: category)
            .environmentObject(storage)
        }
        .navigationDestination(for: Food.self) { food in
            FoodView(food: food)
            .environmentObject(navigation)
            .environmentObject(storage)
        }
    }
}
```

The preview should look like this:

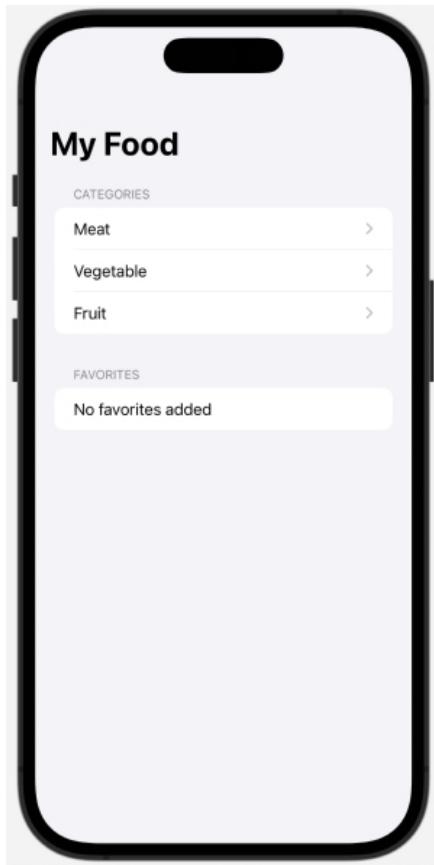


Figure 7.18: ContentView preview with the list with two sections

Spend some time interacting with the app. Navigate to any food, and add or remove the food from the favorite list, go back to the root view, and see how the food appears/disappears from the list of favorite foods. Notice how, in the detail view, the toggle favorite button changes from a plus to a minus symbol, indicating the action of adding or removing the food from the favorite list.

How it works...

We can't use an array when we don't have homogenous data to drive our navigation path. In our root view, the navigation is driven by two different types, the `Category` enum and the `Food` struct. Apple provides the `NavigationPath` struct, as a type-erased list of data representing the content of a navigation stack. When we supply values of different types to `NavigationLink` instances, these values get added to the `NavigationPath` instance and passed as a binding to the `NavigationStack`:

```
NavigationStack(path: $navigation.path)
```

In our root view, the first section uses `NavLink` instances with values of type `Category`:

```
NavLink(category.name, value: category)
```

And the second section uses `NavLink` instances with values of type `Food`:

```
NavLink(value: food) {
    FoodRowView(food: food)
}
```

Now, we need to tell the navigation stack how to link the values of different types to different destination views. A nice feature of the data-driven navigation is that we can have as many `navigationDestination(for:destination:)` modifiers as we need, one for each type of data used to represent the navigation state:

```
.navigationDestination(for: Category.self) { category in
    FoodCategoryView(category: category)
    .environmentObject(storage)
}
.navigationDestination(for: Food.self) { food in
    FoodView(food: food)
    .environmentObject(navigation)
    .environmentObject(storage)
}
```

In our case, we use different types of views for different types of data. `FoodCategoryView` for values of type `Category` and `FoodView` for values of type `Food`. It all looks nice and clean, thanks to the simplicity of the declarative UI.

See also

NavigationPath: <https://developer.apple.com/documentation/swiftui/navigationpath>

Multi-column navigation with `NavigationView`

`NavigationView` is a container view that presents views in two or three columns; selections in leading columns control the presentations in subsequent columns. It was introduced with iOS 16 along with `NavigationStack` and the new `NavLink` functions.

The mail app on the iPad is an example of a three-column layout you may be familiar with.

Getting ready

Our starting point will be the app from the “[Typed data-driven navigation with `NavigationStack`](#)” recipe. Either duplicate the folder of the app you created in that recipe, or download the complete code for that recipe from the GitHub repository.

How to do it...

To show the power of `NavigationSplitView`, we will modify the layout of the app to use a three-column layout for a regular horizontal size class environment, like in the iPad, and fall back to a traditional navigation stack layout for a compact horizontal size class environment, like in the iPhone. The first column of the split view is called the `sidebar` and will include the different categories of food. When we tap on a category, the second column, called `content`, will show a list of all the food in that category. When we tap on a food type, the third column, named `detail1`, will show the detailed view of the food. The detailed view will also include the most recently viewed food at the bottom, with links to navigate to the chosen food. The complete app should look like the following screenshot:

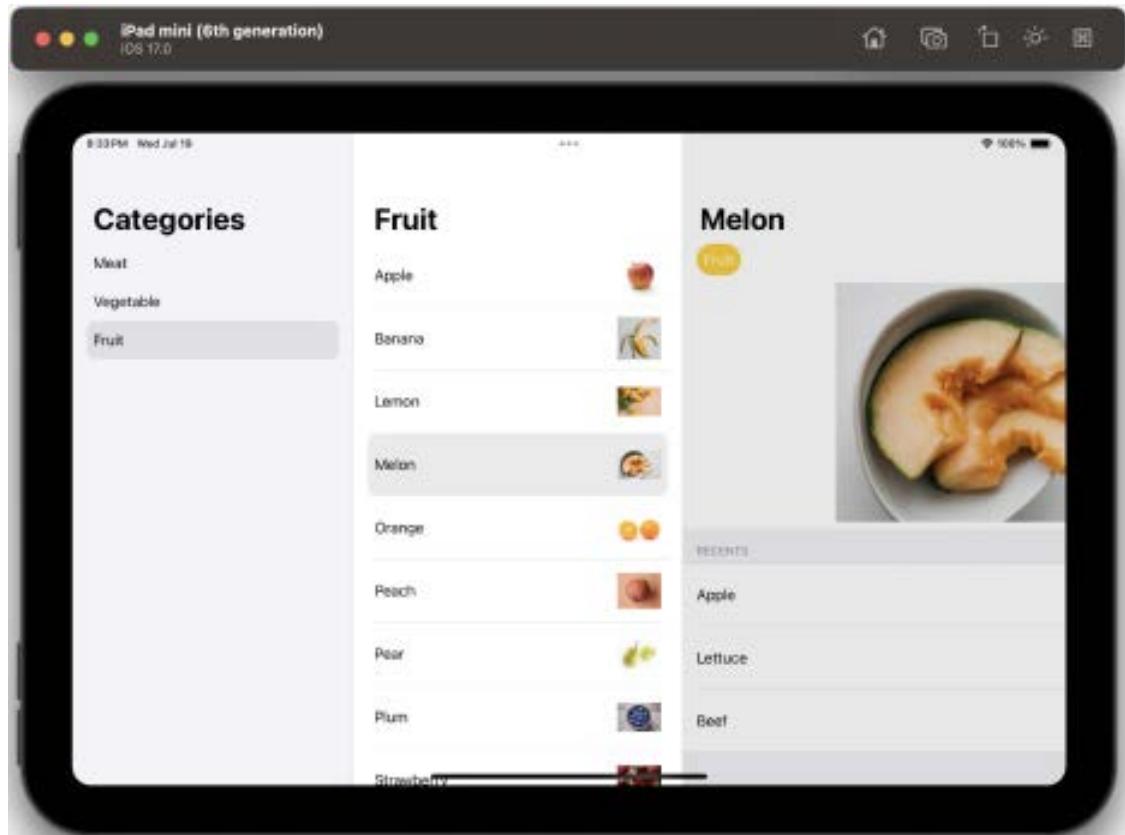


Figure 7.19: The app running on the iPad simulator

Let's build the new app, starting from the app of the aforementioned recipe for typed data-drive navigation. These are the steps:

1. The following files remain unchanged: `ModernNavigationApp`, `Category`, `Food`, `Storage`, `CategoryView`, and `FoodRowView`.

- Now, let's modify the `FoodView` struct, which is our detailed view and will be shown in the third column of the layout. Since the navigation will be controlled by the `NavigationView` container, we don't need the navigation object and the toolbar, so we will delete them. We will add an `onTapGesture` modifier to `FoodRowView`, used in the recent section, so when the user taps on one of the recent food type, we navigate to the selected food type. The code for the struct and the preview should be as follows:

```
struct FoodView: View {  
    @EnvironmentObject private var storage: Storage  
    @Binding var selectedCategory: Category?  
    @Binding var selectedFood: Food?  
  
    var body: some View {  
        if let food = selectedFood {  
            VStack(alignment: .leading) {  
                CategoryView(category: food.category)  
                    .padding(.leading)  
                Image(food.name)  
                    .resizable()  
                    .aspectRatio(contentMode: .fit)  
                    .frame(maxWidth: .infinity, maxHeight: 300)  
                if storage.recents.isEmpty {  
                    Spacer()  
                } else {  
                    List {  
                        Section("Recents") {  
                            ForEach(storage.recents) { recent in  
                                FoodRowView(food: recent)  
                                    .contentShape(Rectangle())  
                                    .onTapGesture {  
                                        if recent != food {  
                                            selectedFood = recent  
                                            selectedCategory = recent.  
                                                category  
                                        }  
                                    }  
                            }  
                        }  
                    }.listStyle(.grouped)  
                }  
            }  
        }  
    }
```

```

        .navigationTitle(food.name)
        .onChange(of: food) { oldValue, newValue in
            storage.addRecent(oldValue)
        }
        .onDisappear() { [selectedFood] in
            storage.addRecent(selectedFood!)
        }
    } else {
        Text("Choose a type of food")
    }
}
}

#Preview {
    let fruits = Array(Food.samples.prefix(3))
    return NavigationStack {
        FoodView(selectedCategory: .constant(.fruit), selectedFood:
            .constant(fruits.first!))
            .environmentObject(Storage(food: Food.samples, recents:
            fruits))
    }
}
}

```

- Now, we need to work on the view for the second column of the layout, which will display all the food types in a category. We will modify the `FoodCategoryView` to account for the case that no category is selected and to change our `selectedFood` when a food type is selected. The contents after the modified code should be as follows:

```

struct FoodCategoryView: View {
    @EnvironmentObject private var storage: Storage
    @Binding var selectedCategory: Category?
    @Binding var selectedFood: Food?
    var body: some View {
        if let selectedCategory {
            List(storage.food(in: selectedCategory), selection:
                $selectedFood) { food in
                NavigationLink(value: food) {
                    FoodRowView(food: food)
                }
            }
            .navigationTitle(selectedCategory.name)
        } else {

```

```
        Text("Choose a category")
    }
}
}

#Preview {
    NavigationStack {
        FoodCategoryView(selectedCategory: .constant(.fruit),
selectedFood: .constant(nil))
            .environmentObject(Storage(food: Food.samples))
    }
}
```

4. And finally, we are going to work on the first column of the layout, which will include the `NavigationSplitView` container, and display a list of the different food categories. Let's modify the `ContentView` struct with the following:

```
struct ContentView: View {
    @StateObject private var storage = Storage(food: Food.samples)
    @State var selectedCategory: Category?
    @State var selectedFood: Food?

    var body: some View {
        NavigationSplitView {
            List(selection: $selectedCategory) {
                ForEach(Category.allCases) { category in
                    NavigationLink(category.name, value: category)
                }
            }
            .navigationTitle("Categories")
        } content: {
            FoodCategoryView(selectedCategory: $selectedCategory,
selectedFood: $selectedFood)
                .environmentObject(storage)
        } detail: {
            FoodView(selectedCategory: $selectedCategory, selectedFood:
$selectedFood)
                .environmentObject(storage)
        }
    }
}
```

5. Choose an iPad simulator and preview the ContentView view in the Xcode canvas, or run the app on an iPad simulator. It should look like this:

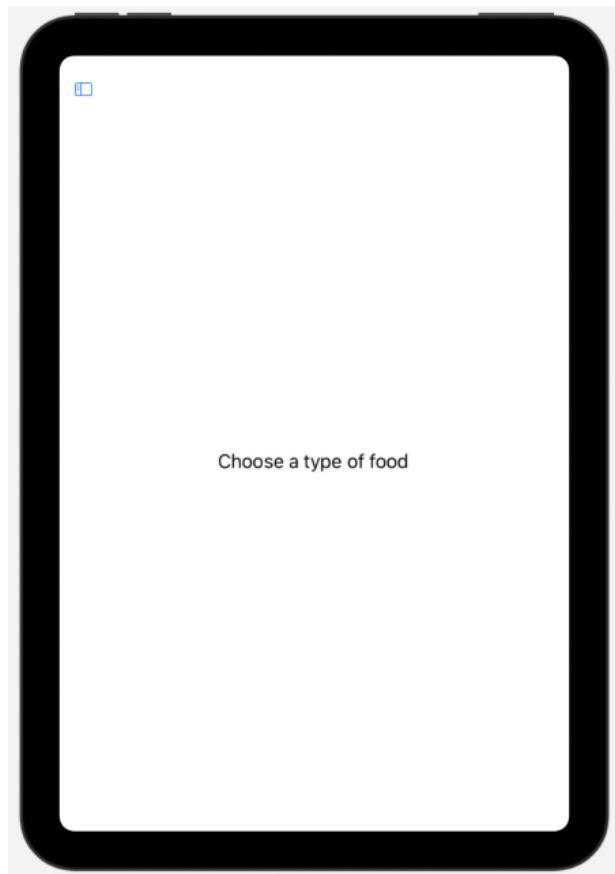


Figure 7.20: The ContentView preview on the iPad

6. Let's preview it in landscape mode too. It should look like this:

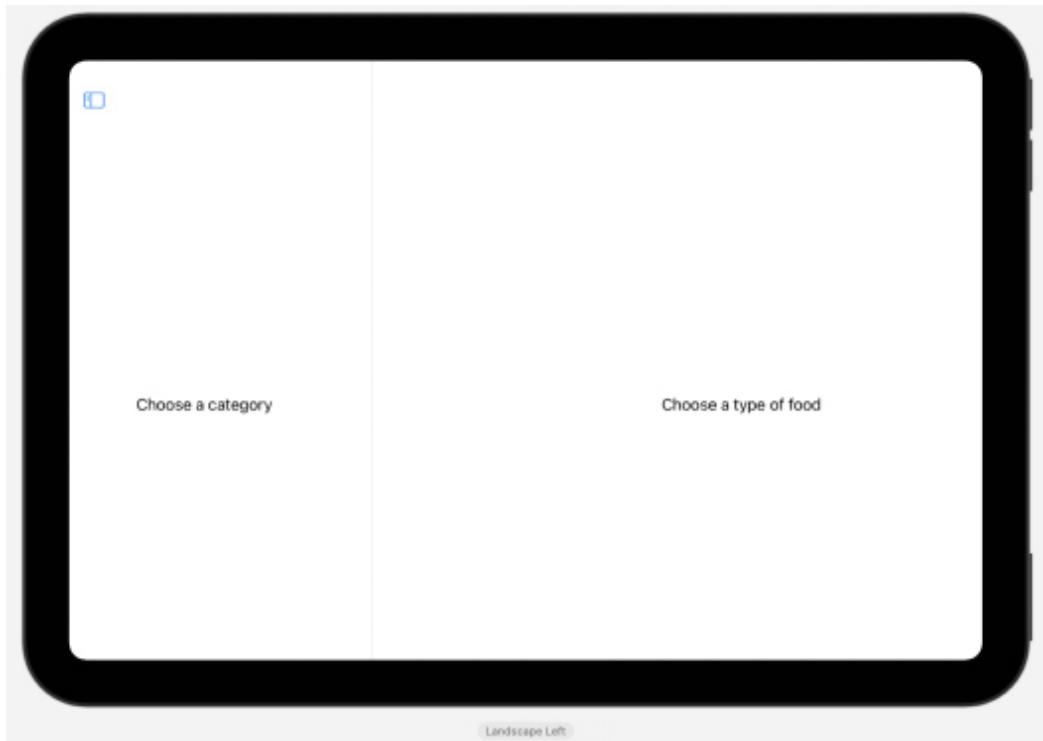


Figure 7.21: The ContentView preview on the iPad on landscape left orientation

You may wonder why the preview looks like that. `NavigationSplitView` changes the layout depending on the environment. In a regular horizontal size class environment like in the iPad in landscape orientation, for a three-column layout, as we have coded in the app, the content and detail columns are shown in a split view. The content column is shown to the leading edge and occupies one-third of the width, and the detail column is shown to the right, occupying two-thirds of the width. When the orientation is portrait, only the detail column is shown on full screen. To reveal the hidden columns, we must tap on the button on the leading edge of the toolbar. Since the sidebar is hidden when we start the app, we have not chosen a category yet, so the app displays empty content and detail views. In our case, we chose to display text instead of an empty view, giving the user an indication of what is going on.

7. Wouldn't it be better to show the sidebar when the user starts the app for the first time? Fortunately, we can use a `NavigationSplitView` initializer that accepts a `columnVisibility` parameter of type `Binding<NavigationSplitViewVisibility>`, which will store the state of the multi-column layout. Let's make a couple of changes to the `ContentView` struct. Add the following variable to the struct, right after the `selectedFood` variable:

```
    @State private var columnVisibility = NavigationSplitViewVisibility.all
```

8. Modify the `NavigationSplitView` initializer. Replace the line:

```
    NavigationSplitView {
```

with the following:

```
    NavigationSplitView(columnVisibility: $columnVisibility) {
```

Run the app, and you'll notice that when the app starts, the three columns show up, making it easier for the user to start interacting with the app.

9. Since the `columnVisibility` variable stores the state of the multi-column layout, when the user interacts with the split view, the variable changes its value to reflect the layout. Similarly, when we programmatically change the value of the variable, the layout will change accordingly. Let's improve the user experience by automatically changing the layout to a two-column layout after the user selects a category. Modify the content closure on the split view with the following:

```
FoodCategoryView(selectedCategory: $selectedCategory, selectedFood:  
$selectedFood)  
    .environmentObject(storage)  
    .onChange(of: selectedCategory, initial: false) {  
        columnVisibility = .doubleColumn  
    }
```

10. Run the app on the iPad in portrait mode. Notice that when you choose a type of food from the content view, the layout is a two-column layout and the detail view is partially covered. We can fix this with the `navigationSplitViewStyle(_:_)` modifier. Add the following line after the closing curly bracket of the detail closure of the split view:

```
    .navigationSplitViewStyle(.balanced)
```

11. Run the app on the iOS simulator using any iPad simulator. Try both orientations and see how the layout behaves. Then, run the app on an iPhone with a big screen, like the iPhone 14 Pro Max. Try the landscape orientation; it should look like the following:

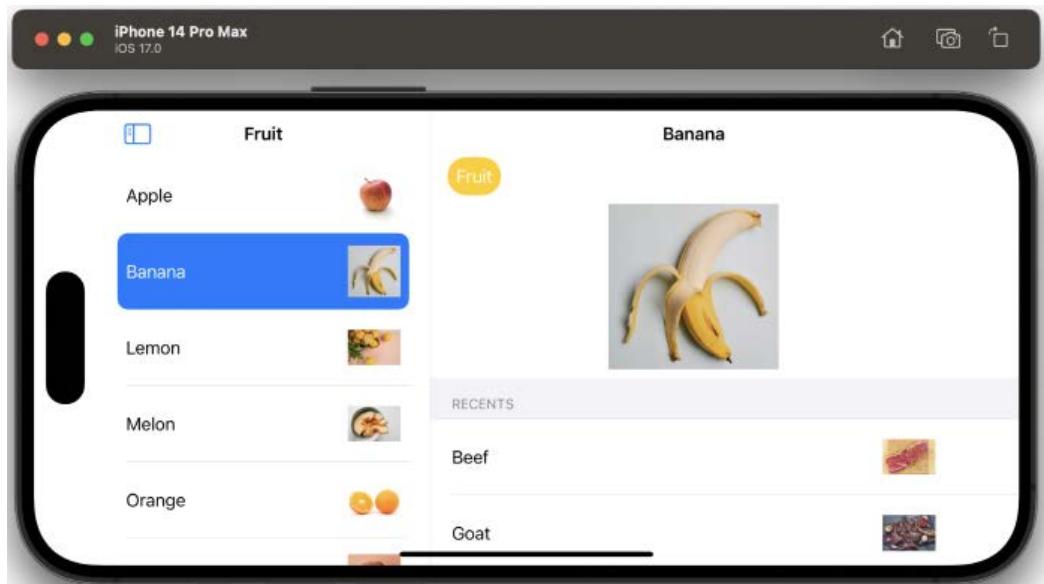


Figure 7.22: The app running on the iPhone 14 Pro Max on a landscape right orientation

How it works...

We can use a `NavigationSplitView` with two or three columns and typically use it as the root view in our app. To create a two-column layout, we use the `init(sidebar:detail:)` initializer. When we select an item in the sidebar, the detail view updates accordingly. In our app, we have implemented the three-column layout using the `init(sidebar:content:detail:)` initializer. As we saw, the selection in the first column (sidebar) updates the second column (detail), and the selection in the second column updates the third column (detail).

For the sidebar or first column, we used a list:

```
List(selection: $selectedCategory)
```

with a selection binding:

```
@State var selectedCategory: Category?
```

and then we used the selection binding in our content or second column:

```
FoodCategoryView(selectedCategory: $selectedCategory, selectedFood:  
$selectedFood)
```

This also uses another selection binding:

```
@State var selectedFood: Food?
```

which is passed to the detail or third column:

```
FoodView(selectedCategory: $selectedCategory, selectedFood: $selectedFood)
```

Since the selection bindings are of an optional type, we need to account for a `nil` value in our views. Both the `FoodCategoryView` and `FoodView` start with a conditional, and if the selection received is `nil`, they display a `Text` view with an informational message, instead of the intended content. `FoodView` would only need the `selectedFood` binding if it wasn't for the recent list feature. When we tap on a food type on the recent list, we change the `selectedFood` variable, so we display the details for the new food type. Since the newly selected food could have a different category, we update the category too, keeping the detail and content views in sync:

```
selectedFood = recent
selectedCategory = recent.category
```

Another interesting feature to mention on the `FoodView` struct is how we add a food type to the recent list:

```
.onChange(of: food) { oldValue, newValue in
    storage.addRecent(oldValue)
}
.onDisappear() { [selectedFood] in
    storage.addRecent(selectedFood!)
}
```

The `onDisappear(perform:)` modifier is used to add the selected food to the recent list. The view will disappear in a navigation stack environment when the user dismisses the view, like when our app runs on an iPhone in portrait mode. The `onChange(of:initial:_:)` modifier, introduced in iOS 17, is used to trigger the addition of the food to the recent list when the value changes. This happens in a split view environment, like when our app runs on the iPad, because the view does not disappear but gets updated. The reason why we have two modifiers is that we support both the iPhone and the iPad with the same code.

The last feature of `NavigationSplitView` worth mentioning is that, in a compact horizontal size class, like in the iPhone or the iPad in Slide Over mode, the split view collapses all its columns into a stack and shows the last column, which displays valuable information. We even have a parameter to choose which columns show by default when collapsed.

See also

NavigationSplitView: <https://developer.apple.com/documentation/swiftui/navigationsplitview>

Navigating between multiple views with TabView

TabView is another navigation container that switches between multiple child views, using a bar with buttons commonly placed at the bottom of the screen. When the user taps on one of the buttons in the tab bar, the view associated with that tab replaces the previously selected view. A typical example of using this navigation container is the Music app on iOS.

Tab views are usually used as the root view of an app, and they are useful to show unrelated content in each tab. For related content, we would use a navigation stack or a navigation split view, as we saw in the previous recipes.

Getting ready

Create a new SwiftUI iOS app named `UsingTabViews`.

How to do it...

We will create an app with a TabView with two child views. One will display the made-up best games of 2021, while the other will display various currencies used around the world. The steps are given here:

1. Create a new SwiftUI view file named `HomeView`, and update `HomeView.swift` to display a list of games from a string array named `games`:

```
struct HomeView: View {
    let games = ["Doom", "Final F", "Cyberpunk", "avengers", "animal
trivia", "sudoku", "snakes and ladders", "Power rangers", "ultimate
frisbee", "football", "soccer", "much more"]
    var body: some View {
        NavigationStack {
            List {
                ForEach(games, id: \.self){ game in
                    Text(game).padding()
                }
            }.navigationTitle("Best Games for 2021")
        }
    }
}
```

The HomeView preview should look like this:

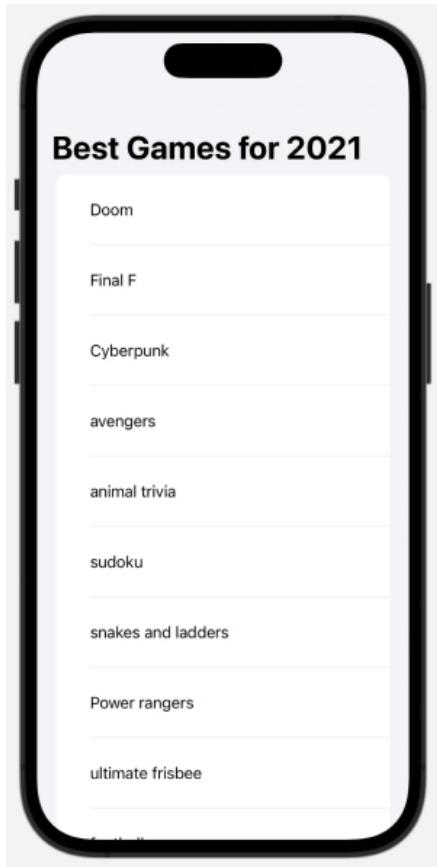


Figure 7.23: HomeView preview

2. We would also like to display a list of currencies. Each currency should have an ID and name and conform to the `Identifiable` protocol. Let's create a `Currency.swift` file that describes the fields we should expect from a currency. In `Currency.swift`, define a `Currency` model and create an array of currencies:

```
struct Currency: Identifiable {  
    let id = UUID()  
    var name:String  
    var image:String  
}
```

```
extension Currency {  
    static 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")  
    ]  
}
```

3. Create a new SwiftUI view file named `CurrenciesView`, and update it as follows:

```
struct CurrenciesView: View {  
    var body: some View {  
        NavigationStack {  
            VStack {  
                ForEach(Currency.currencies) { currency in  
                    HStack {  
                        Text(currency.name)  
                        Spacer()  
                        Image(systemName: currency.image)  
                    }  
                    .font(Font.system(size: 32, design: .default))  
                    .padding()  
                }.navigationTitle("Currencies")  
            }  
        }  
    }  
}
```

At this point, the CurrenciesView preview should look like this:

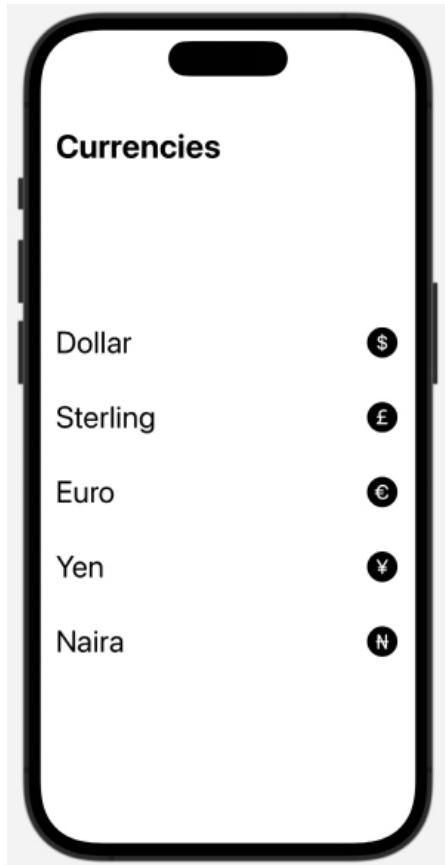


Figure 7.24: CurrenciesView preview

4. Now that we have set up the child views of our tab view, 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:

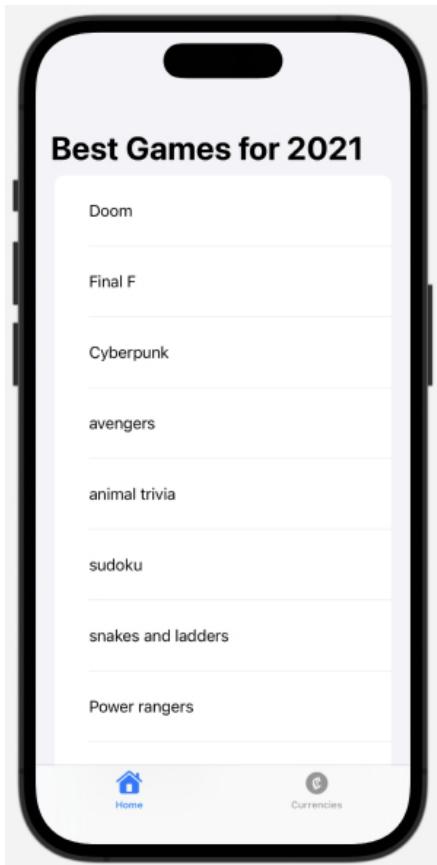


Figure 7.25: ContentView preview

Now, run the app and tap on the **Home** or **Currencies** tab to switch between them.

How it works...

In its most basic form, a **TabView** struct can hold multiple child 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 to display 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 viewed in *Chapter 1, Using the Basic SwiftUI Views and Controls*, and *Chapter 2, Displaying Scrollable Content 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()` modifier to the `TabView` as follows:

```
TabView {
}.tabViewStyle(.page)
```



Important Note

If you are using `TabView` and `NavigationStack` at the same time, make sure that `TabView` is the parent and `NavigationStack` is nested within `TabView`.

See also

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

Programmatically switching tabs on a TabView

In the preceding recipe, we learned how to switch between tabs by tapping 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 selectedTab = 0
    ...
}
```

2. Replace the Text view in the `body` variable with a TabView, with two tab items:

```
var body: some View {
    TabView(selection: $selectedTab) {
        Text("Left Tab. Tap to switch to right")
            .onTapGesture {
                selectedTab = 1
            }
            .tabItem {
                Label("Left", systemImage: "l.circle.fill")
            }
            .tag(0)
        Text("Right Tab. Tap to switch to left")
            .onTapGesture {
                selectedTab = 0
            }
            .tabItem {
                Label("Right", systemImage: "r.circle.fill")
            }
            .tag(1)
    }
}
```

The resulting preview should look like this:



Figure 7.26: *TabViewWithGestures* preview

Run the app in a live preview, and click on the text to switch between tabs. Note that the tab items at the bottom of the screen indicate which tab you're currently viewing.

How it works...

To programmatically switch between tabs, we first create a state variable, `selectedTab`, that stores the value of the currently selected tab. We then add a `selection` argument to our `TabView` struct that binds to the `selectedTab` variable. We then add a `.tag(_:) modifier` to each of the tabs with a value that uniquely identifies each tab. Finally, we use the `onTapGesture(count:perform:)` modifier to change the value of our `selectedTab` variable.

When the `selectedTab` is `0`, our left tab becomes the active tab displayed on the screen, and when `selectedTab` changes to `1`, our right tab becomes active.

See also

Apple's documentation on the `onTapGesture(count:perform:)` modifier: [https://developer.apple.com/documentation/swiftui/view/ontapgesture\(count:perform:\)](https://developer.apple.com/documentation/swiftui/view/ontapgesture(count:perform:))

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

<https://packt.link/swiftUI>



8

Drawing with SwiftUI

One of the strongest points of SwiftUI is that all the components are uniform and 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 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

Technical requirements

The code in this chapter is based on Xcode 15.0 and iOS 17.0. You can download and install the latest version of Xcode from the App Store. You'll also need to be running macOS Ventura (13.5) or newer.

Simply search for Xcode in the App Store and select and download the latest version. Launch Xcode, and follow any additional installation instructions that your system may prompt you with. Once Xcode has fully launched, you're ready to go.

All the code examples for this chapter can be found on GitHub at <https://github.com/PacktPublishing/SwiftUI-Cookbook-3rd-Edition/tree/main/Chapter08-Drawing-with-SwiftUI>.

Using SwiftUI's built-in shapes

SwiftUI provides six different basic shapes:

- Rectangle
- RoundedRectangle
- UnevenRoundedRectangle
- 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 built-in shapes to discover their potential in 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)  
    UnevenRoundedRectangle(  
        cornerRadii: RectangleCornerRadii(  
            topLeading: 50,  
            bottomLeading: 35,  
            bottomTrailing: 20,  
            topTrailing: 0  
        ),  
        style: .circular  
    )  
        .fill(.teal)  
    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 8.1: SwiftUI's built-in shapes

How it works...

The code is self-explanatory, thanks to SwiftUI and its declarative syntax, 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 8.2: 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.

Secondly, 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.

Thirdly, the rectangle uses the `.stroke()` modifier, which creates a border centered in the frame, as shown in the following diagram:



Figure 8.3: `stroke` applied to a shape

Conversely, the circle uses the `.strokeBorder()` modifier, which creates a border contained inside the frame:

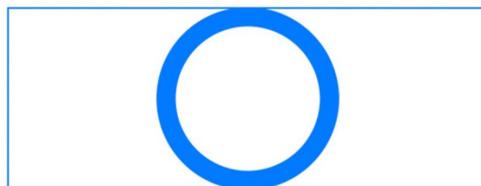


Figure 8.4: `strokeBorder` applied to a shape

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:

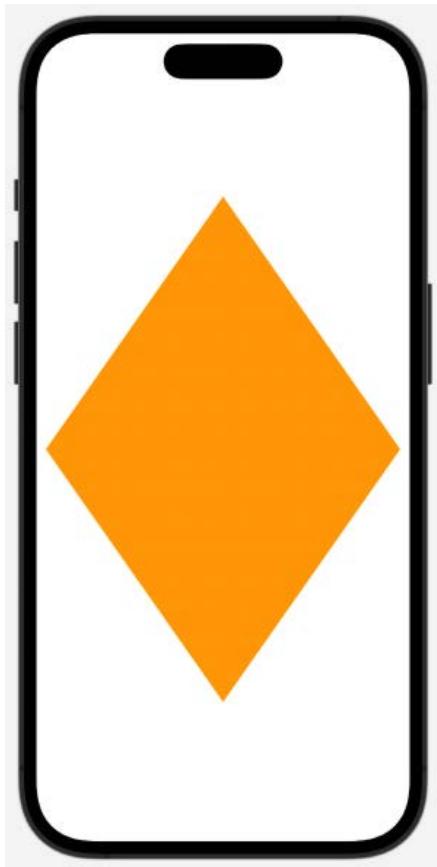


Figure 8.5: A rhombus as a custom SwiftUI shape

How it works...

To create a custom shape in SwiftUI, our struct must conform to the `Shape` protocol, whose only required function without a default implementation is a `path(in:)`. This function provides a rectangle that represents the frame, where the shape will be rendered as a parameter and return 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. Logo is a language designed to introduce kids to the world of programming. In this programming language, you are in charge of guiding a turtle that draws a line on the screen. The turtle is controlled by 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 coordinate system of SwiftUI has its origin in the top-left corner of the screen, with positive x -axis values increasing from left to right, and positive y -axis values increasing from top to bottom.

The following diagram shows the coordinate system and the shortcuts defined in iOS for the most important points:

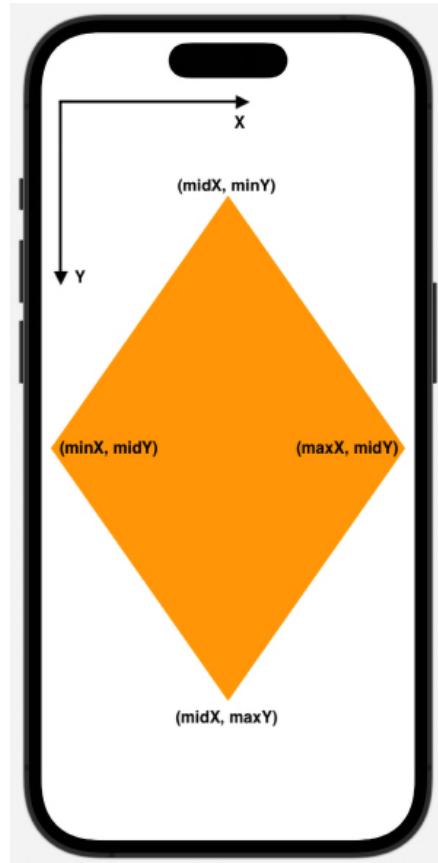


Figure 8.6: SwiftUI coordinate system

As mentioned in the introduction section, 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 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 this recipe, we will 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 that 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.

`CGRect` is a structure inherited from the Core Graphics framework and has been present in iOS since the beginning. It is used to describe a rectangle in the drawing space. You define a rectangle with the coordinates of its origin and its two dimensions, width, and height.

Let's get started with our custom shape. Perform the following steps:

1. 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  
    }  
}
```


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 drawn using our code:

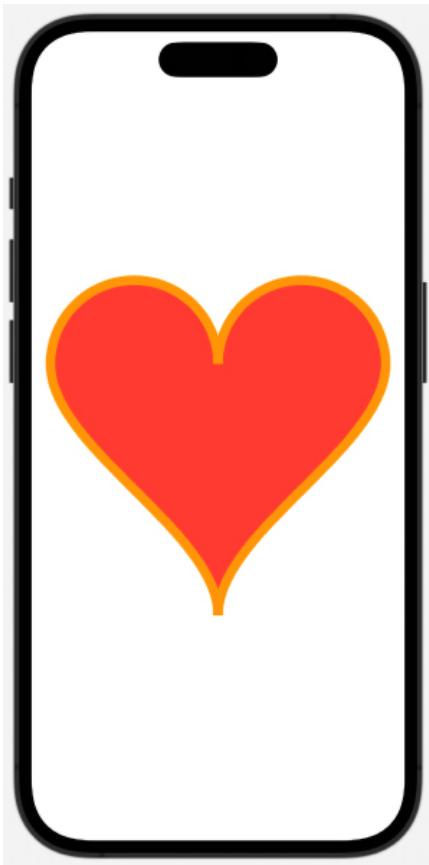


Figure 8.7: Custom heart-shaped component

How it works...

Like in the *Drawing a custom shape* recipe, we initially set the starting point to be the bottom tip of the shape, and then add 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:

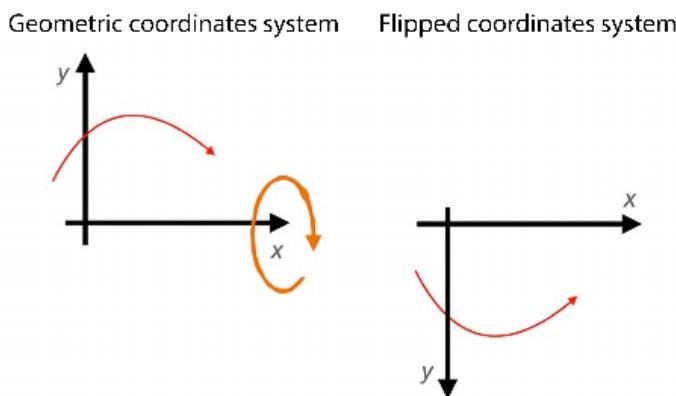


Figure 8.8: The clockwise direction in the geometric and flipped-coordinate systems

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, we 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, storing the lines we will be drawing:

```
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  
        }  
        let lastLine = lines[lastIdx]  
        lastLine.points.append(position)  
        lines[lastIdx] = lastLine  
    }  
}
```

```
        }
        lines[lastIdx].points.append(position)
    }
})
```

With the gesture code, the app is ready, and we can now draw our masterpieces:



Figure 8.9: 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 achieve more 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.

- Creating an array of points for each finger movement: The `DrawGesture` object listens for changes in the touches and movement. If the translation of the movement is zero, this means that the touch has started, so a new line must be added to the model. If the translation is greater than zero, the finger is moved from the previous position. Then, a new point must be added to the last point of the line.

There's more...

We have just seen how simple it is to draw in a `GraphicsContext` object so that 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 will implement a progress ring like those on the Apple Watch, and we will 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 will 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)

    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 `ProgressRingsView` view, with the definition of some common constant properties and some other 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 {
        .init(lineWidth: ringWidth,
              lineCap: .round,
              lineJoin: .round)
    }

    var progressExternal: Double
    var progressCentral: Double
    var progressInternal: Double

    var body: some View {
    }
}
```

3. The `body` area of the View presents the three rings, one on top of the 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)
}
```

- Finally, replace the contents of the body property of `ContentView` with an instance of `ProgressRingsView` and three `Slider` views:

```
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)
        }
    }
}
```

By 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 8.10: 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 @State in the ContentView, because they are mutated in the same component by the sliders, which have bindings to these properties.

When we change the value of a slider, the mutation is reflected in the @State property, and the ContentView is rendered again by SwiftUI. The values of the sliders are passed to ProgressRingsView, which gets redrawn again.

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 will 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 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 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()
    }
}
```

6. 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
        }
    }
}
```

7. 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))
```

8. 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:

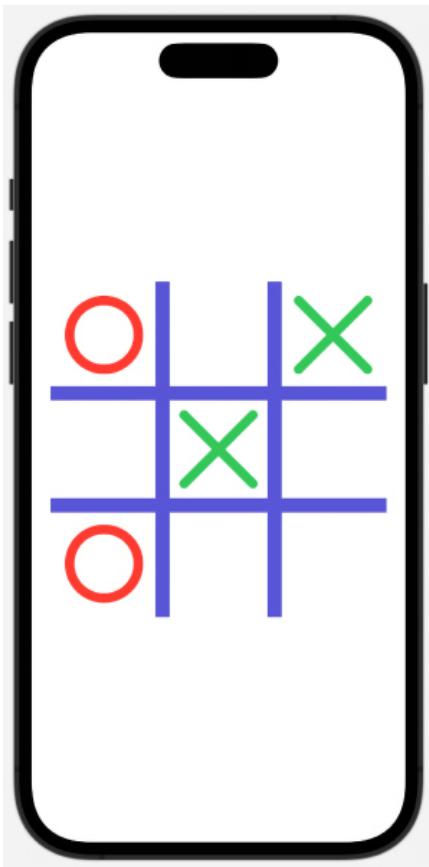


Figure 8.11: Playing Tic-Tac-Toe with SwiftUI

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 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 that 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 a `Rectangle` is used, but we could use `Circle`, a `Capsule`, or even a custom shape.

There's more...

The game is almost complete, but it is missing 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 four different types of gradients:

- Linear gradients
- Radial gradients
- Elliptical 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 view to select the type of gradient we want to draw on the screen.

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 third gradient we will create is the elliptical gradient, where the colors transition through concentric ellipses:

```
struct EllipticalGradientView: View {  
    var body: some View {  
        ZStack {  
            EllipticalGradient(  
                gradient: Gradient(colors:  
                    [.orange, .green,  
                     .blue, .black]),  
                center: .center,  
                startRadiusFraction: 0,  
                endRadiusFraction: 0.75)  
            Text("Elliptical Gradient")  
                .bigLight()  
        }  
    }  
}
```

5. 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()
}
}
}
```

6. 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()
        case 2:
            EllipticalGradientView()
        default:
            AngularGradientView()
        }
    }
}

var body: some View {
}
```

7. 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("Elliptical").tag(2)
        Text("Angular").tag(3)
    }
    .pickerStyle(SegmentedPickerStyle())
    .padding(.horizontal, 32)
}
}
```

Preview the app in the live view of the canvas, and select different segments of the picker to 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 8.12: Preview of the elliptical gradient

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 four different ways of representing a gradient: linear, radial, elliptical, and angular. For each of them, we defined a list of the colors that each one 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.

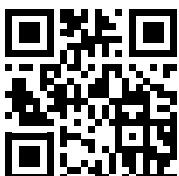
Each different gradient type offers further possibilities for customization:

- 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.
- For the `elliptical` gradient, you can set the start and end radii as a fraction of the unit circle.
- 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.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

<https://packt.link/swiftUI>



9

Animating with SwiftUI

SwiftUI has introduced a new way of describing UI elements and components and 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 SwiftUI 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 a `@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 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 about 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 an animation view modifier 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
- Chained animations with `PhaseAnimator`

- Custom animations with `KeyframeAnimator`
- Creating custom view transitions
- Creating a hero view transition with `.matchedGeometryEffect`
- Implementing a stretchable header in SwiftUI
- Implementing a swipeable stack of cards in SwiftUI

Technical requirements

The code in this chapter is based on Xcode 15.0 and iOS 17.0. You can download and install the latest version of Xcode from the App Store. You'll also need to be running macOS Ventura (13.5) or newer.

Simply search for Xcode in the App Store and select and download the latest version. Launch Xcode and follow any additional installation instructions that your system may prompt you with. Once Xcode has fully launched, you're ready to go.

All the code examples for this chapter can be found on GitHub at <https://github.com/PacktPublishing/SwiftUI-Cookbook-3rd-Edition/tree/main/Chapter09-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

Create a new SwiftUI iOS app named `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 a confirmation dialog to choose the animation for the red circle, while the animation for the blue circle is always the default one. The default animation prior to iOS 17 was `.easeInOut`, and in iOS 17, it was changed to `.spring`. We will select the animation for the red circle with a button that presents a confirmation dialog 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 the possible animations and then select one:

```
struct AnimationType: Hashable {  
    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 = AnimationType.all.first!
    @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. Add a button to trigger the animation, a Text view to display the currently chosen animation type, and a button to toggle the visibility of the confirmation dialog, which will display the list of available animation types to choose from:

```
struct ContentView: View {
    //...
    var body: some View {
        VStack(spacing: 12) {
            GeometryReader { geometry in
                //...
            }
            Button {
                onTop.toggle()
            } label: {
                Text("Animate")
            }
            Text("Current animation: \(type.name)")
            .confirmationDialog("Animations", isPresented:
                $showSelection) {
                    ForEach(AnimationType.all, id: \.self) { type in
                        Button(type.name) {
                            self.type = type
                        }
                    }
                } message: {
                    Text("Animations")
                }
            }
        }
    }
}
```

If everything goes well, the preview should look like the following:

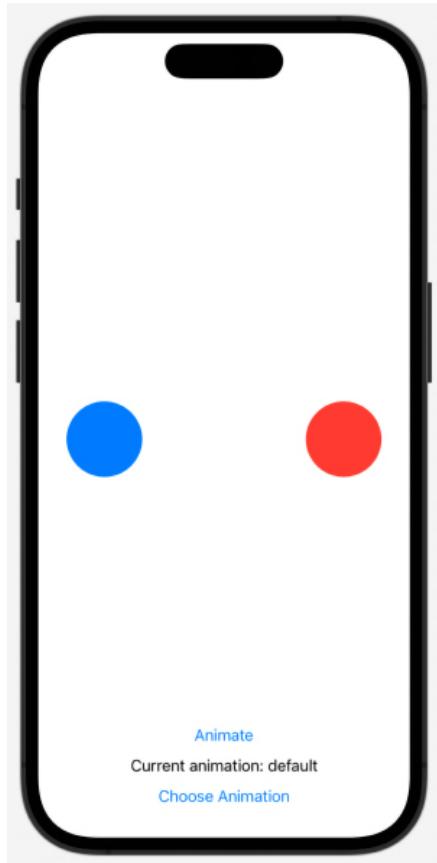


Figure 9.1: Basic animations in SwiftUI

Run the app or use the live preview in the canvas and observe how the different animations run compared to each other.

How it works...

As you can see, the result is 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(_:value:)` view modifier, SwiftUI will then apply that animation.

Note that we must pass the change of data parameter to the `.animation(_:value:)` 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 animation 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 types of animations.

There's more...

Changing the offset is just one of the possible changes in 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 the site is not SwiftUI-related, so the names are slightly different.

Transforming shapes

In the recipe for creating basic animations, you'll find that SwiftUI allows you to animate changes in fundamental attributes like position, colour, size, and more. 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. However, if we run the app, we can see that changes in the triangle shape are not animated.

This is because SwiftUI doesn't know which data it has to animate.

3. To instruct SwiftUI what to animate, we must add a property called `animatableData` to the `Triangle` struct. This property is a requirement of the `Animatable` protocol, which `Shape` inherits from, and indicates which property needs to be animated:

```
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:

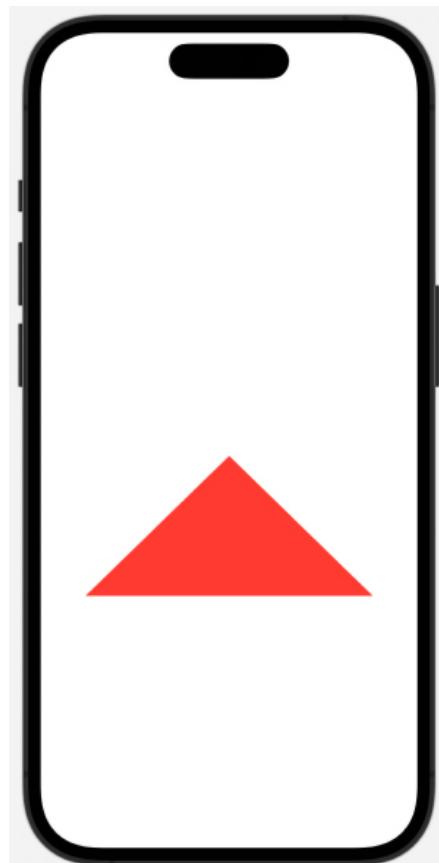


Figure 9.2: Shape animated transformation

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. In our case, our `Triangle` struct conforms to `Shape`, which inherits from `Animatable`.

To inspect the behavior, let's add `print` to the setter:

```
set {
    multiplier = newValue
    print("value: \(multiplier)")
}
```

Use the live preview in the canvas and make sure that the console is shown and set to `Previews`. You should see the `print` statements as you click on the triangle. The result should be 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 9.3: Intermediate value of `animatableData`

For every step, SwiftUI calculates the value for `animatableData`, sets it in the shape, and then renders the shape.

The `Triangle` struct conforms to `Shape`, which already conforms to `Animatable`, so the only thing we have to do is define the `animatableData` property and specify 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 simple recipe, where we create a banner view that can be animated when we tap on a button:

1. Define a new view named `BannerView`. The code should be as follows:

```
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)  
            .foregroundStyle(.white)  
            .background(Color.green)  
            .cornerRadius(10)  
            .offset(y: show ?
```

```
        -UIScreen.main.bounds.height / 3 :  
        -UIScreen.main.bounds.height)  
    .animation(  
        .interpolatingSpring(  
            duration: 0.888,  
            bounce: 0.646  
        ),  
        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:

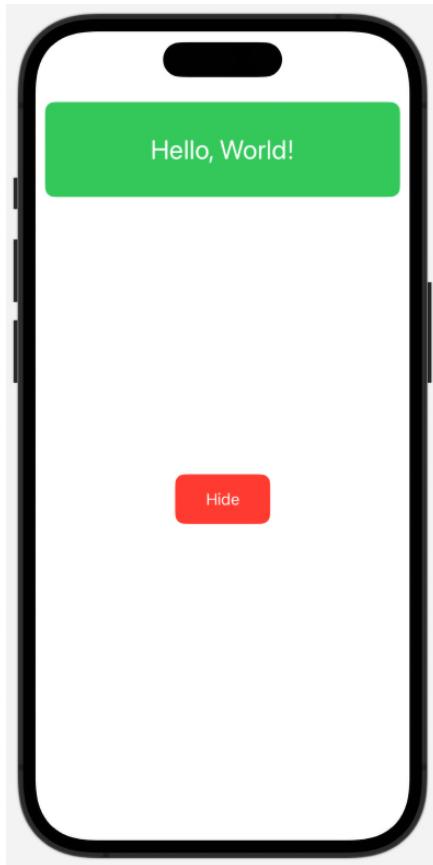


Figure 9.4: A bouncing banner view

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 animation curve used is an interpolating spring curve, which uses a damped spring model to produce values in the range $[0,1]$ that are then used to interpolate the range of values of the animated property. We can use several initializers, but we used the easiest to understand, which has a few parameters:

- **duration:** This represents the perceptual duration of the animation.
- **bounce:** This is how bouncy the spring should be. A value of 0 means no bounce at all, and a value of 1.0 means an undamped oscillation, bouncing forever.
- **initialVelocity:** This is the velocity when the animation starts. If omitted, the default value is 0.

Feel free to change the parameters and see how the animation changes. For example, change the bounce value to 1.0 and observe how the banner oscillates forever.

Applying a delay to an animation view modifier to create a sequence of animations

Until the introduction of iOS 17, it was not possible to create a chained sequence of different animations. If we wanted to accomplish this, we could do it by implementing several animations, each one with a different delay, so the animations would occur in a staggered way, one immediately after another.

Since the introduction of SwiftUI with iOS 13, there have been two ways of defining an animation:

- Using the `.animation(_:value:)` view modifier
- Using the `withAnimation(_::_:)` function

In this recipe, we'll see how to use the `.animation(_:value:)` view modifier, and we'll cover the `withAnimation(_::_:)` function in the next recipe.

In subsequent recipes, we will cover the new, powerful animation APIs introduced in iOS 17. The reason for this and the next recipe is that they show the way to perform chained animations since the introduction of SwiftUI, and you should be familiar with it since you could come across these techniques when dealing with legacy code.

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 achieve 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)
                .foregroundStyle(.white)
                .padding()
                .background(.green)
                .cornerRadius(5)
        }
    }
}
```

Running the app, we can see the animations joining together as if they were a single animation:

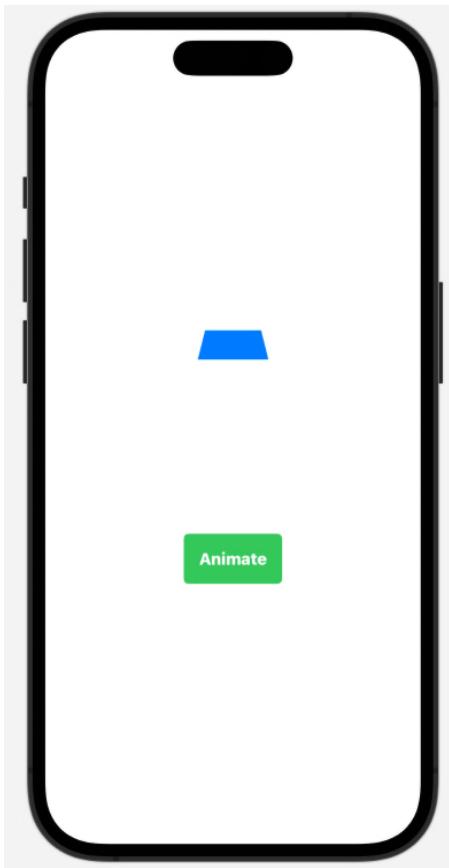


Figure 9.5: Preview of the state of the animation after tapping on the button

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.

Even though the technique is simple, it doesn't work for all the view modifiers. So, play around and test them to find the correct sequence of animations.

If you tap the button while the views are animating, the value of the `initialState` variable will change, and it will trigger a new computation of the `body` property. The practical effect is that the animations will reverse course and the view will go back to the original state, without completing the full cycle of animations.

Applying a delay to a `withAnimation(_:_:)` function to create a sequence of animations

As mentioned in the previous recipe, since the introduction of SwiftUI with iOS 13, there have been 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 `withAnimation` function 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  
            .foregroundStyle(foreground)  
            .frame(width: 200)  
            .padding()  
            .background(background)  
            .cornerRadius(cornerRadius)  
    }  
}
```

2. For our code to be less verbose, and following Apple's recommendations, we use `modifier` in an extension to the `Text` struct, 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` view:

```
struct ContentView: View {
    @State var hideFirst = true
    @State var hideSecond = true
    @State var hideThird = true
    var body: some View {
        VStack {
            Spacer()
            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()
        VStack(spacing: 30) {
            //...
        }
        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 `Text` views changes in sequence, simulating a single animation:

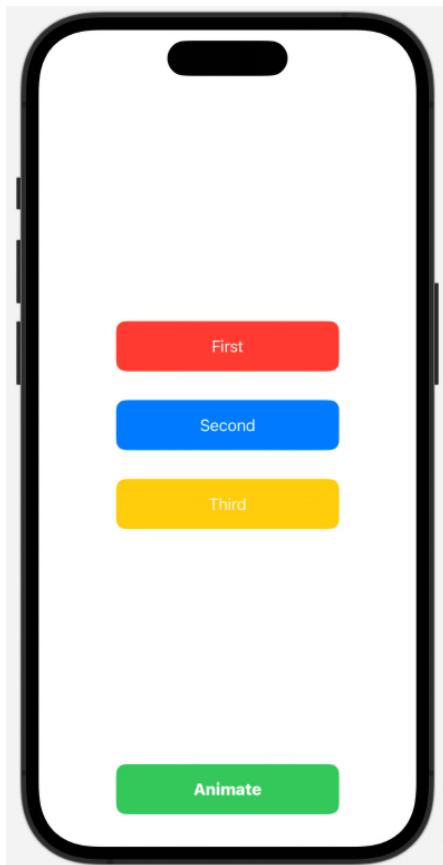


Figure 9.6: Delayed animations to create a sequential appearance of Text views

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. In iOS 17, Apple introduced new animation APIs that allow us to chain animations together, as we will see later in this chapter.

There's more...

In iOS 17, Apple introduced `withAnimation(_:_completionCriteria:_:_completion:)`, a modified version of the `withAnimation(_:_)` function used in the recipe. This new function accepts a completion callback, which will be called after the animation completes. We could have written our chained animations using the completion callback instead of introducing delays. Using the new function, the alternative code for the `Animate` button would be the following:

```
Button {  
    withAnimation(Animation.easeInOut) {  
        hideFirst.toggle()  
    } completion: {  
        withAnimation(Animation.easeInOut) {  
            hideSecond.toggle()  
        } completion: {  
            withAnimation(Animation.easeInOut) {  
                hideThird.toggle()  
            }  
        }  
    }  
} label: {  
    Text("Animate")  
    .fontWeight(.heavy)  
    .styled(color: .green)  
}
```

Applying multiple animations to a view

SwiftUI allows us to animate multiple view properties 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 properties 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(duration:)` curve for the former and `.linear(duration:)` 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)
                    .foregroundStyle(.white)
                    .padding()
                    .background(.green)
                    .cornerRadius(5)
            }
        }
    }
}
```

2. We can now add the first set of changes to the `Rectangle` instance, with an `.easeInOut(duration:)` animation:

```
//...
Rectangle()
    .fill(initialState ? .blue : .red)
    .cornerRadius(initialState ? 50 : 0)
    .offset(y: initialState ? 0 : -200)
    .rotation3DEffect(
```

```
        initialState ? .zero : .degrees(45),  
        axis: (x: 1, y: 0, z: 0)  
    )  
.animation(.easeInOut(duration: 2), value: initialState)
```

- Finally, we add the second set of changes, with a `.linear(duration:)` animation:

```
Rectangle()  
//...  
.scaleEffect(initialState ? 1 : 0.8)  
.rotationEffect(initialState ? .zero : .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 together:



Figure 9.7: Multiple animations on the same view

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.

Chained animations with PhaseAnimator

In iOS 17, Apple introduced new, powerful animation APIs. With the `PhaseAnimator` container view, we can finally apply a series of animations chained together. `PhaseAnimator` animates its content automatically by cycling through a collection of values, named phases, each defining a step within the animation. We can also apply the `phaseAnimator(_:content:animation:)` view modifier to obtain the same result.

We will see how to use the `PhaseAnimator` container and the `phaseAnimator(_:content:animation:)` view modifier in this recipe.

Getting ready

No external resources are needed, so let's just create a SwiftUI project in Xcode called `ChainedAnimations`.

How to do it...

To illustrate how `PhaseAnimator` and `phaseAnimator(_:content:animation:)` work, we will use some images and animate some of the properties:

1. In `ContentView`, modify the struct with the following:

```
struct ContentView: View {  
    @State var trigger = false  
    var body: some View {  
        VStack(spacing: 60) {  
            HStack(spacing: 60) {  
                Image(systemName: "sun.max")  
                    .resizable()  
                    .foregroundStyle(.yellow)  
                    .frame(width: 80, height: 80)  
            ZStack {  
                Image(systemName: "soccerball")  
                    .resizable()  
                    .frame(width: 120, height: 120)  
                Image(systemName: "baseball")  
                    .resizable()  
                    .frame(width: 20, height: 20)  
                    .foregroundStyle(.brown)  
                    .offset(x: 0, y: -70)  
            }  
        }  
        Button {  
            trigger.toggle()  
        } label: {
```

```
        Text("Animate")
            .fontWeight(.heavy)
            .foregroundStyle(.white)
            .padding()
            .background(.green)
            .cornerRadius(5.0)
    }
}
}
}
```

This is what our view looks like so far:

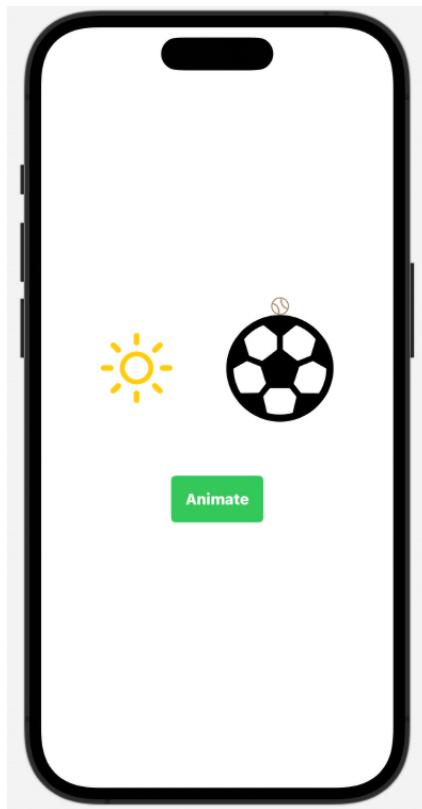


Figure 9.8: ContentView preview

2. Now let's animate the image of the sun so it appears from the top of the screen, comes down to the center of the screen, and goes back to the top and disappears. We will also want to change the scale accordingly and introduce a rotation movement. Modify the Image view representing the sun with the following:

```
Image(systemName: "sun.max")
```

```
.resizable()
.foregroundColor(.yellow)
.frame(width: 80, height: 80)
.phaseAnimator(Array(-30...30), trigger: trigger) { content, phase in
    content
        .rotationEffect(.degrees(24 * Double(phase)))
        .scaleEffect(1.0 - CGFloat(abs(phase)) / 30.0)
        .offset(y: CGFloat(-12 * abs(phase)))
} animation: { phase in
    .linear(duration: 0.04 )
}
```

As soon as we finish typing the code, the sun will not be visible on the screen. But when we tap the **Animate** button, we will observe a nice animation.

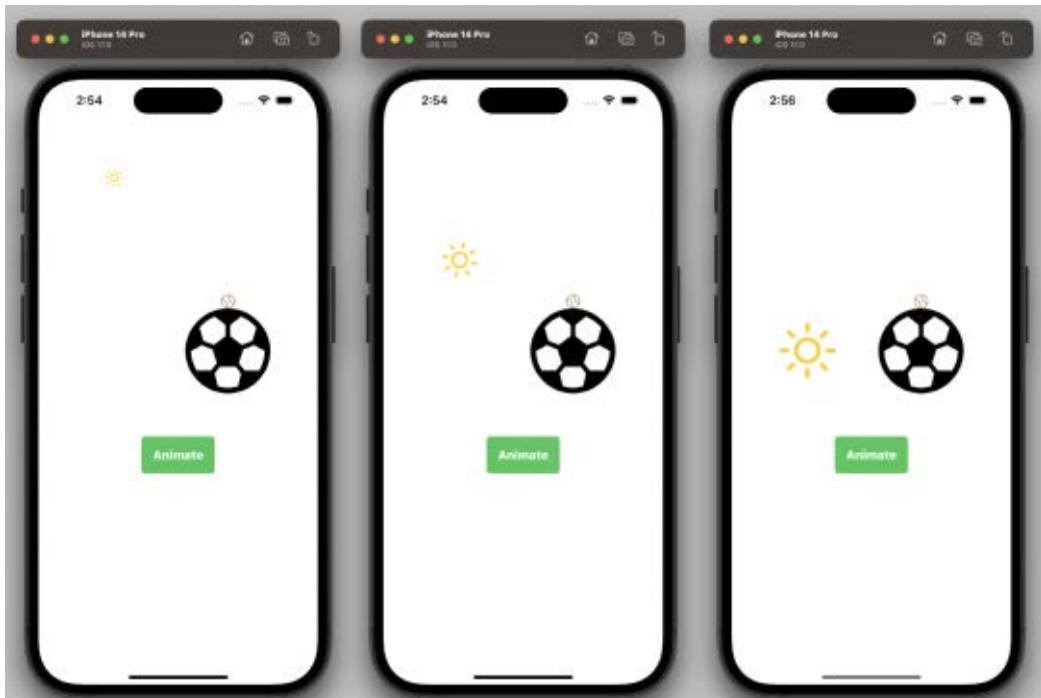


Figure 9.9: Animation of the sun image

- Now we will proceed to animate the soccer ball and the baseball ball. We want to make the soccer ball spin clockwise and make the baseball ball rotate around the soccer ball in the opposite direction. We will enclose the ZStack with the images of the two balls in a PhaseAnimator container and add rotationEffect(_:anchor:) modifiers to both images. Replace the ZStack created in *step 1* of this recipe with the following code:

```
PhaseAnimator([360, 0], trigger: trigger) { phase in
```

```
ZStack {  
    Image(systemName: "soccerball")  
        .resizable()  
        .frame(width: 120, height: 120)  
        .rotationEffect(.degrees(Double(phase - 360)))  
    Image(systemName: "baseball")  
        .resizable()  
        .frame(width: 20, height: 20)  
        .foregroundStyle(.brown)  
        .offset(x: 0, y: -70)  
        .rotationEffect(.degrees(Double(-phase)))  
}  
} animation: { phase in  
    if phase == 0 {  
        .none  
    } else {  
        .linear(duration: 4.8)  
    }  
}
```

4. Tap the **Animate** button and observe the final animation of the three images.

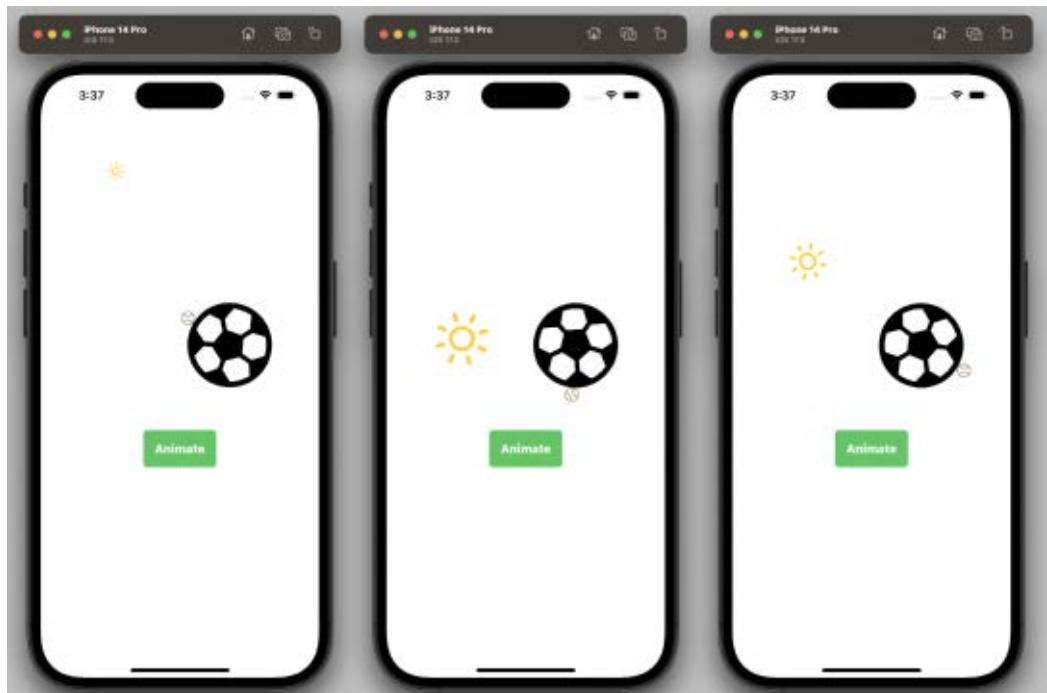


Figure 9.10: Animation of the three images

How it works...

With the use of the phase animation features introduced in iOS 17, there is no need to stagger animations and spend time calculating delays and timing to make it look right.

In our sun image animation, we animated the rotation, scale, and vertical offset. We used the `phaseAnimator(_:content:animation:)` modifier with 61 different phases, from -30 to 30, in one-unit increment. We chose these values so we can easily calculate the rotation angles, the scale, and the vertical offset values. We also used the `animation` closure to choose a particular duration for our animation phases; otherwise, the default would have been used.

In the ball animations, we used a `PhaseAnimator` container. The use of the container allows us to apply different animation effects to the views inside the container. Notice how the two images have different rotation angles depending on the phase. We can even choose to animate different properties for each of the views inside the container. If we had chosen to apply the `phaseAnimator(_:content:animation:)` modifier to the `ZStack`, we would not have implemented independent rotation effects for the two balls.

Custom animations with KeyframeAnimator

The most powerful animation feature introduced in iOS 17 is keyframe animation. We can implement keyframe animations with the `KeyframeAnimator` container. This container allows us to perform animations frame by frame. It is a very advanced feature and requires a deep understanding of animation techniques or animation software to generate the keyframe values for us. This is outside the scope of this book.

Alternatively, we can use the `keyframeAnimator(initialValue:repeating:content:keyframes:)` view modifier to implement keyframe animations.

In this recipe, we will show how to use the `KeyframeAnimator` container.

Getting ready

No external resources are needed, so let's just create a SwiftUI project in Xcode called `CustomAnimations`.

How to do it...

To illustrate how the `KeyframeAnimator` container works, we will include a couple of images and we will animate some of the properties:

1. In `ContentView`, modify the struct with the following:

```
struct ContentView: View {  
    @State var animate = false  
    var body: some View {  
        VStack(spacing: 60) {  
            ZStack {  
                Ellipse()  
                    .foregroundStyle(.gray)  
                    .frame(width: 60, height: 20)  
                    .offset(y: 30)  
                Image(systemName: "basketball.fill")  
                    .resizable()  
                    .foregroundStyle(.orange)  
                    .background(.black)  
                    .clipShape(Circle())  
                    .frame(width: 60, height: 60)  
            }  
            Button {  
                animate.toggle()  
            } label: {  
                Text(animate ? "Stop" : "Animate")  
                    .fontWeight(.heavy)  
                    .foregroundStyle(.white)  
                    .padding(.vertical, 10)  
                    .frame(width: 100)  
                    .background(animate ? .red : .green)  
                    .cornerRadius(5.0)  
            }  
        }  
    }  
}
```

This is what our view looks like so far:

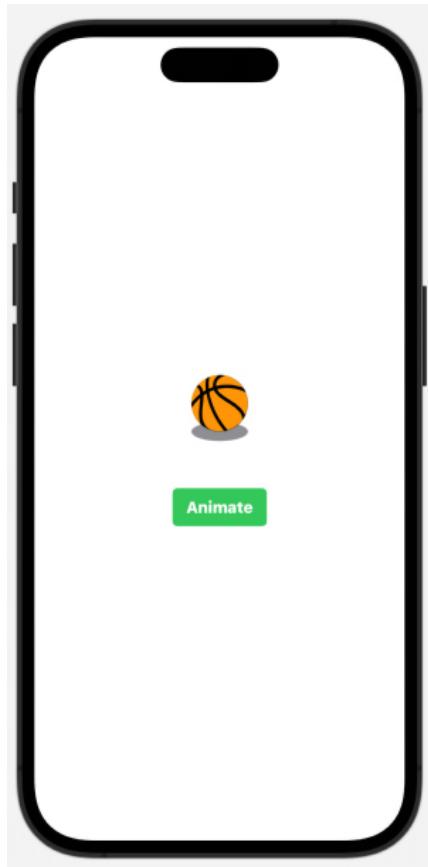


Figure 9.11: ContentView preview

2. We want to animate the image of the basketball so it bounces up and down, and at the same time, modify the shape of the shadow so it looks more realistic. We will create a custom struct to hold the values of the properties we want to animate. Go ahead and create the `AnimationValues` struct above the declaration of `ContentView`:

```
private struct AnimationValues {  
    var verticalOffset = 0.0  
    var verticalStretch = 0.8  
    var shadowScale = 1.0  
}  
  
struct ContentView: View { ... }
```

3. Now let's animate the images. We will enclose the ZStack with the images in a KeyframeAnimator container. Replace the ZStack with the following:

```
KeyframeAnimator(initialValue: AnimationValues(), repeating: animate) {
    value in
        ZStack {
            Ellipse()
                .foregroundStyle(.gray)
                .frame(width: 60 * value.shadowScale, height: 20 * value.
shadowScale)
                .offset(y: 30)
            Image(systemName: "basketball.fill")
                .resizable()
                .foregroundStyle(.orange)
                .background(.black)
                .clipShape(Circle())
                .frame(width: 60, height: 60)
                .scaleEffect(y: value.verticalStretch, anchor: .bottom)
                .offset(y: value.verticalOffset)
        }
    } keyframes: { value in
        KeyframeTrack(\.verticalStretch) {
            SpringKeyframe(0.8, duration: 0.15)
            CubicKeyframe(1.0, duration: 0.2)
            CubicKeyframe(1.2, duration: 0.8)
            CubicKeyframe(1.0, duration: 0.2)
            SpringKeyframe(0.8, duration: 0.15)
        }
        KeyframeTrack(\.verticalOffset) {
            CubicKeyframe(0, duration: 0.2)
            CubicKeyframe(-120, duration: 0.4)
            CubicKeyframe(-120, duration: 0.4)
            CubicKeyframe(0, duration: 0.5)
        }
        KeyframeTrack(\.shadowScale) {
            CubicKeyframe(1, duration: 0.2)
            CubicKeyframe(0.6, duration: 0.4)
            CubicKeyframe(0.6, duration: 0.4)
            CubicKeyframe(1, duration: 0.5)
        }
    }
}
```

- Once we finish typing the code, run the app on the simulator of your choice and try the animation by tapping on the **Animate** button.

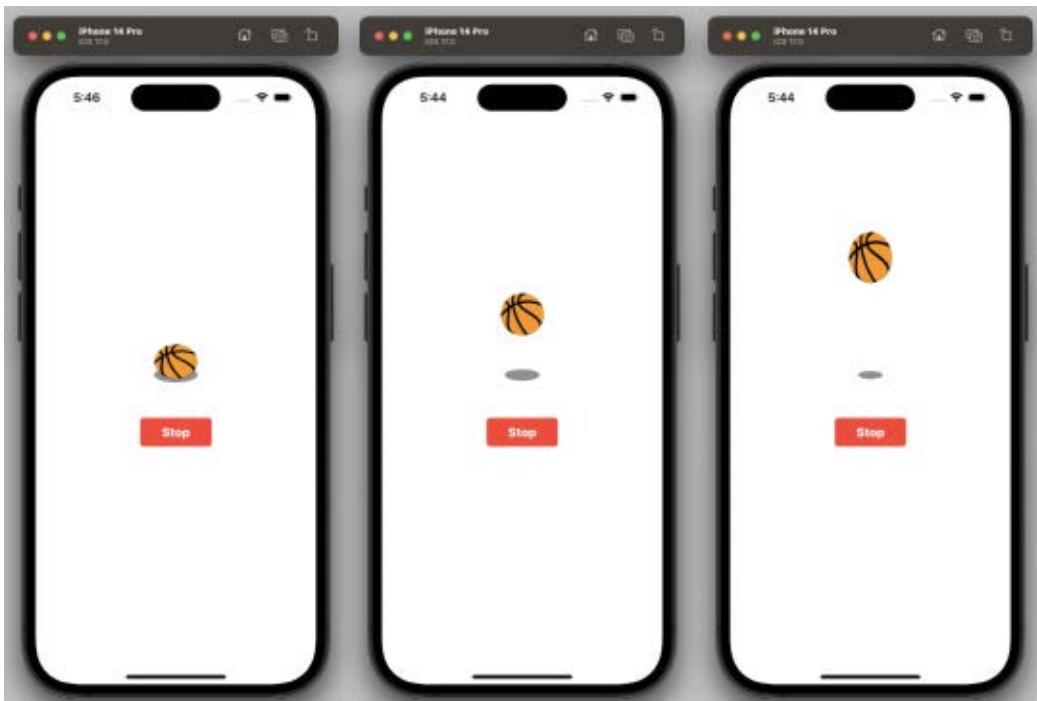


Figure 9.12: Animation of the basketball ball

How it works...

With the `KeyframeAnimator` container, we can animate different animatable properties totally independently of each other. We start by defining a starting value for all the properties we want to animate. Then, for each of the animated properties, we define a keyframe. A keyframe consists of three parameters: a value, a time interval that represents the time it takes to transition to that value, and the type of animation curve used to animate the changes in the value.

For example, for the scale of the ellipse representing the shadow of the ball, we used four keyframes:

```
KeyframeTrack(\.shadowScale) {  
    CubicKeyframe(1, duration: 0.2)  
    CubicKeyframe(0.6, duration: 0.4)  
    CubicKeyframe(0.6, duration: 0.4)  
    CubicKeyframe(1, duration: 0.5)  
}
```

The initial value of the scale is `1.0`, then it stays at `1.0` for 0.2 seconds, changes to `0.6` for 0.4 seconds, stays at `0.6` for 0.4 more seconds, transitions to `1.0` for 0.5 seconds, and the sequence repeats over and over. We used cubic keyframes, which offer a smooth transition between values.

We can animate the different properties independently of each other; we control the values, the timing, and the type of curve used by SwiftUI to interpolate the values between keyframes.

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 repository at <https://github.com/PacktPublishing/SwiftUI-Cookbook-3rd-Edition/tree/main/Resources/Chapter09/recipe9>, 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`.

Now, create a new SwiftUI project in Xcode called `CustomViewTransition`, and copy the `image1.jpg` and `image2.jpg` images to the Assets catalog:

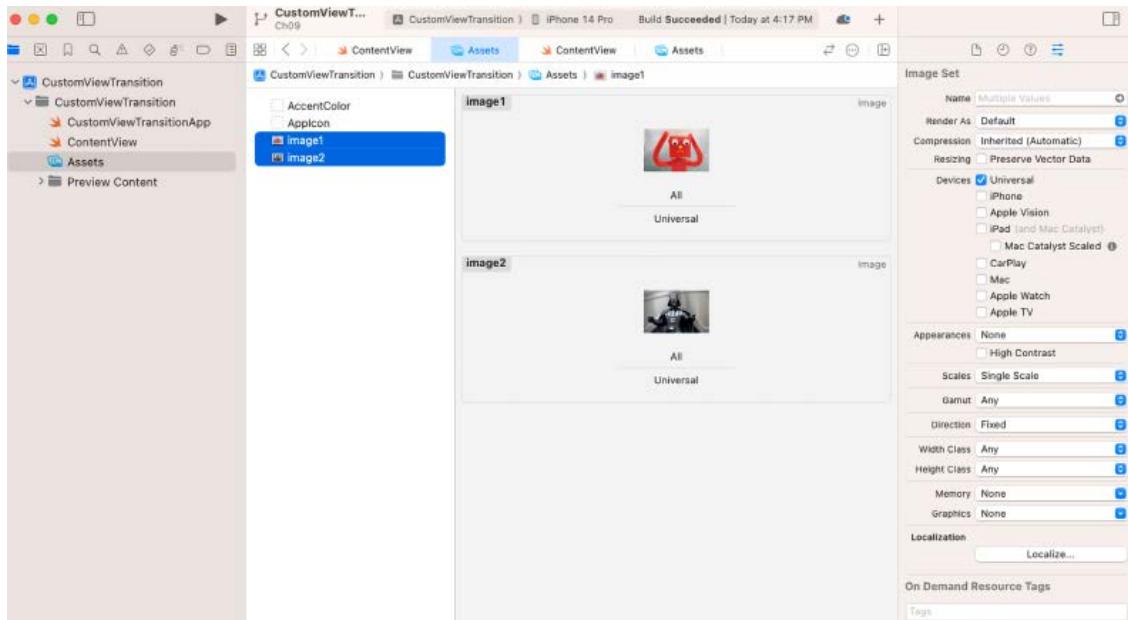


Figure 9.13: Adding the images to the Assets catalog

How to do it...

We are going to implement a simple app where we switch two views when we tap on a button. For simplicity, the two custom views are just wrappers around an `Image` view, but this will work for any kind of view:

1. First, we will implement the two views that wrap around an `Image` view:

```
extension Image {  
    func custom() -> some View {  
        self  
            .resizable()  
            .aspectRatio(contentMode: .fit)  
            .cornerRadius(20)  
            .shadow(radius: 10)  
    }  
}  
  
struct FirstView: View {  
    var body: some View {  
        Image(.image1)  
            .custom()  
    }  
}  
  
struct SecondView: View {  
    var body: some View {  
        Image(.image2)  
            .custom()  
    }  
}
```

2. Now put the custom views in the `ContentView`, selecting which one to present depending on a `Bool` variable that will be toggled by a button:

```
struct ContentView: View {  
    @State var showFirst = true  
    var body: some View {  
        VStack(spacing: 24) {  
            if showFirst {  
                FirstView()  
            } else {  
                SecondView()  
            }  
        }  
    }  
}
```

```
        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 {
//...
    FirstView()
        .transition(.moveScaleAndFade)
//...
    SecondView()
        .transition(.moveScaleAndFade)
//...
}
```

Running the app now, we can see that the views appear by scaling up and moving from the left, and disappear by scaling down and moving to the top:

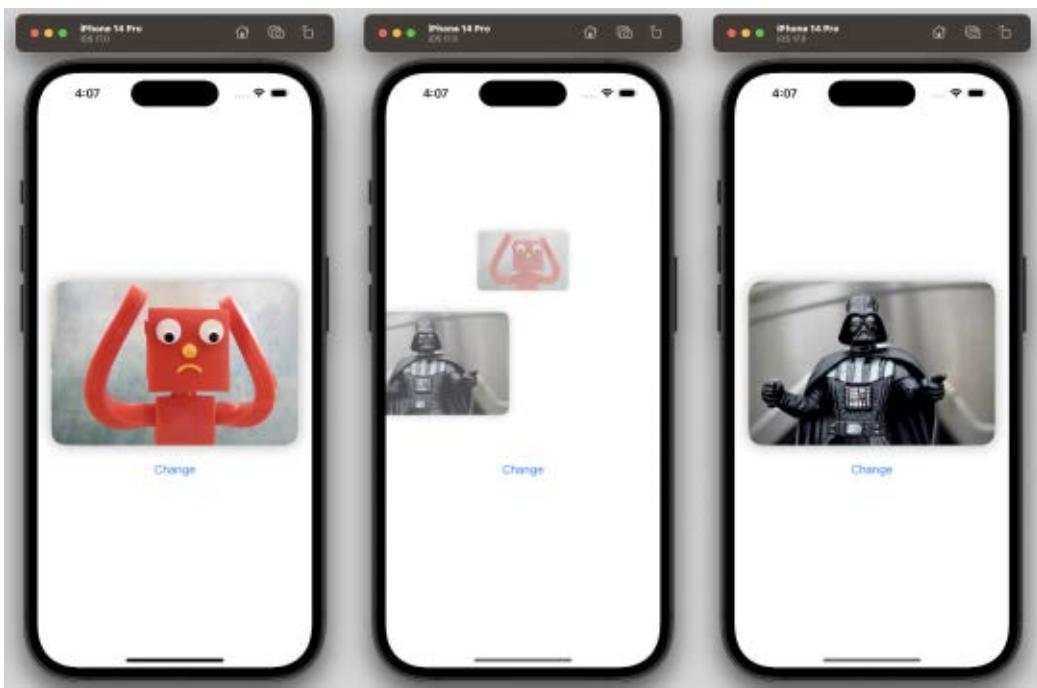


Figure 9.14: Custom view transitions

How it works...

A transition is basically a list of transformations that can be applied when a view appears or disappears.

Using the `.combined(with:)` function, a transition can be combined with others to create more sophisticated animations.

In our example, we used an `asymmetric(insertion:removal:)` transition function to create our custom transition. It means that the removal will look different than the insertion. But of course, we could also have a symmetric transition. Try returning the `insertion` or `removal` transition in `static var moveScaleAndFade` 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 for sale, each product also has a thumbnail to show the product. Selecting a product thumbnail flies 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 view modifier, `.matchedGeometryEffect(id:in:properties:anchor:isSource:)`, which makes it very easy to implement a hero view animation with little effort.

Getting ready

This recipe uses a few images, courtesy of *Pixabay* (<https://pixabay.com>).

You can find the images in the GitHub repository at <https://github.com/PacktPublishing/SwiftUI-Cookbook-3rd-Edition/tree/main/Resources/Chapter09/recipe10>, 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:

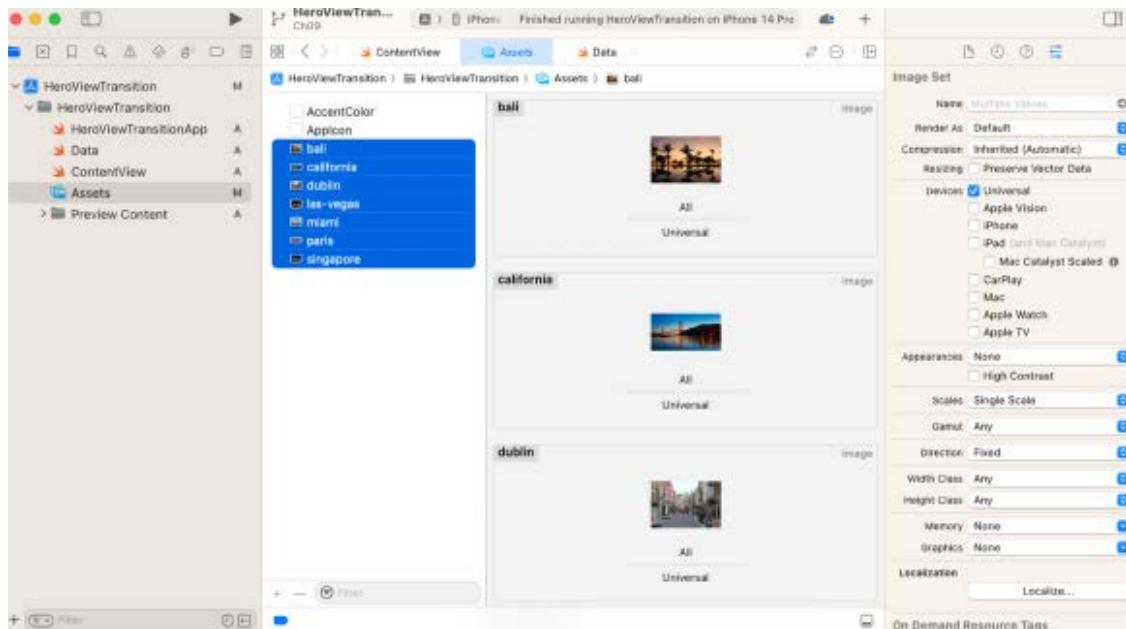


Figure 9.15: Adding the images to the Assets catalog

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 details 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:

1. First, we implement the model for our holiday destination app. Go to `ContentView.swift` and implement the following at the top of the file, after the `import` statement:

```
struct Item: Identifiable {  
    let id = UUID()  
    let image: String  
    let title: String  
    let details: String  
}
```

2. With this model, we can create an array of items:

```
Extension Item {  
    static 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"),  
        //...  
    ]  
}
```

3. 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-3rd-Edition/tree/main/Resources/Chapter09/recipe10/Data.swift>. Copy this file into your project.
4. Below the `Item` struct closing bracket, 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 {
```

```
        }
    }
}
```

5. 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(item.data) { item in
                }
            }
        .padding(.all, 20)
    }
}
```

6. Inside the `ForEach` loop body, we present an `Image` component:

```
ForEach(item.data) { item in
    Image(item.image)
        .resizable()
        .aspectRatio(contentMode: .fill)
        .cornerRadius(10)
        .shadow(radius: 5)
}
```

7. 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, isSource:
!showDetail)
```

8. Finally, an `.onTapGesture` callback will select the item to open and display the detail page:

```
Image(item.image)
//...
.onTapGesture {
    selectedItem = item
    withAnimation {
        showDetail.toggle()
    }
}
```

9. Let's move on to implementing the details page, where we define `DestinationDetailView` and its properties. After the closing bracket of `DestinationListView`, add:

```
struct DestinationDetailView: View {  
    var selectedItem: Item  
    @Binding var showDetail: Bool  
    let animation: Namespace.ID  
    var body: some View {  
    }  
}
```

10. 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))  
    }  
}
```

11. Let's start with the `Button` component, where we simply dismiss the page by toggling the `showDetail` flag:

```
Button {  
    withAnimation {  
        showDetail.toggle()  
    }  
} label: {  
    Image(systemName: "xmark")  
        .foregroundStyle(.white)  
        .padding()  
        .background(.black.opacity(0.8))  
        .clipShape(Circle())  
}  
.padding(.trailing,10)
```

12. Add the following code to the `VStack` component, which contains the information about 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)
```

13. Finally, apply the `.matchedGeometryEffect` modifier to the `Image` component on the details page:

```
Image(selectedItem.image)
    .resizable()
    .aspectRatio(contentMode: .fit)
    .matchedGeometryEffect(id: selectedItem.image, in: animation,
isSource: showDetail)
```

14. We will now move on to the `ContentView` view, where we add the properties to drive the animation:

```
struct ContentView: View {
    @State private var selectedItem: Item!
    @State private var showDetail = false
    @Namespace var animation
    var body: some View {
        //...
    }
}
```

15. Inside the body, we add the two components: `DestinationListView`, with the list of holiday destinations, and `DestinationDetailView`, with the information for the selected item:

```
var body: some View {
    ZStack {
        DestinationListView(selectedItem: $selectedItem,
                            showDetail: $showDetail,
                            animation: animation)
        if showDetail {
            DestinationDetailView(selectedItem: selectedItem,
                                showDetail: $showDetail,
                                animation: 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:

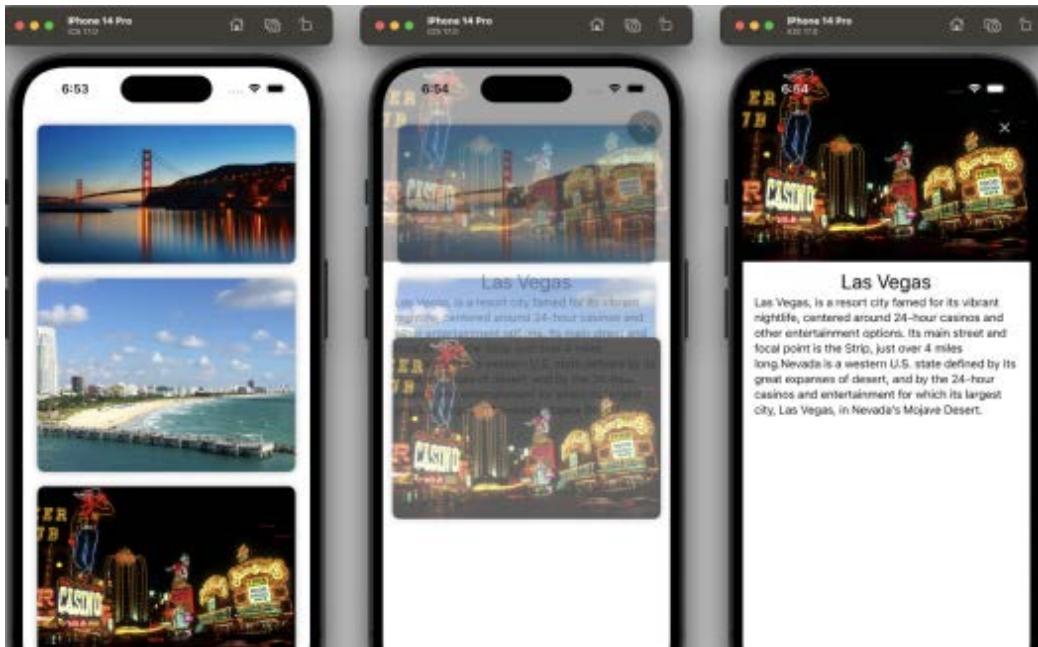


Figure 9.16: Hero view transition

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 of the transformation from 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, isSource: ...)
```

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 view and an ending view in different structs, and then we tell SwiftUI that those components are the same. After that, SwiftUI will figure out what transformations 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 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.

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 expect 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-3rd-Edition/tree/main/Resources/Chapter09/recipe11>:

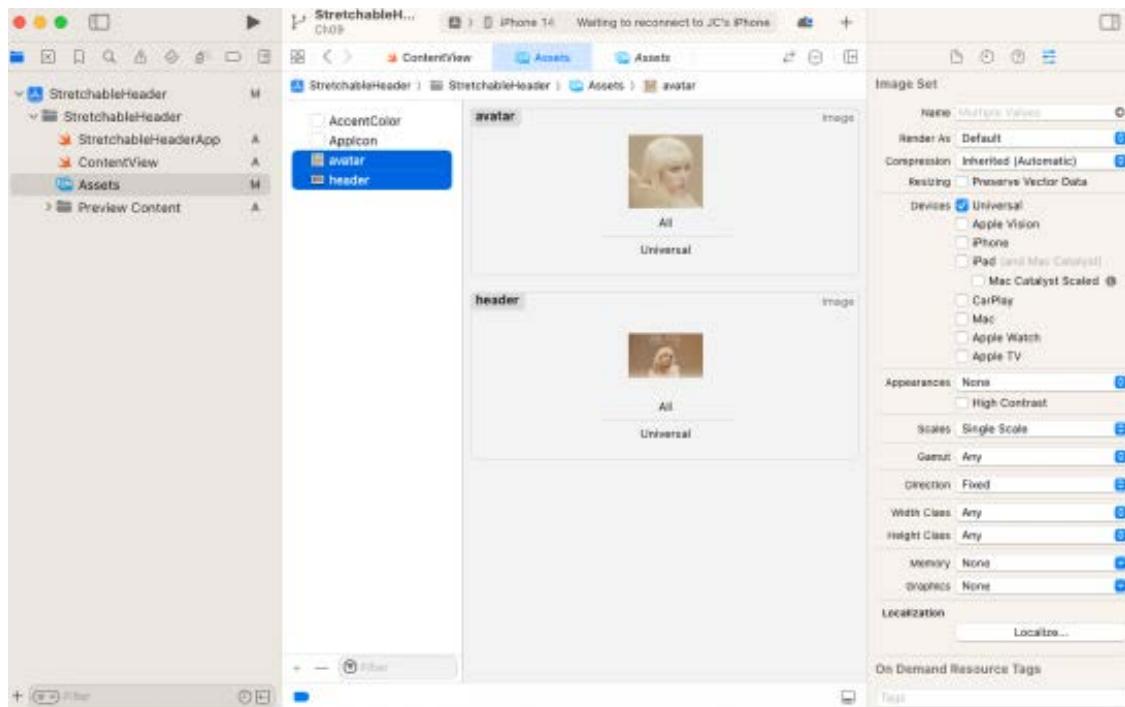


Figure 9.17: Importing the images

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 view:

```
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 imageResource: ImageResource
    var body: some View {
        GeometryReader { geometry in
            Image(imageResource)
                .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 the Xcode errors, we need to declare the following computed variables in an extension to `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(imageResource: .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 9.18: 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 `.scaledToFill()`, so that when the height increases, the width increases proportionally, and we have a scale-up effect.

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 higher 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, we will use a `ForEach` view to iterate over the array of users and, for each user, display a custom view, `CardView`, which will be declared later in the recipe:

```
var body: some View {
    GeometryReader { geometry in
        ZStack {
            ForEach(users) { user in
                if user.id > users.maxId - 4 {
                    CardView(user: user) { 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 and an initializer:

```
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, replace the content of the `body` variable with a `VStack`, wrapped in `GeometryReader`, which will be used in the next step in a `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) > threshold {  
                    onRemove(user)  
                } else {  
                    translation = .zero  
                }  
            }  
    )
```

9. The last thing that is missing is the `ratio` convenience function used in the previous step. Define it in an extension of the `DragGesture` value with the following code:

```
extension DragGesture.Value {  
    func percentage(in geometry: GeometryProxy) -> Double {  
        abs(translation.width / geometry.size.width)  
    }  
}
```

10. 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 9.19: Stack of swipeable cards

How it works...

Even though the code looks a bit long, 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 calculate the width and offset as a simple proportion of the ID of the user, assuming that they are discrete and incremental.

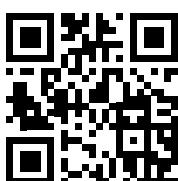
Step 6 is where most of the logic resides. Let's skip almost all the code, which basically configures the layout of the card, and concentrate on the gesture. The `onChange` callback in the `DragGesture` object sets the `@State` variable `translation` to the value of the dragged object. The `translation` 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.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

<https://packt.link/swiftUI>



10

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 the imperative world of UIKit, you should tell a view what it must do depending on the state. In SwiftUI, views react to changes. SwiftUI has three ways of reaching these goals:

- Using the binding property wrappers that SwiftUI provides
- Using `Observation`, a new framework introduced in iOS 17
- Using `Combine`, the framework that Apple introduced to implement functional reactive programming

We will learn about the first two mechanisms in this chapter and in *Chapter 11, Driving SwiftUI with Combine*, we'll learn how to use Combine to drive changes in SwiftUI views.

Property wrappers are a way to decorate a property and were introduced in Swift 5.1. There are four different ways of using the binding property wrappers 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 and the type must conform to the `ObservableObject` protocol.
- `@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.

If you have already read the recipes in the previous chapters, you must have encountered these property wrappers, but it is now time to concentrate on them and understand them better.

Using the `state` binding 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 views, depending on the relationships between the state and multiple or single views.

Once you master these techniques, we will cover `Observation`, a Swift-specific implementation of the observer design pattern. `Observation` uses Swift macros to optimize the way `ObservableObject` types work with SwiftUI. The `Observation` framework is Apple's recommended way to manage model data in your apps, and it is recommended to migrate code that uses the `ObservableObject` protocol to the `Observable()` macro, which is part of `Observation`.

In this chapter, 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`
- Using `Observation` to manage model data

Technical requirements

The code in this chapter is based on Xcode 15.0 and iOS 17.0. You can download and install the latest version of Xcode from the App Store. You'll also need to be running macOS Ventura (13.5) or newer.

Simply search for Xcode in the App Store and select and download the latest version. Launch Xcode and follow any additional installation instructions that your system may prompt you with. Once Xcode has fully launched, you're ready to go.

All the code examples for this chapter can be found on GitHub at <https://github.com/PacktPublishing/SwiftUI-Cookbook-3rd-Edition/tree/main/Chapter10-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 done, 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:

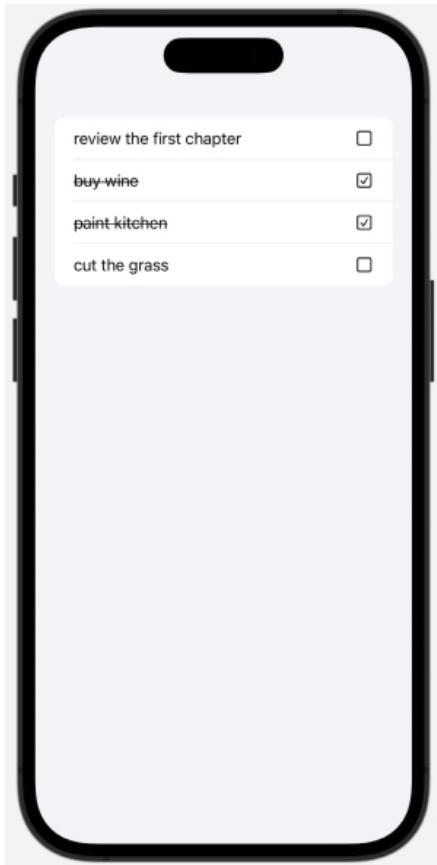


Figure 10.1: Different to-do renderings, depending on their completion status

How it works...

If there is only one recipe that you must understand and remember from this chapter, I believe that this is it.

Even if it is 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 being 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 a mutable and even better 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 `HStack`. This is necessary to give the whole row as a hit area to the gesture. Otherwise, only the text and the image would have been sensitive to the touches, leaving the white area in between not sensitive.

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 initial 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)
    ]
//..
}
```

- Now, render the to-dos, striking a line through the description if they are complete, and changing the checkmark accordingly. Do this by replacing the contents of the body variable with 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")
        }
    }
}
```

- Then, add a tap gesture to the Todo component to change the state of done:

```
HStack {
    //...
}
.contentShape(Rectangle())
.onTapGesture {
    todo.done.toggle()
}
```

- Create a new view 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 {
    // action code will come later
} label: {
    Text("Add")
        .padding(.horizontal, 16)
        .padding(.vertical, 8)
        .foregroundStyle(.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 to create a new task. We will also disable the button if the description is empty:

```
//...
Button {
    todos.append(Todo(description: newTodoDescription, done: false))
    newTodoDescription = ""
} label: {
    //...
}
.disabled(newTodoDescription.isEmpty)
```

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)
}
```

The app is improved: not only we can change the state of the to-dos, but we can also add new to-dos, as shown in the following screenshot:

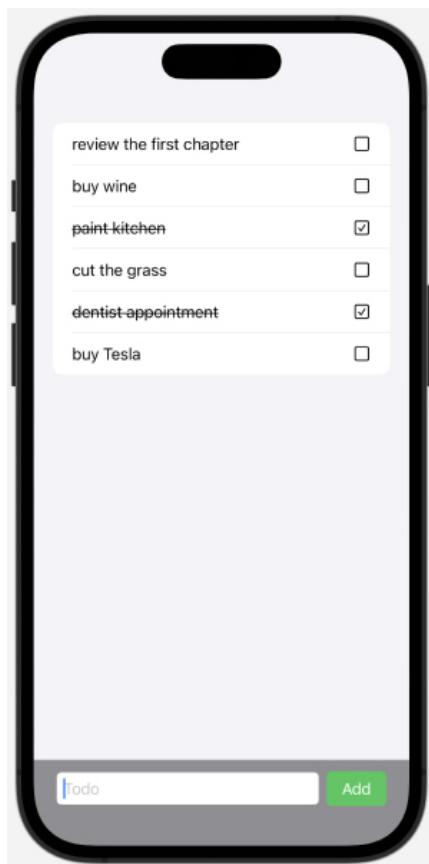


Figure 10.2: Adding new to-dos to the list

How it works...

As we mentioned in the introduction to the recipe, when a child view must 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, like 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. 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. We must use a different property wrapper, `@ObservedObject`, and the observed object must conform to the `ObservableObject` protocol. Furthermore, the properties of this object that will be observed in the view must be decorated with `@Published` property wrapper. With this property wrapper, when the properties mutate, the view will be notified, and the body of the view will be 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 comes with associated privacy concerns, and the user must be notified that the app is accessing the device's location. For this reason, Apple forces the app to ask the user to explicitly grant permission to use the `CoreLocation` framework.

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, as shown in the following screenshot:

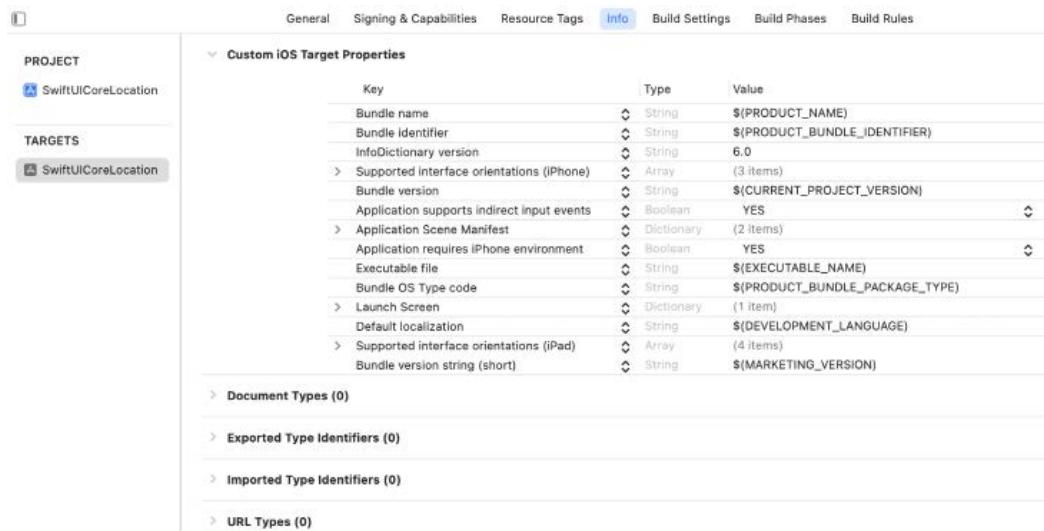


Figure 10.3: Selecting the Project Info tab

2. Hover the mouse over any key text box and a plus button will appear to the right of the box. Click on the + button and a new row will be added. In the newly added row, click on the drop-down menu, scroll through the options and, among the various options available, select the **Privacy - Location When In Use Usage Description** key. After the key is selected it will appear in the left text box. Move to the value text box and type a message to the user. In the following figure we show how we did it:

InfoDictionary version	String	6.0
Privacy - Location When In Use Usage Description	String	Enable Location Services please
Bundle version	String	<code>\$CURRENT_PROJECT_VERSION</code>

Figure 10.4: 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 authorization 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()
        locationManager.delegate = self
        locationManager.distanceFilter = 10
        locationManager.desiredAccuracy = kCLLocationAccuracyBest
        locationManager.requestWhenInUseAuthorization()
        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 }
        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 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 latitude = self?.coordinate.latitude else { return "-" }
        return String(format: "%.3f", latitude)
    }

    var longitudeDescription: String {
        guard let longitude = self?.coordinate.longitude else { return "-" }
        return String(format: "%.3f", longitude)
    }
}
```

6. 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)"))
    }
}

#Preview {
    ContentView(locationManager: LocationManager())
}
```

- Finally inject an instance of `LocationManager` to the `ContentView` from the app struct:

```
struct SwiftUICoreLocationApp: App {
    var body: some Scene {
        WindowGroup {
            ContentView(locationManager: LocationManager())
        }
    }
}
```

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:

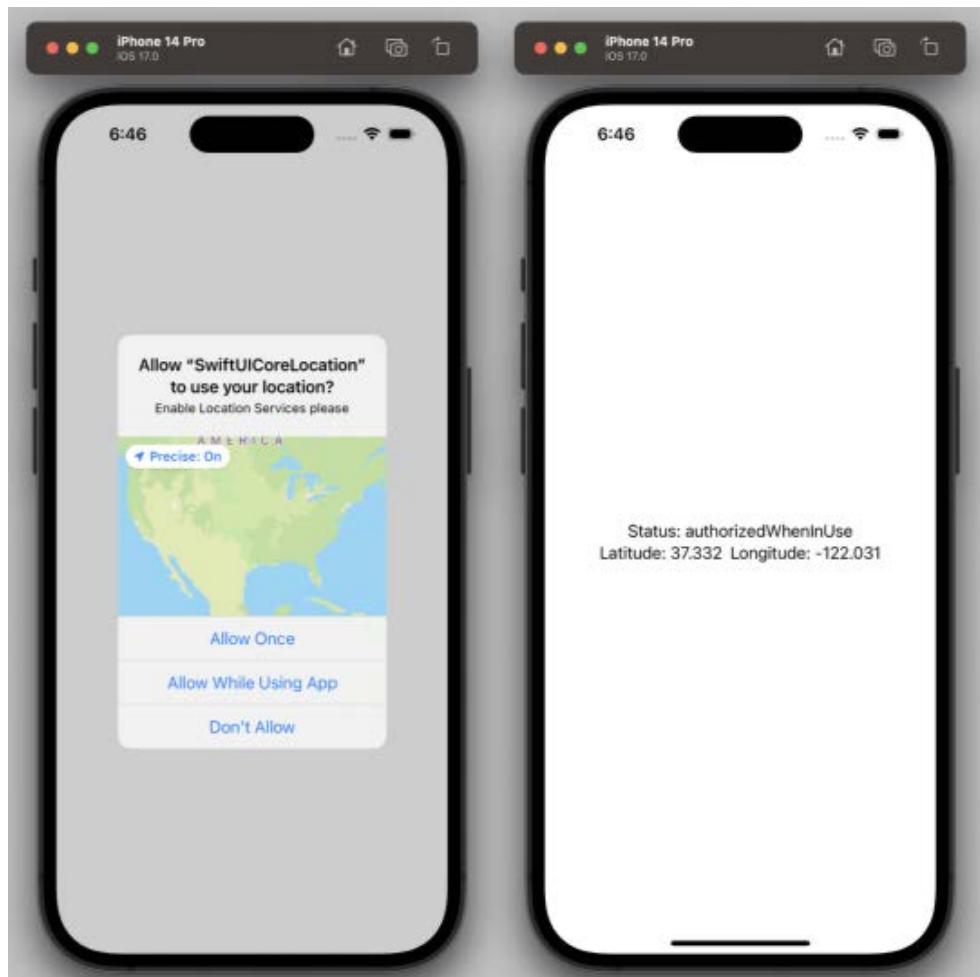


Figure 10.5: Showing user locations with SwiftUI

How it works...

Using a class as a state variable for SwiftUI is a matter of conforming the class to the `ObservableObject` protocol 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 a Combine publisher. If you are curious to see what you can do with Combine, we'll introduce it in *Chapter 11, 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 increase the feature set and bring it to parity with UIKit.

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 `@ObservedObject` 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 developers assumed 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 counterintuitive effect, and how `@StateObject` is 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, which will display a counter in the screen with two buttons. One button to increment the counter and another button to decrement the counter. The counter is owned by a child view, which is contained in the *container* view `ContentView`. The container view has a `Text` view, which displays the current time and a button labeled **Refresh**. When the **Refresh** button is tapped, the current time is updated in the text view:

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()
                        .foregroundStyle(.white)
                        .background(.red)
                }
                Button {
                    counter.inc()
                } label: {
                    Text("+")
                        .padding()
                        .foregroundStyle(.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 with two forward slashes:

```
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 value of the refreshedAt variable to the current date:

```
VStack(spacing:12) {
    //...
    CounterView()
    Button {
        refreshedAt = Date()
    } label: {
        Text("Refresh")
            .padding()
            .foregroundStyle(.white)
            .background(.blue)
    }
}
```

While running the app, we can increment or decrement the counter and refresh the container view, as shown in the following screenshot:

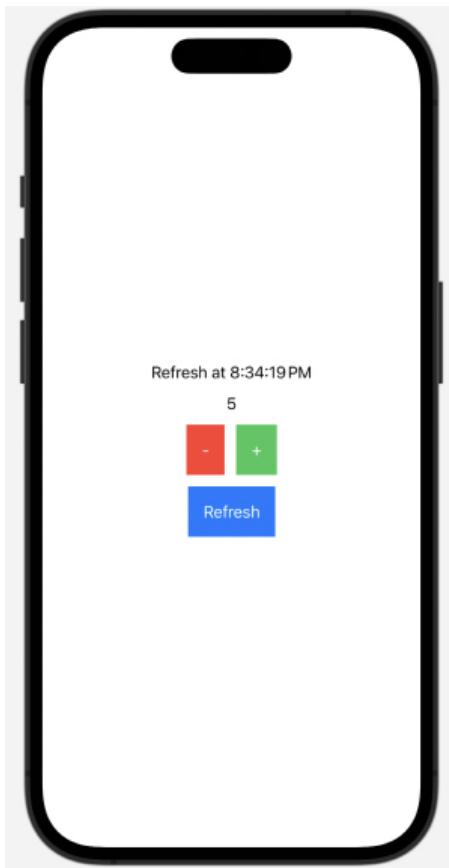


Figure 10.6: Incrementing or decrementing a child view

How it works...

Now, let's see how the app behaves with an `@StateObject` and with a `@ObservedObject` variable.

We will start with the Counter model, which we defined as an `@ObservedObject`, by implementing 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 something other 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` variable `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 subview.

Sharing state objects with multiple Views using `@EnvironmentObject`

In many cases, there are dependent instances that must be shared among several views, even without having a tight relationship with the views. Think of a `Theme`, or a `NetworkService`, or a `UserProfile`. Passing these shared instances through the view hierarchy can be annoying. If a view doesn't need the `NetworkService` instance, for example, it still should have it as a property in case one of the subviews needs it.

SwiftUI solves this dependency injection with the concept of the view's `Environment`, a place to store common objects: usually `ObservableObject` objects: that will be shared among views. An `Environment` is started in the root ancestor of the view hierarchy and can be changed further down in the hierarchy by adding new objects to it.

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 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 player. For that, we will use a simple `MusicPlayer` object that simulates the actual song that's playing, which is saved in the `Environment`.

By the end of the recipe, you will have a grasp of what to put in the `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-3rd-Edition/tree/main/Resources/Chapter10/recipe5>.

Add them to the **Assets** images and we will be ready to go:

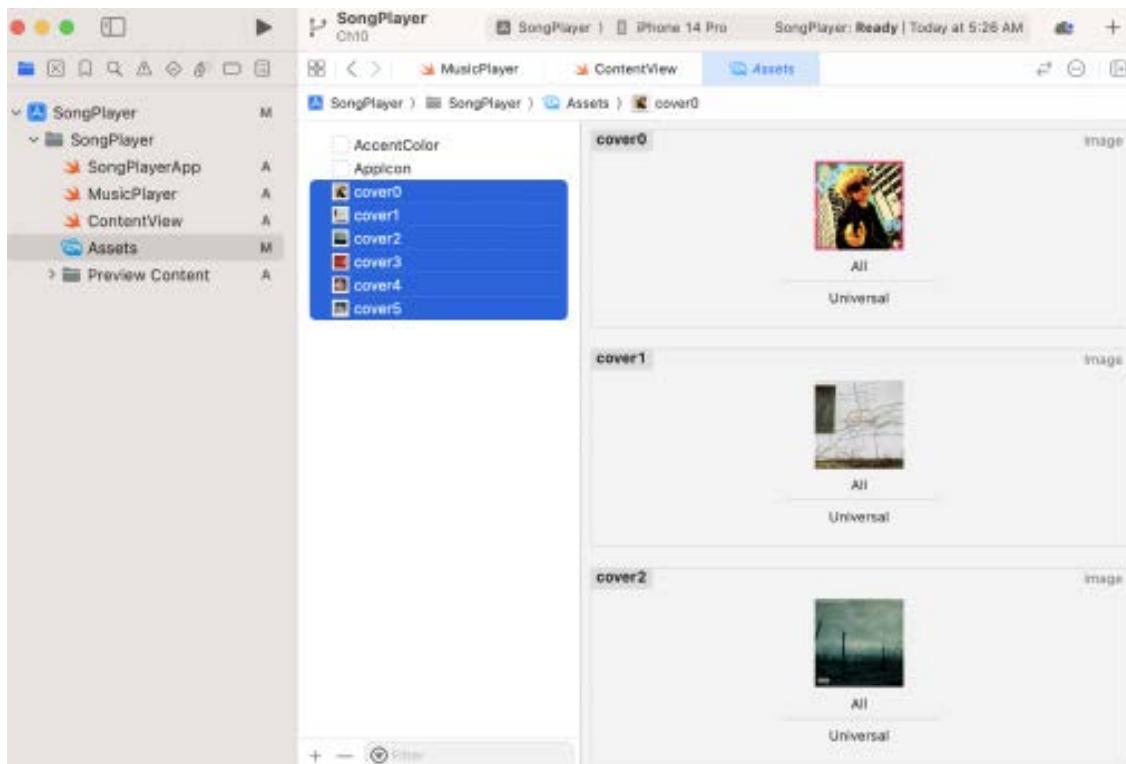


Figure 10.7: Importing the images

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 implementation of the views, 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 tapButton(song: Song) {  
        if currentSong == song {  
            currentSong = nil  
        } else {  
            currentSong = song  
        }  
    }  
}
```

3. Add the list of songs to the `ContentView` struct and present it in a `ListView`:

```
struct ContentView: View {  
    @EnvironmentObject private var musicPlayer: MusicPlayer  
  
    private let songs = [  
        Song(artist: "Luke", name: "99", cover: "cover0"),  
        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 {  
        NavigationStack {  
            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 {
            withAnimation {
                musicPlayer.tapButton(song: song)
            }
        } label: {
            Image(systemName: buttonText)
                .font(.system(.largeTitle))
                .foregroundStyle(.black)
        }
    }
}
```

5. For each song in the list in `ContentView`, we will add a proper `SongView`:

```
NavigationStack {
    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)
                    .resizable()
                    .aspectRatio(contentMode: .fill)
                    .frame(width: 100, height: 100)
            }
            VStack(alignment: .leading) {
                Text(song.name)
                Text(song.artist).italic()
            }
        }

        Spacer()
        PlayButton(song: song)
    }
    .buttonStyle(.plain)
}
}
```

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)
                .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 the `Environment` with `@EnvironmentObject`, but we don't initialize its value anywhere. Inject an instance of `MusicPlayer` into the `Environment` in the main scene using the `.environmentObject` view modifier, so it can be passed down the view hierarchy:

```
@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 use the preview in the canvas. Can you spot the reason as to why?

The reason for the error is that we are not running the app from the `@main` struct in the preview, but from the `#Preview` macro. We need to inject a `MusicPlayer` instance into the preview:

```
#Preview {
    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(24)
        } else {
```

```
        EmptyView()
    }
}
```

11. Finally, in the `ContentView` struct, embed the `NavigationStack` with the list of songs into a `ZStack` and add `MiniPlayerView` at the bottom:

```
var body: some View {
    ZStack(alignment: .bottom) {
        NavigationStack {
            //...
        }
        MiniPlayerView()
            .background(.gray)
            .offset(y: 44)
            .zIndex(1.0)
    }
}
```

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:

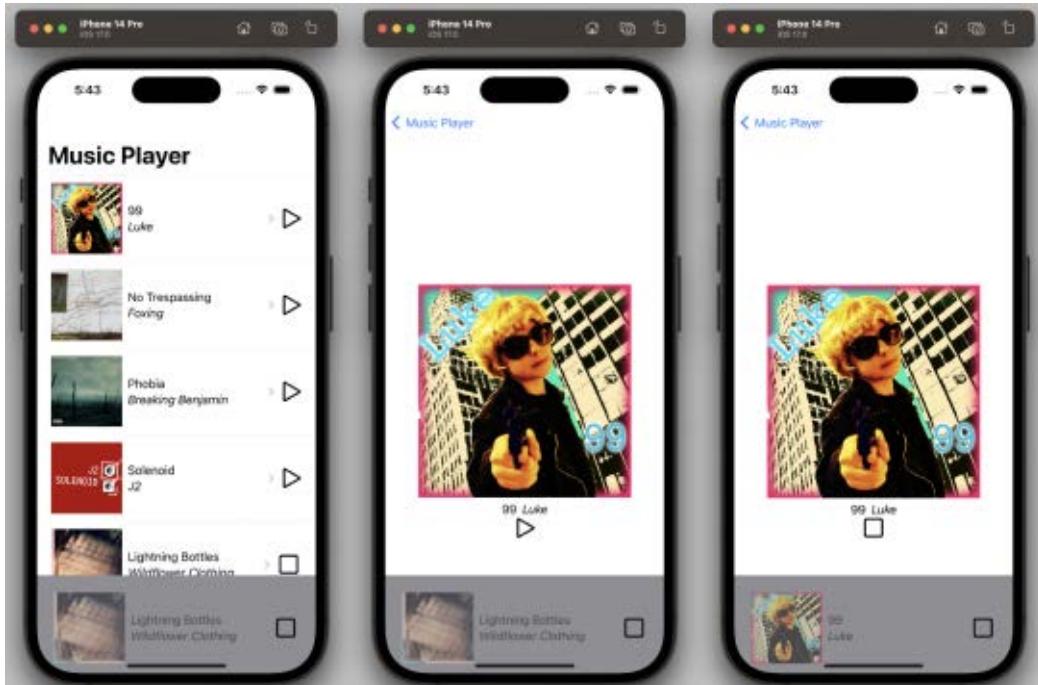


Figure 10.8: Playing a song in different views

How it works...

The view's Environment feature is a nice (and Apple's official way) of solving a problem that has often been solved using singletons or global variables: the problem of common dependencies and how to inject them. If you have something that could be needed in multiple unrelated views, storing it in the Environment could be a viable solution.

Every view has access to the Environment set by its 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 Dictionary, 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 the 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>).

Using Observation to manage model data

As we mentioned in the introduction section, Observation is a new framework introduced in iOS 17 to simplify the use of model data with SwiftUI. It replaces the use of `ObservableObject` instances with `@Published` properties, with `Observable` classes. To declare a class as observable we attach the `Observable()` macro to the class declaration. This macro declares and implements the conformance to the `Observable` protocol for us.

When we work with observable types, SwiftUI automatically tracks changes to the properties used in views, minimizing view rendering, and optimizing the performance of our view code.

To illustrate the use of observable types, we are going to implement a simplified version of the Reminders app, which comes installed by default in any iOS device. Our app will have two views:

- The root view with the list of reminders
- A detail view used to view and edit the different properties of a reminder

Both views will have the ability to modify the reminders and consequently will share data. In the List view, we will be able to add a new reminder, delete an existing one, and even modify the titles of reminders. In the detail view, we will be able to modify all the properties of reminders.

The final app should look like the following:

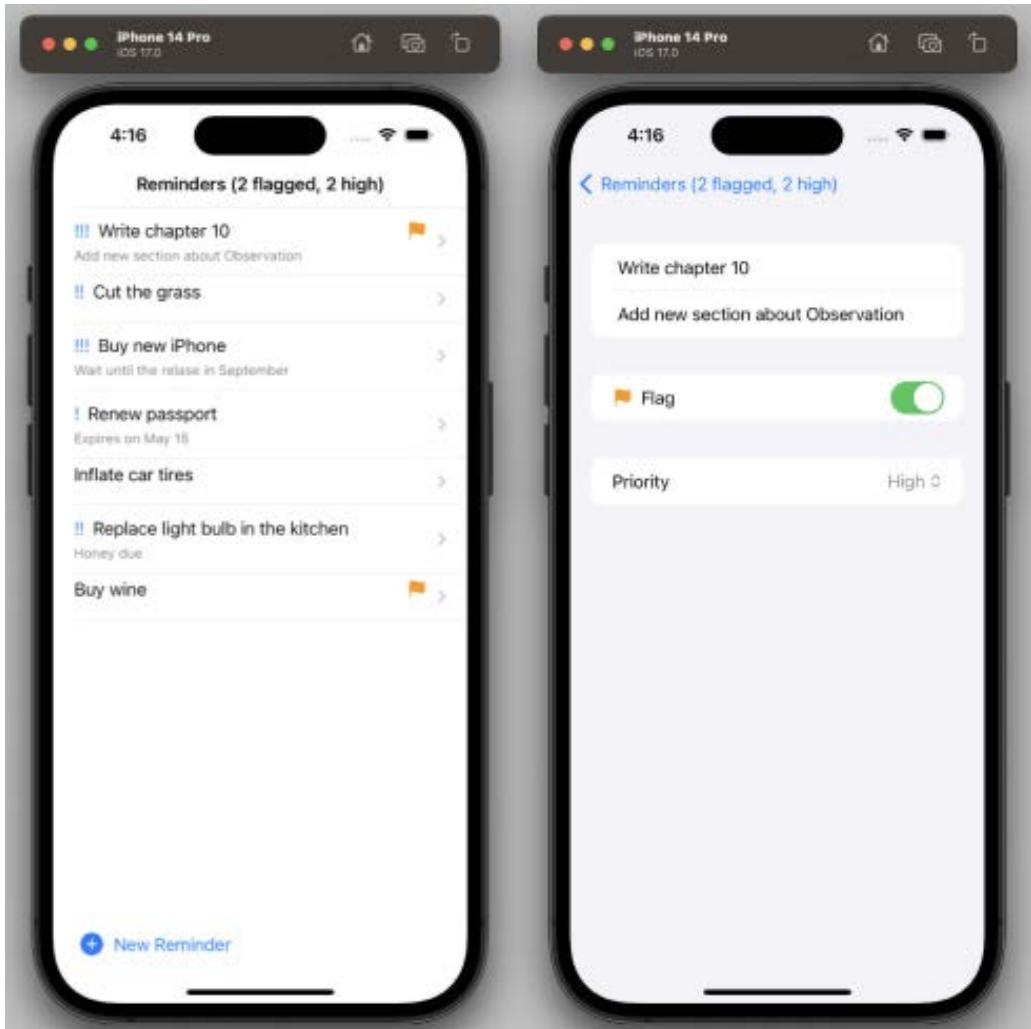


Figure 10.9: Our reminders app

For our data model, we will have two classes: a `Reminder` class to represent a reminder and a `Storage` class to store our reminders. We will use the `Environment` to share our data.

By the end of the recipe, you will have an understanding of how to use observable objects to simplify your SwiftUI code.

Getting ready

Let's start this recipe by creating a SwiftUI app called `Reminders`.

How to do it...

We are going to define our data model first, and then we will work on the views.

1. Let's start by creating a `Priority` enum to represent the priority of the reminder. Add a new file named `Priority.swift` and declare the type:

```
enum Priority: String, CaseIterable, CustomStringConvertible {  
    case none  
    case low  
    case medium  
    case high  
  
    var description: String {  
        self.rawValue.capitalized  
    }  
  
    var shortDescription: String {  
        switch self {  
        case .none: ""  
        case .low: "!"  
        case .medium: "!!"  
        case .high: "!!!"  
        }  
    }  
  
    static let allDescriptions = Self.allCases.map { $0.description }  
}
```

2. Now create a file named `Reminder.swift` and declare the `Reminder` class, which represents our reminder data:

```
import SwiftUI  
  
@Observable final class Reminder: NSObject, Identifiable {  
    var title: String  
    var notes: String  
    var flag: Bool  
    var priority: Priority  
  
    init(  
        title: String,  
        notes: String = "",
```

```
        flag: Bool = false,
        priority: Priority = .none
    ) {
    self.title = title
    self.notes = notes
    self.flag = flag
    self.priority = priority
}

var isEmpty: Bool {
    title.isEmpty
}
}
```

3. At the end of the file, add an extension with some sample data:

```
extension Reminder {
    static let sample = Reminder(title: "Write chapter 10", notes: "Add new section about Observation", flag: true, priority: .high)
    static let samples = [
        sample,
        Reminder(title: "Cut the grass", priority: .medium),
        Reminder(title: "Buy new iPhone", notes: "Wait until the release in September", priority: .high),
        Reminder(title: "Renew passport", notes: "Expires on May 15", priority: .low),
        Reminder(title: "Inflate car tires"),
        Reminder(title: "Replace light bulb in the kitchen", notes: "Honey due", priority: .medium),
        Reminder(title: "Buy wine", flag: true)
    ]
}
```

4. Let's now move on to the `Storage` class, which will be responsible for managing our list of reminders. Create a new Swift file named `Storage.swift` and type the following:

```
import SwiftUI

@Observable final class Storage {
    var reminders: [Reminder]

    init(reminders: [Reminder] = []) {
        self.reminders = reminders
    }
}
```

```
func add(reminder: Reminder) {
    reminders += [reminder]
}

func remove(reminder: Reminder) {
    reminders.removeAll { $0 === reminder }
}

func purgeEmptyReminders() {
    reminders.removeAll { $0.isEmpty }
}

func flaggedReminders() -> [Reminder] {
    reminders.filter { $0.flag }
}

func reminders(withPriority priority: Priority) -> [Reminder] {
    reminders.filter { $0.priority == priority }
}
```

5. Since we are going to use the `Environment` to pass an instance of `Storage` to the view hierarchy, add the following at the end of the file:

```
private struct StorageKey: EnvironmentKey {
    static var defaultValue: Storage = Storage()
}

extension EnvironmentValues {
    var storage: Storage {
        get { self[StorageKey.self] }
        set { self[StorageKey.self] = newValue }
    }
}
```

6. With our data model completed, we can work on the views. As is customary when working with a list of items in SwiftUI, we will have three views: a view with a list, a view for a row in the list, and a detail view, which we will show when the user taps on an item in the list. Let's start by creating the row view. Add a new SwiftUI file named `ReminderRowView.swift` with the following content:

```
struct ReminderRowView: View {
```

```
@BindableView reminder: Reminder
@FocusState private var isFocused: Bool
var body: some View {
    VStack(alignment: .leading) {
        HStack {
            if reminder.priority != .none {
                Text(reminder.priority.shortDescription)
                    .foregroundStyle(.blue)
            }
            TextField("New Reminder", text: $reminder.title)
                .focused($isFocused)
            Spacer()
            if reminder.flag {
                Image(systemName: "flag.fill")
                    .foregroundStyle(.orange)
            }
        }
        Text(reminder.notes)
            .font(.footnote)
            .foregroundStyle(.secondary)
            .lineLimit(3)
    }
    .onAppear {
        if reminder.isEmpty {
            isFocused = true
        }
    }
}
}

#Preview {
    ReminderRowView(reminder: .sample)
}
```

If everything went well, the preview should look like *Figure 10.10*. Using the live preview, notice how you can change the title of the reminder:

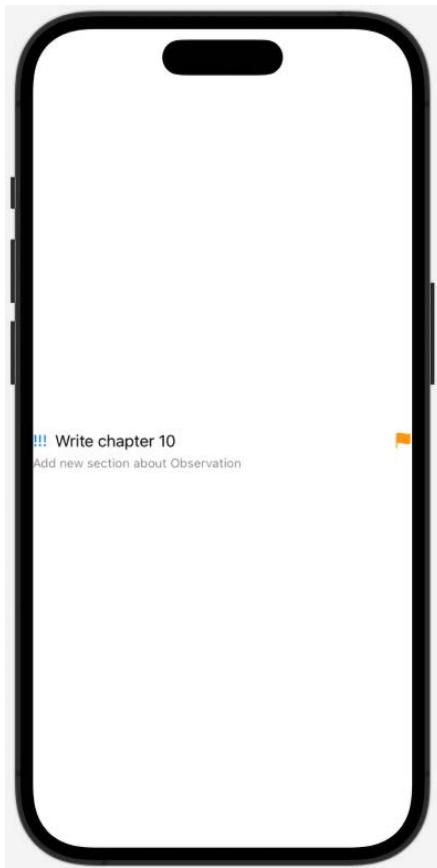


Figure 10.10: ReminderRowView preview

- Once we have the row view defined, we now need to tackle the view for the reminder details. Create a new SwiftUI file named `ReminderView.swift`. This view will be a form with three sections: the first section for the title, the second for the flag, and the third for the priority:

```
struct ReminderView: View {  
    @Bindable var reminder: Reminder  
  
    private var titleNotesView: some View {  
        EmptyView()  
    }  
}
```

```
private var flagToggleView: some View {
    EmptyView()
}

@State private var priorityDescription = Priority.none.description

private var priorityPicker: some View {
    EmptyView()
}

var body: some View {
    Form {
        Section {
            titleNotesView
        }
        Section {
            flagToggleView
        }
        Section {
            priorityPicker
        }
    }
}

#Preview {
    ReminderView(reminder: .sample)
}
```

8. Let's define the section for the title and the notes of the reminder. Replace the declaration of `titleNotesView` with the following:

```
private var titleNotesView: some View {
    Group {
        TextField("Title", text: $reminder.title)
            .padding(.horizontal, 5)
        ZStack(alignment: .topLeading) {
            if reminder.notes.isEmpty {
                Text("Notes")
                    .foregroundStyle(.placeholder)
```

```
        .padding(.horizontal, 5)
        .padding(.top, 8)
    }
    TextEditor(text: $reminder.notes)
        .frame(maxHeight: 100)
    }
}
}
```

9. For the section representing the flag of the reminder, replace the declaration of `flagToggleView` with the following:

```
private var flagToggleView: some View {
    Toggle(isOn: $reminder.flag) {
        HStack {
            Image(systemName: reminder.flag ? "flag.fill" : "flag")
                .foregroundStyle(.orange)
            Text("Flag")
        }
    }
}
```

10. Finally, complete the section for the priority picker. Replace the definition of `priorityPicker` with the following:

```
private var priorityPicker: some View {
    Picker("Priority", selection: $priorityDescription) {
        ForEach(Priority.allDescriptions, id:\.self) { description in
            Text(description)
            if description == Priority.none.description {
                Divider()
            }
        }
    }
    .onAppear {
        priorityDescription = reminder.priority.description
    }
    .onChange(of: priorityDescription) { _, newValue in
        let priority = Priority(rawValue: newValue.lowercased())!
        $reminder.priority.wrappedValue = priority
    }
}
```

It is a complex view, because we can not only view the properties of the reminder but also edit them all. If everything went smoothly, the preview should look like *Figure 10.11*. Using the live preview, experiment with the different controls and change the properties of the reminder:

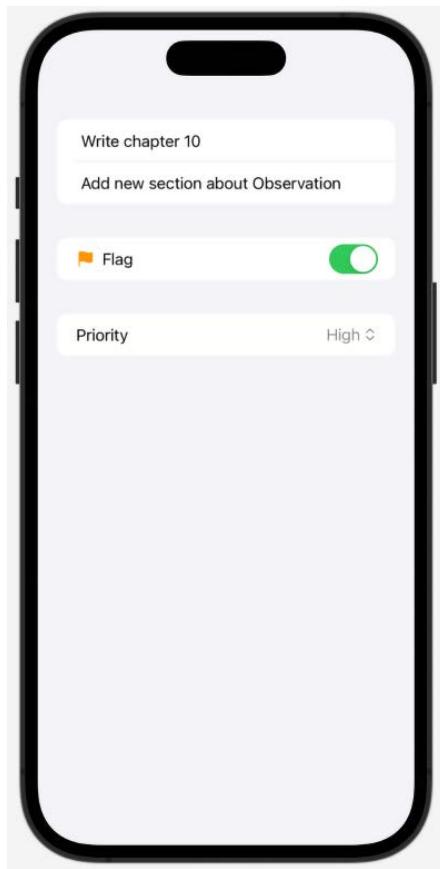


Figure 10.11: ReminderView preview

11. Next, let's work on the view with the list of reminders, which is going to be the root view of the app. Create a new SwiftUI file named `ReminderListView.swift`. The contents of the file should be as follows:

```
struct ReminderListView: View {  
    @Environment(\.storage) private var storage  
    var body: some View {  
        NavigationStack {  
            List(storage.reminders) { reminder in  
                NavigationLink(value: reminder) {
```

```
        ReminderRowView(reminder: reminder)
            .onSubmit {
                storage.purgeEmptyReminders()
            }
        }
        .swipeActions {
            Button("Delete", role: .destructive) {
                storage.remove(reminder: reminder)
            }
        }
    }
    .listStyle(.plain)
    .toolbar {
        ToolbarItemGroup(placement: .bottomBar) {
            Button {
                let newReminder = Reminder(title: "")
                storage.add(reminder: newReminder)
            } label: {
                HStack {
                    Image(systemName: "plus.circle.fill")
                    Text("New Reminder")
                }
            }
            Spacer()
        }
    }
    .navigationTitle("Reminders (\(storage.flaggedReminders().count) flagged, \(storage.reminders(withPriority: .high).count) high)")
        .navigationBarTitleDisplayMode(.inline)
        .navigationDestination(for: Reminder.self) { reminder in
            ReminderView(reminder: reminder)
        }
        .onAppear {
            storage.purgeEmptyReminders()
        }
    }
}
```

```
#Preview {
    ReminderListView()
        .environment(\.storage, Storage(reminders: Reminder.samples))
}
```

If everything went smoothly, the preview should look like this:

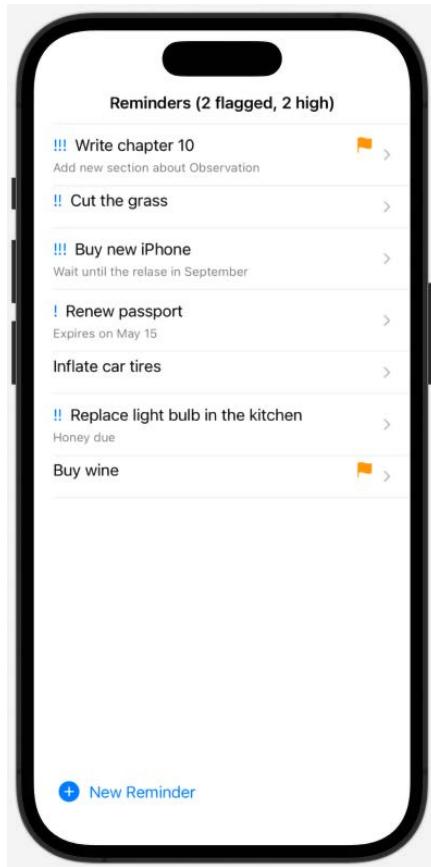


Figure 10.12: ReminderListView preview

At this point, using the live preview, you can try out the features of the app.

- Finally, let's close the recipe by working on the App struct. Head to `ReminderApp` and modify its content so it looks like the following:

```
struct RemindersApp: App {
    @State private var storage = Storage(reminders: Reminder.samples)
```

```
var body: some Scene {  
    WindowGroup {  
        ReminderListView()  
            .environment(\.storage, storage)  
    }  
}
```

13. Now run the app in the iOS simulator of your choice and try all the features: add a reminder from the root view, delete a reminder by swiping left on the corresponding item in the list, modify the title without leaving the list, tap on an item to edit its properties, and so on.



Figure 10.13: Our Reminders app running in the iOS simulator

How it works...

According to Apple's documentation, by using `Observable` instances and the `Observable` macro, we benefit from several advantages over using `ObservableObject` instances with `@Published` properties:

- We can track optional types, which is not possible with `ObservableObject`.
- We can simply use `State` and `Environment` instead of `StateObject` and `EnvironmentObject`.
- Views will update to the observable property changes that a view's body reads, instead of any property changes that occur to an observable object, which can help improve the app's performance.

In our app, we need to share the data model among the three views we have. We need to share the `Storage` class, which holds the list of reminders, and we need to share `Reminder` instances as well, as our app displays the attributes of a reminder too. We used the `Observable()` macro in both classes by prepending the class declaration with `@Observable`. We didn't need to conform to the `ObservableObject` protocol or mark the properties we wanted to share with the views as `@Published`, as `Observation` takes care of detecting changes to the properties automatically. Even better, `Observation` only tracks changes to the view's properties which are used in the views.

We marked `Storage` as `Observable` because we use the class in `ReminderListView` to retrieve and display the list of reminders. When we add a reminder or modify a reminder our view will react to the changes. The way we access the storage is through the `Environment`. In the `ReminderApp` struct, we create a `@State` variable to store an instance of `Storage` initialized with sample data:

```
@State private var storage = Storage(reminders: Reminder.samples)
```

Then we pass it to the root view of the app using the `Environment`:

```
ReminderListView()  
    .environment(\.storage, storage)
```

In `ReminderListView`, we use the `@Environment` property wrapper to access the value:

```
@Environment(\.storage) private var storage
```

In the other two views that interact with the reminder, we declare a variable of type `Reminder`. Since we need to change the properties of the reminder, we use the new `@Bindable` property wrapper instead of the `@ObservedObject` used before iOS 17:

```
@Bindable var reminder: Reminder
```

If our view only needs to display the values without changing them, a simple variable declaration would be sufficient, and we could eliminate the `@ObservedObject` property wrapper:

```
var reminder: Reminder
```

See also

- Discover `Observation` in `SwiftUI`: <https://developer.apple.com/videos/play/wwdc2023/10149/>
- `Observation`: <https://developer.apple.com/documentation/Observation>
- Managing model data in your app: <https://developer.apple.com/documentation/swiftui/managing-model-data-in-your-app>
- Migrating from the `Observable Object` protocol to the `Observable` macro: <https://developer.apple.com/documentation/swiftui/migrating-from-the-observable-object-protocol-to-the-observable-macro>
- Create custom environment values: [https://developer.apple.com/documentation/swiftui/environmentvalues/](https://developer.apple.com/documentation/swiftui/environmentvalues)

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

<https://packt.link/swiftUI>



11

Driving SwiftUI with Combine

In this chapter, we'll learn how to manage the state of SwiftUI views using **Combine**. At 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 has been 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 a typically Apple way, Apple took the best practices that have matured over years of trial and error in the community, and instead of acquiring either **ReactiveSwift** or **RxSwift**, Apple decided to reimplement their concepts, simplify the 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 foundation of **Combine** is the concept of *publisher* and *subscriber*. A *publisher* emits elements that can change over time, and a *subscriber* receives those elements from the publisher. For example, **URLSession** uses the `dataTaskPublisher(for:)` instance method to return a publisher that wraps a URL session data task. One or more *subscribers* can observe that publisher and react when a new element 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 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, re-tryers, 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 functional reactive programming and the way it is implemented in `Combine`.

By the end of the chapter, you will know when and how to use `Combine` (or whether 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 the `Combine` framework to populate and change views when data changes.

We are going to cover the following recipes in this chapter:

- 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 15.0 and iOS 17.0. You can download and install the latest version of Xcode from the App Store. You'll also need to be running macOS Ventura (13.5) or newer.

Simply search for Xcode in the App Store and select and download the latest version. Launch Xcode and follow any additional installation instructions that your system may prompt you with. Once Xcode has fully launched, you're ready to go.

All the code examples for this chapter can be found on GitHub at <https://github.com/PacktPublishing/SwiftUI-Cookbook-3rd-Edition/tree/main/Chapter11-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 services 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 10, Driving SwiftUI with Data*.

Usually, when a reactive framework is used, instead of the common **Model-View-Controller (MVC)** pattern used in many iOS apps, people tend to use **Model-View-ViewModel (MVVM)**.

In this architectural pattern, 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 itself. For example, it will have a property for the current location, which 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 a mediator between the two.

The *ViewModel* is bound to the view with a one-to-one relationship between the view 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 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:

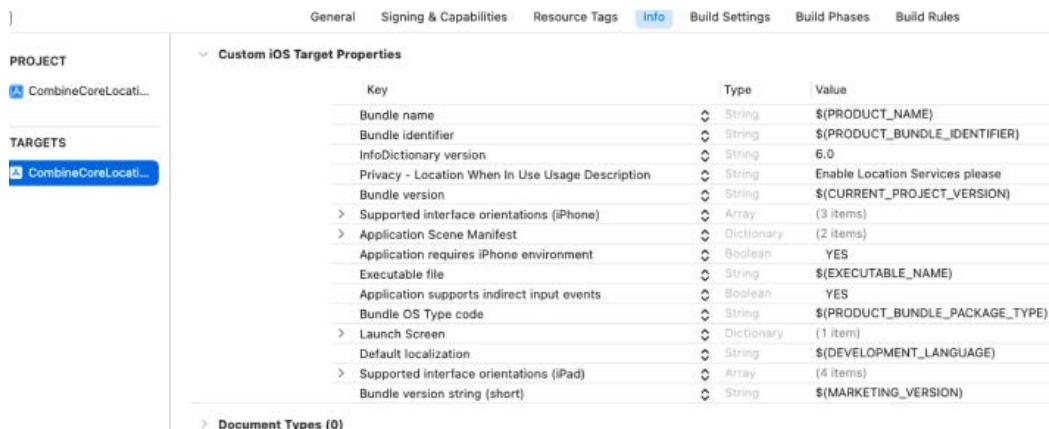


Figure 11.1: Selecting the Info tab

2. Hover over any key text box and a plus button will appear to the right of the box. Click on the plus button and a new row will be added. Select the **Privacy - Location When In Use Usage Description** key from the dropdown list of keys, then move to the value text box and type a value with a message to the user:

InfoDictionary version	String	6.0
Privacy - Location When In Use Usage Description	String	Enable Location Services please
Bundle version	String	<code>\$(CURRENT_PROJECT_VERSION)</code>

Figure 11.2: Message to the user to enable location services

How to do it...

As I have already mentioned in the introduction, we are going to implement this recipe following the MVVM pattern.

We will build three components:

- The status and location view in ContentView: the View

- A `LocationViewModel` class that interacts with the model and updates the view: the `ViewModel`
- A `LocationManager` 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. Go to `ContentView.swift` and start at the top of the file:

1. First, add import statements for `CoreLocation` and `Combine`, then implement a custom class as 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()
        locationManager.delegate = self
        locationManager.desiredAccuracy = kCLLocationAccuracyBest
        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 values and errors:

```
class LocationManager: NSObject {
    //...
    let statusPublisher = PassthroughSubject<CLAuthorizationStatus,
    LocationError>()
    let locationPublisher = PassthroughSubject<CLLocation?, Never>()
```

```
//...  
}
```

- Now, let's publish the values. We will implement the `CLLocationManagerDelegate` functions, where we collect location events and publish them with 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)  
    }  
}
```

- 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()  
    }  
}
```

- The public interface is not complete. Add a few more computed properties to serve the UI's needs:

```
class LocationViewModel: ObservableObject {  
    //...
```

```
var thereIsAnError: Bool {
    !errorMessage.isEmpty
}

var latitude: String {
    locationManager.latitudeDescription
}

var longitude: String {
    locationManager.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 an extension to the `Optional` protocol definition with a couple of computed variables:

```
extension Optional where Wrapped == CLLocation {
    var latitudeDescription: String {
        guard let latitude = self?.coordinate.latitude else { return "-" }
    }
}
```

```
        return String(format: "%.4f", latitude)
    }

    var longitudeDescription: String {
        guard let longitude = self?.coordinate.longitude else { return
        "-" }
        return String(format: "%.4f", 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 lhs, let rhs else {  
        return false  
    }  
    return lhs.distance(from: rhs) < 1  
}
```

9. After finishing the reactive `ViewModel`, let's complete the app with the `View`. First, create `ContentView` with a reference to the `ViewModel` 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 and 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:

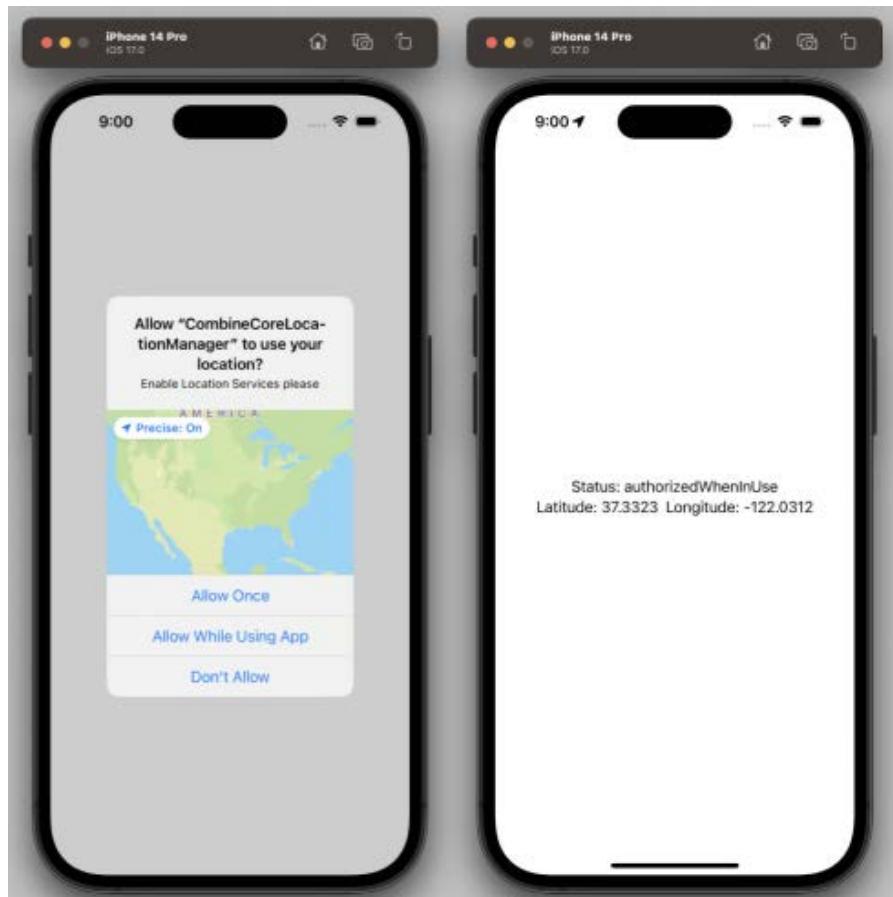


Figure 11.3: A reactive location manager

Congratulations: you've implemented your first Combine app!

How it works...

Something to notice before diving into the *Publishers* description is that we set the two `@Published` variables, `status` and `currentLocation`, 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 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 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 send 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(manager.authorizationStatus)
//...
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 `.failure` as its parameter instead of the authorization status:

```
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 explain 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 publishers.

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 deallocated when the container is deallocated (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 will 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 *Rx-Marbles* 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 subscription should be stored somewhere. Usually, it is stored in a Set of AnyCancellable: `private var cancellableSet: Set<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 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 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.
- Also, we will 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. We will include all our code in `ContentView.swift`. 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 { [self] _ in
                deciseconds = (deciseconds + 1) % 10
            }
        Timer.publish(every: 1.0, on: RunLoop.main, in: .default)
            .autoconnect()
            .sink { [self] _ in
                seconds = (seconds + 1) % 60
            }
        Timer.publish(every: 60.0, on: RunLoop.main, in: .default)
            .autoconnect()
            .sink { [self] _ in
                minutes = (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 the `ContentView`. 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(width: 1))")
                    .font(.system(size: 80))
            }
        }
    }
}
```

```
        .frame(width: 50)
    }
}
}
}
```

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")
                .foregroundStyle(.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 {
    func formatted(width: Int = 2) -> String {
        String(format: "%0\(width)d", self)
    }
}
```

Now, we can run the app and see that it measures the time in a nice and precise way:

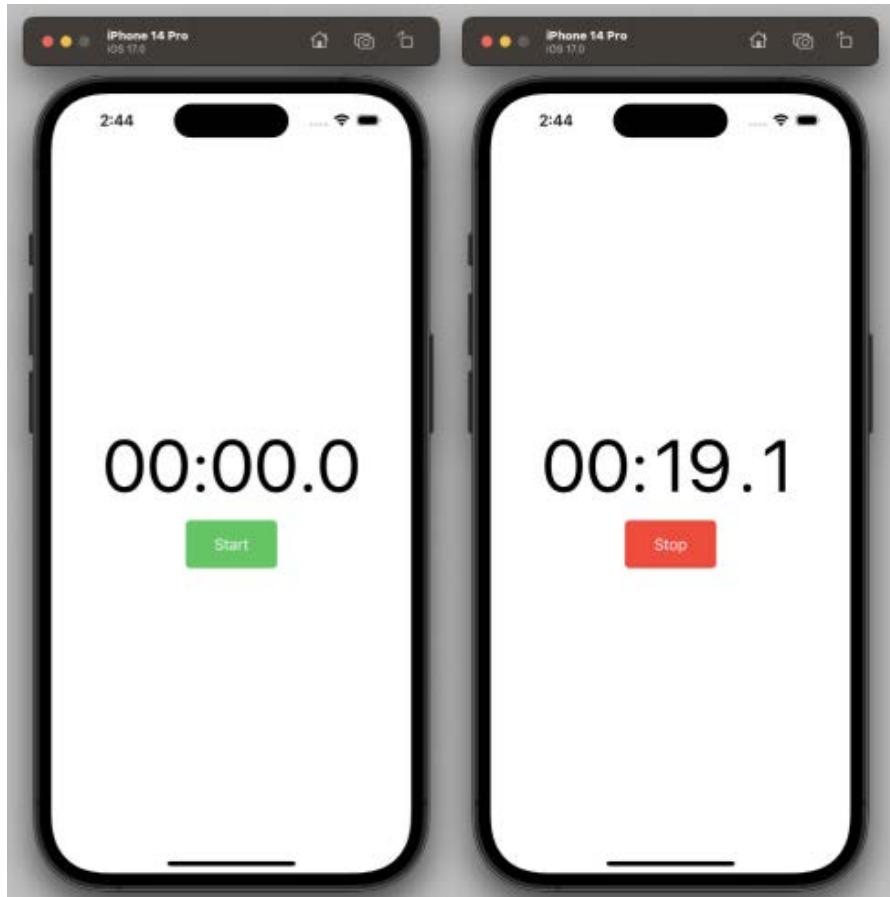


Figure 11.4: A reactive StopWatch application in action

How it works...

In this recipe, we introduced a couple of new things: a timer publisher and a way to collect all the cancellable chains 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` class, which is part of the Foundation framework, where you can set the interval and the RunLoop where it will emit the events. The `.autoconnect()` modifier will make the publisher start immediately provide a lead-in.

Regarding the `store()` function in the `CancellableSet` extension, it uses the `CancellableBuild` 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 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 must solve. 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 must 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 will move a step further toward 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 the MVVM pattern.

In our case, the business logic is the validation logic, and it will be encapsulated in a class called `SignupViewModel`. For simplicity, we will work in `ContentView.swift`:

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 validation  
    @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, we will add the *publishers* as computed properties in a *private extension* of *SignupViewModel*. We start with the publisher for the username, whose only logic is checking if the length of the *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 whether 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. Now, add the check for the strength of the password:

```
private extension SignupViewModel {
    //...
    var strongPasswordPublisher: AnyPublisher<Bool, Never> {
        $password
            .debounce(for: 0.2, scheduler:
```

```
       RunLoop.main)
    .removeDuplicates()
    .map(\.isStrongPassword)
    .eraseToAnyPublisher()
}
}
```

8. Note that here, we are checking if a string is strong by using an `isStrongPassword` 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 isStrongPassword: 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, confirmedPassword in
            password == confirmedPassword
        }
        .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)
            .foregroundStyle(.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)
            .foregroundStyle(.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. This button simulates the functionality of registering a user once the validations pass:

```
 VStack(spacing: 24) {
    //...
    Button {
        print("Successfully registered!")
    } label: {
        Text("Register")
            .foregroundStyle(.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. If you use the live preview of the canvas, you'll notice that once all the validations pass, the **Register** button is enabled. On a tap of the button, a text gets printed in the console. In a real app, we would send the information to the server and register the user:

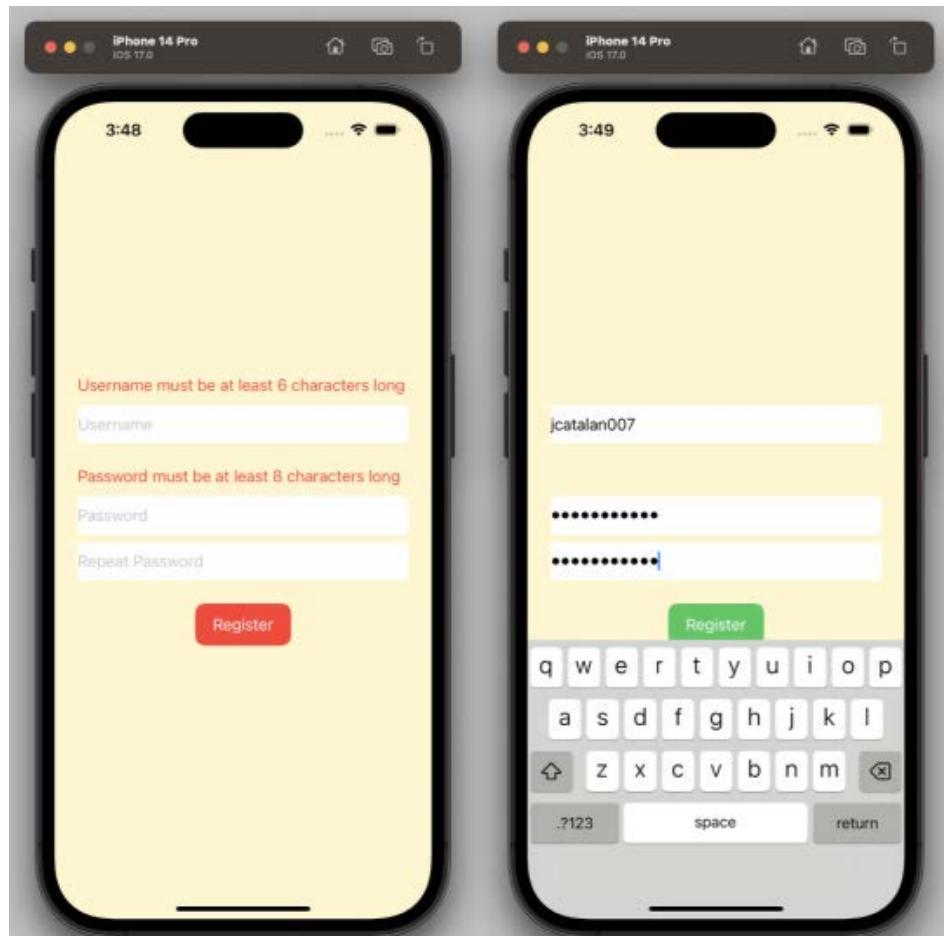


Figure 11.5: A reactive signup page

How it works...

This is the kind of problem where a Combine solution 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, where in 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(\.isStrongPassword)` function is a nice shortcut that was added in Swift 5.2 and it is equivalent to this:

```
.map { password in
    return password.isStrongPassword
}
```

- 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 you have 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 to 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.

In iOS 16, Apple introduced WeatherKit, a new Swift framework powered by the Apple Weather service, which includes a free tier with 500,000 API calls/month. However, to use WeatherKit you must be a member of the Apple Developer program, which has a yearly fee. Since we want this book to reach as many developers as possible, we opted to use OpenWeather, which does not have any fee to sign up for the free tier.

Getting ready

We will start by creating a SwiftUI app called Weather.

To use this service, we must create an account on **OpenWeather**:

1. Go to the **OpenWeather** signup page (https://home.openweathermap.org/users/sign_up) and fill in the form:

The screenshot shows the 'Create New Account' form on the OpenWeather website. At the top, there is a navigation bar with links: Guide, API, Dashboard, Marketplace, Pricing, Maps, Our Initiatives, and Partners. Below the navigation bar is the form itself, which has a light gray background. It contains four input fields: 'Username', 'Enter email', 'Password', and 'Repeat Password'. The 'Password' and 'Repeat Password' fields are side-by-side.

Figure 11.6: OpenWeather signup page

2. After confirming the login, you must create a new API key from the API keys page (https://home.openweathermap.org/api_keys):

The screenshot shows the 'API keys' page on the OpenWeather website. At the top, there is a navigation bar with links: Guide, API, Dashboard, Marketplace, Pricing, Maps, Our Initiatives, Partners, Blog, and For Business. Below the navigation bar is a search bar with the placeholder 'Weather in your city'. Underneath the search bar is a horizontal menu with links: New Products, Services, API keys (which is the active tab), Billing plans, Payments, Block logs, My orders, My profile, and Ask a question. A message box states: 'You can generate as many API keys as needed for your subscription. We accumulate the total load from all of them.' Below this message is a table showing existing API keys. The table has columns: Key, Name, Status, Actions, and Create key. One key is listed: 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx' with 'Default' as the name, 'Active' as the status, and two checkboxes in the Actions column. To the right of the table are input fields for 'API key name' and a 'Generate' button.

Figure 11.7: OpenWeather API keys

3. Note that you must 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 API endpoints to fetch the current weather and the 5-day weather 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:

1. Before we implement the objects, let's decide on the model we want to have and how to create it from the response we get from the service, as we will show in the next step. 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 response that's returned from the service. The response is encoded in the JSON data format:

```
{
//...
"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.

3. 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)
    }
}
```

4. Create the `forecast` struct as a simple list of the `Weather` records:

```
struct ForecastWeather: Decodable {
    let list: [Weather]
}
```

5. Now, let's move on to the `WeatherService` class, which conforms to the `ObservableObject` protocol. The purpose of this class is to connect to the weather service and fetch the data. We are exposing three variables: the current weather, the forecast, and a message in case 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:

```
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> = []  
}
```

6. 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)
}
```

7. Within 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)
}
```

8. 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)
    }
}
```

9. 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, we will create a function to translate the icon code from the weather service to an equivalent **SF Symbols** icon, starting with the icon to use during the daytime:

```
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"  
        }  
    }  
}
```

10. Now, let's add the icons for the forecast during the nighttime:

```
extension String {  
    var weatherIcon: String {  
        //...  
        case "01n":  
            return "moon"  
        case "02n":  
            return "cloud.moon"  
        case "03n":  
            return "cloud"  
        case "04n":  
            return "cloud.fog"  
    }  
}
```

```
        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"
    }
}
}
```

11. Create a `CurrentWeather` view to present the weather that's been fetched from 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)")
        }
    }
}
```

12. Each hourly forecast is presented in a `WeatherRow` view, 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)
}
}
```

13. 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 a `ContentView` struct 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)
        }
    }
}
```

14. 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)
            }
        }
    }
}
```

15. Finally, add the initial task to fetch the current weather and the hourly forecast when `ContentView` appears for the first time:

```
var body: some View {
    VStack {
        //...
    }
    .task {
        weatherService.load(latitude: 30.267222, longitude: -97.743056)
    }
}
```

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 11.8: 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, 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 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 emits a `Weather` object and `.receive(on: RunLoop.main)` moves the computation in `MainThread`, ready to be used for UI updates.

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 stored 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 like 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, combining different reactive streams of data, 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 data streams finish, and then creates a single object with both variables?

Finally, we hardcoded the coordinates, but how about using `CoreLocationManager`, 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 with delegates and callbacks. 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 reactive streams.

Getting ready

In this recipe, we will implement an app to showcase the debugging features of Combine. 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 simple publishers 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")
            .foregroundStyle(.white)
            .frame(width: 200, height: 50)
            .background(Color.green)
    }
    Button {
        reactiveObject.printDebug()
    } label: {
        Text("Print")
            .foregroundStyle(.white)
            .frame(width: 200, height: 50)
            .background(Color.orange)
    }

    Button {
        reactiveObject.breakPoint()
    } label: {
        Text("Breakpoint")
            .foregroundStyle(.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; it only uses Combine's `print()` function:

```
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:

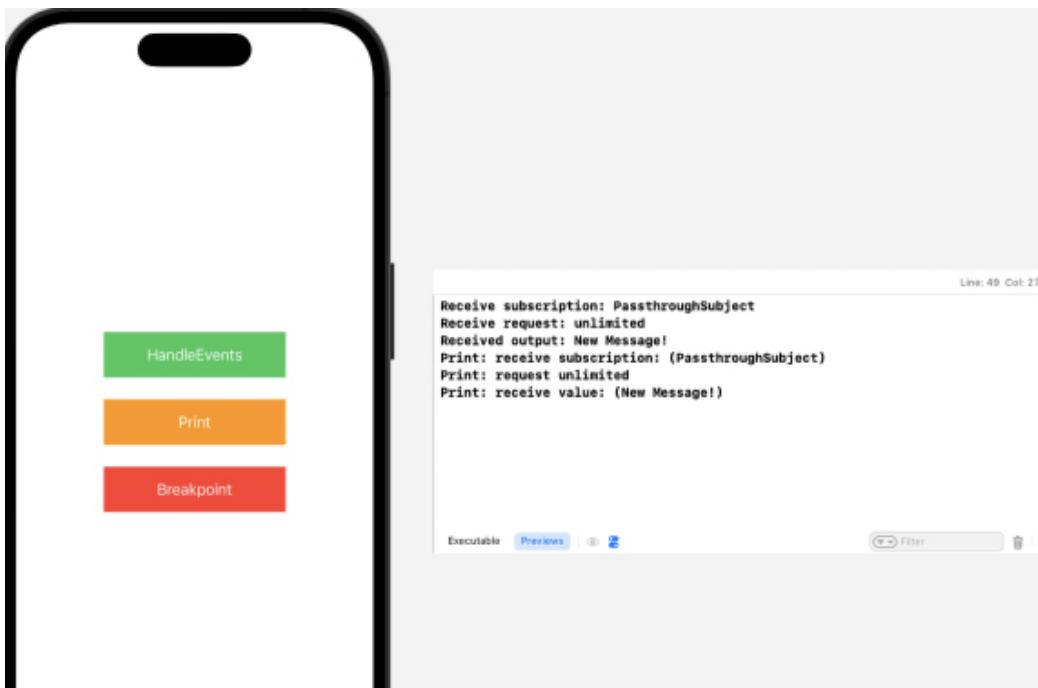


Figure 11.9: Debugging Combine handling and printing events

In the following events, the Xcode debugger stopped when we used the breakpoint strategy:

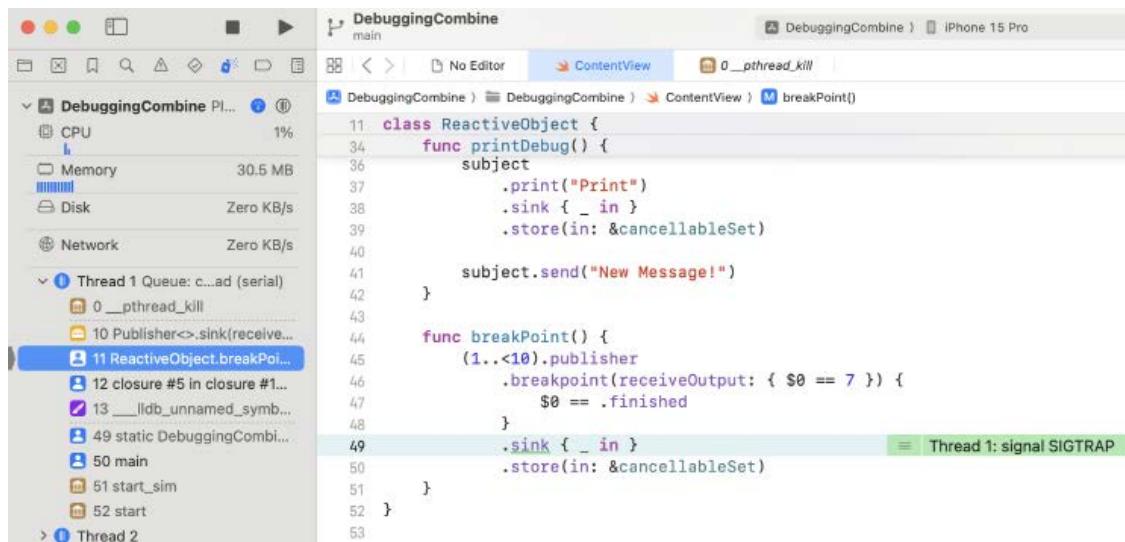


Figure 11.10: Breakpoint in a Combine app

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 than simple `print` statements, as we did in our code.

The `.print()` function is a Combine's own shortcut for the previous `.handleEvents()` function implementation, where every event is printed in the Xcode console with the prefix we pass. We can use it 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 created by *Marin Todorov* called **Timelane**, which allows you to follow the streams and present the results in Instruments. It is an advanced tool, but also sophisticated and flexible, and it's worth giving it a try.

It can be found at <http://timelane.tools/> and is also illustrated here:

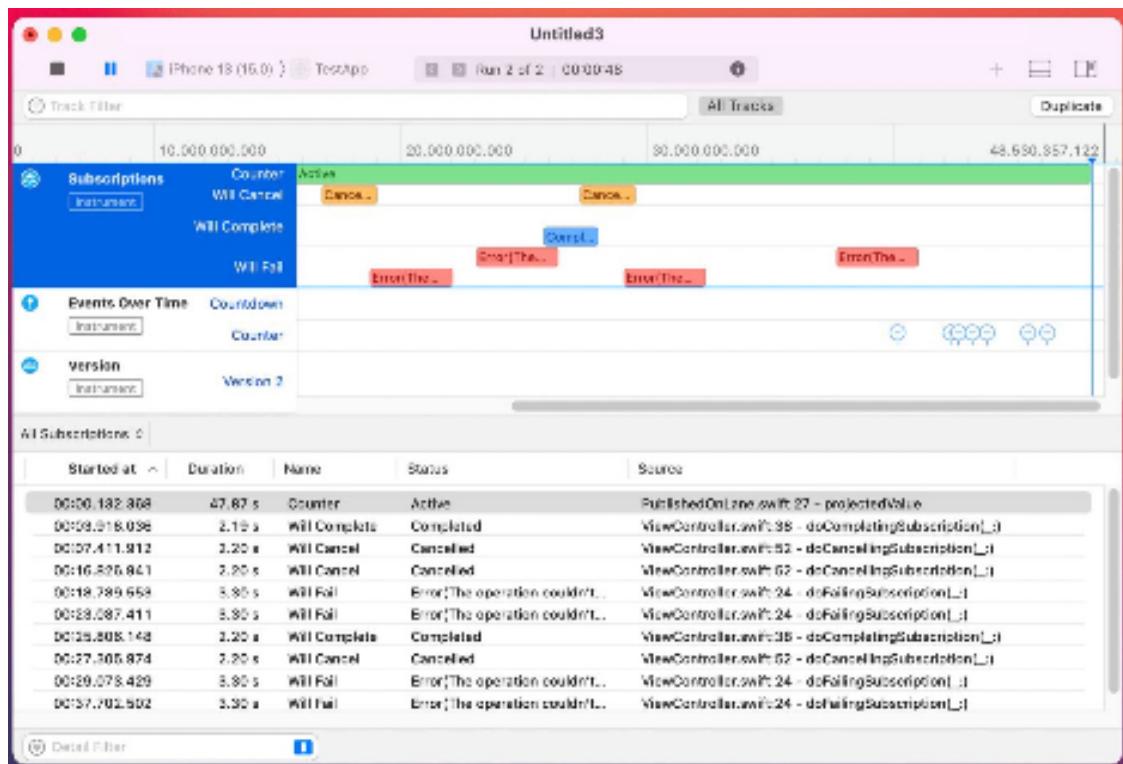


Figure 11.11: Timelane tool in action

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 like 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 though 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:

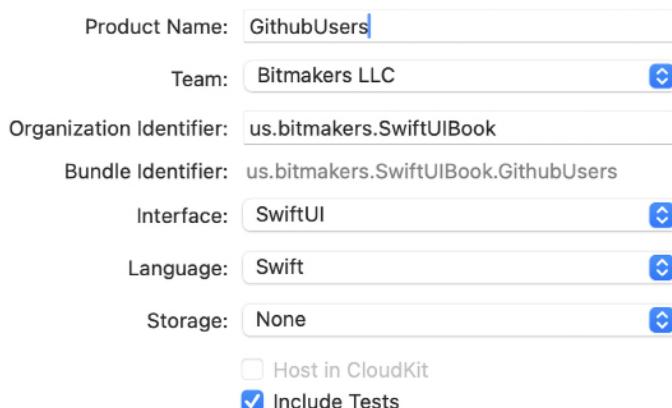


Figure 11.12: Creating a SwiftUI app with tests enabled

- 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-3rd-Edition/tree/main/Resources/Chapter11/recipe6>:

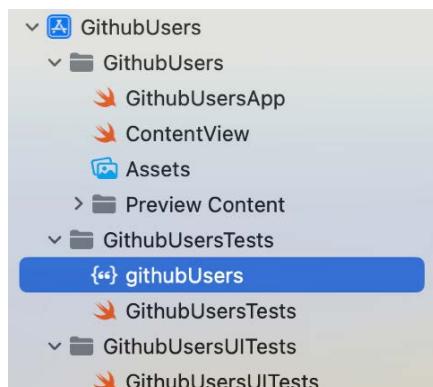


Figure 11.13: Fixture for the tests

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 users from `github.com`, creating a publisher from `URLSession`:

```
class Github: ObservableObject {
    @Published var users: [GithubUser] = []

    private var cancellableSet: Set<AnyCancellable> = []
    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 the View is presented:

```
struct ContentView: View {
```

```
@StateObject var github = Github()

var body: some View {
    List(github.users) {
        GithubUserView(user: $0)
    }
    .task { github.load() }
}
```

- 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:

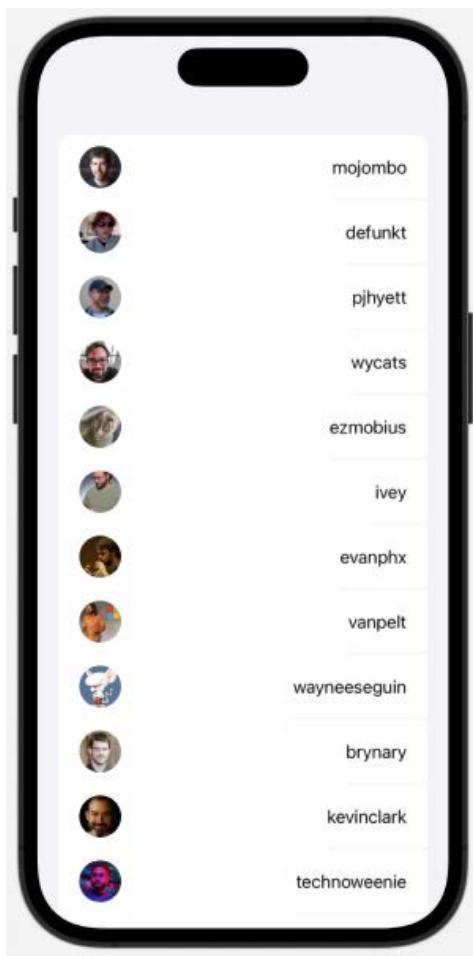


Figure 11.14: Fetching and presenting GitHub users

How can you be sure that it will always work when we add more features and change the code? The answer to this 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 `XCTestCase` 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 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 {
        true
    }

    override class func canonicalRequest(for request: URLRequest) ->
    URLRequest {
        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. 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 {
        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)
}

}
```

10. 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)
    }
}
```

Run the test by choosing the **Test** option from the **Product** menu in Xcode. From the Xcode navigator pane, click to show **Test Navigator** and you should see that the test we wrote is green, which means it passed successfully:

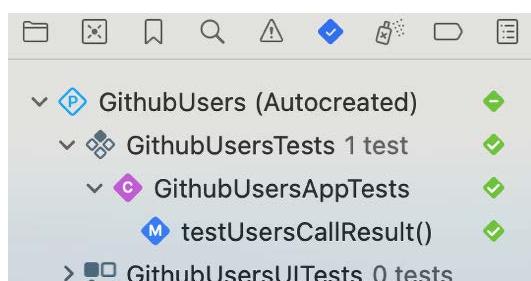


Figure 11.15: Test result

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 automated testing is something that should be part of the skill set of every professional developer.

The trick here is to realize that any variable decorated with the `@Published` property wrapper, is 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 })
```

The `exp1` expectation is not going to be fulfilled and the test will fail due to the time out on the `wait` statement:

```
wait(for: [exp1.expectation, exp2.expectation], timeout: 1)
```

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

<https://packt.link/swiftUI>



12

SwiftUI Concurrency with `async` `await`

One of the most important features of Swift 5.5, introduced in iOS 15, was the introduction of the `async` and `await` keywords. With `async` and `await`, we can write asynchronous concurrent code almost as if it were synchronous code, one statement after the other.

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 the 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 for 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 languages such as JavaScript, Kotlin, and C#, Swift provides `async` `await` to handle this scenario. An asynchronous function must be decorated with the `async` keyword and the calling function must add the `await` keyword 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 for transforming 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 15.0 and iOS 17.0. You can download and install the latest version of Xcode from the App Store. You'll also need to be running macOS Ventura (13.5) or newer.

Simply search for Xcode in the App Store and select and download the latest version. Launch Xcode and follow any additional installation instructions that your system may prompt you with. Once Xcode has fully launched, you're ready to go.

All the code examples for this chapter can be found on GitHub at <https://github.com/PacktPublishing/SwiftUI-Cookbook-3rd-Edition/tree/main/Chapter12-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 with 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, verify that the UI interaction is not blocked, and we can use a button to increase a counter shown in the view.

Getting ready

Create a 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 {  
        try? await Task.sleep(for: .seconds(5))  
    }  
}
```

```
        return "Result"
    }
}
```

2. In `ContentView`, add two `@State` properties, as shown in the following code:

```
struct ContentView: View {
    private let service = Service()
    @State private var value: String = ""
    @State private 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, the UI is responsive and we can change the value of the counter while waiting for the function to return the result:



Figure 12.1: SwiftUI and an `async` function

How it works...

In this recipe, we saw how to declare an asynchronous function: you just add the `async` keyword to the function declaration. The `Task.sleep()` function is asynchronous, so we must call it using the `await` keyword, as shown in the following code:

```
try? await Task.sleep(for: .seconds(5))
```

Apple introduced the `task(priority:_:)` modifier in iOS 15 to allow us to run an asynchronous task before the view appears. This is a very common scenario in many apps: the app starts up, makes a call to a remote server to retrieve some data, and once the data is retrieved, it gets transformed and displayed in the UI. Meanwhile, the app displays old data and shows some sort of progress indicator to give feedback to the user about the task being performed. We added the `task(priority:_:)` modifier to the `VStack` and made the call to the `async` function in its trailing closure. This modifier works like the `onAppear(perform:)` modifier with the difference that we can call asynchronous functions from its trailing closure:

```
.task {  
    value = await service.fetchResult()  
}
```

If we replace the `task(priority:_:)` modifier with the `onAppear(perform:)` modifier, the code won't compile anymore since in `.onAppear{}`, we can't call an `async` function:

```

22     Text("increment")
23   }
24 }
25 }
26 .onAppear {
27 }
28 }
29 }
30 }
  
```

Figure 12.2: A compile error when an `async` function is called from `.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:

Figure 12.3: 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:

Figure 12.4: 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 data transfer operations*. Fetching data from a network resource is an asynchronous operation by definition and until now, we had to manage it with the completion block mechanism.

iOS 15 added 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 `JSONPlaceholder` (<https://jsonplaceholder.typicode.com>) that provides a free fake API for testing purposes. The service has multiple resources with relations among them.

The app we'll implement will have a view with a list of fake users to choose from, and another view to present the different fake blog posts for the user we selected.

Getting ready

Create a SwiftUI app called `FakePosts`.

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 the Swift struct:

```
struct User: Decodable, Identifiable {
    let id: Int
    let name: String
    let email: String
    let phone: String
}

struct Post: Decodable, Identifiable {
    let id: Int
    let title: String
    let body: String
    let userId: Int
}
```

2. Create a `PostsService` struct to call the API service and decode the response into our model structs, which conform to the `Decodable` protocol:

```
struct PostsService {
    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 JSONDecoder().decode(type, from: data)
        } catch {
            return nil
        }
    }
}
```

3. Finally, add two `async` functions to the `PostsService` struct to call the `user` and `posts` endpoints:

```
struct PostsService {
    // ...
    func fetchUsers() async -> [User] {
        await fetch(type: [User].self,
                    from: "https://jsonplaceholder.typicode.com/users")
    ?? []
    }

    func fetchPosts(user: User) async -> [Post] {
        await fetch(type: [Post].self,
                    from: "https://jsonplaceholder.typicode.com/
posts?userId=\(user.id)") ?? []
    }
}
```

4. Add a `List` view in `ContentView` to present the fake users:

```
struct ContentView: View {
    private let service = PostsService()
    @State private var users: [User] = []

    var body: some View {
        NavigationStack {
            List(users) { user in
                NavigationLink(destination: PostsView(user: user)) {
                    VStack(alignment: .leading) {
                        Text(user.name)
                            .font(.title3)
                        Label(user.email, systemImage: "envelope")
                        Label(user.phone, systemImage: "phone")
                    }
                    .font(.footnote)
                }
            }
            .navigationTitle("Users")
        }
        .listStyle(.plain)
    }
}
```

5. To update the `users` `@State` property, add a `.task{}` modifier to `NavigationStack`:

```
// ...
var body: some View {
    NavigationStack {
        // ...
    }
    .listStyle(.plain)
    .task {
        users = await service.fetchUsers()
    }
}
```

6. Now implement a view to show the fake posts for the selected user. Create a `PostsView`, as shown in the following code:

```
struct PostsView: View {
    private let service = PostsService()
    @State private var posts: [Post] = []
    let user: User

    var body: some View {
        List(posts) { post in
            VStack(alignment: .leading) {
                Text(post.title)
                    .font(.title3)
                    .foregroundStyle(.blue)
                Divider()
                    .background(.blue)
                Text(post.body)
                    .font(.subheadline)
            }
        }
        .navigationTitle(user.name)
    }
}
```

7. Similarly, add a `.task{}` modifier to the `List` view, calling the `fetchPosts()` function:

```
List(posts) { post in
    // ...
}
.navigationTitle(user.name)
```

```
.task {
    posts = await service.fetchPosts(user: user)
}
```

The app is now ready, and we can test it by opening and selecting a user. Then, we can see the list of fake posts for that user:

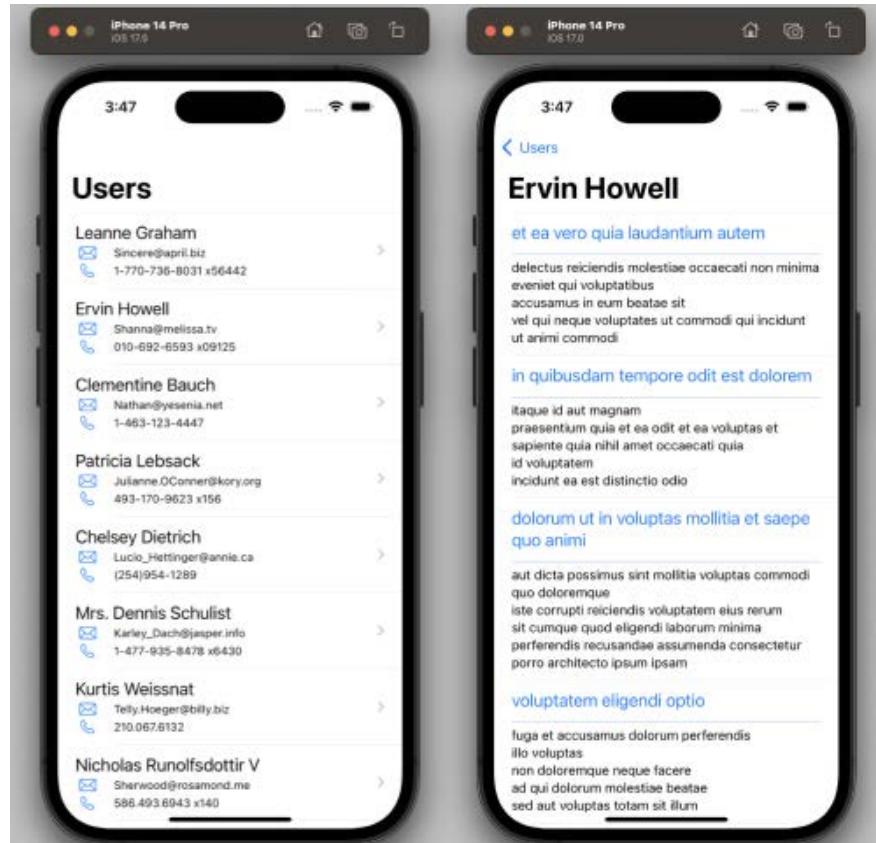


Figure 12.5: Selecting a user and seeing its fake posts

How it works...

Apple made a good attempt to extend all the block-based APIs in the Foundation framework with an `async await` counterpart. Prior to iOS 15, `NSURLSession` had the following interface:

```
NSURLSession.shared.dataTask(with: request) { 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 completion blocks.

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 right before the view appears and to update a `@State` property, triggering a re-evaluation of the 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 became part of the iOS SDK. Originally supported only in `UIKit`, iOS 15 introduced 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.

Upon 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 provides several types of random data, as you can see on 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

In Xcode create a new `SwiftUI` app called `RefreshableCrypto`.

How to do it...

As is 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 model 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`), which is the naming convention of the properties in the JSON payload, to *camel case* (`camelCase`), which is the naming convention used in Swift:

```
struct Service {  
    private let decoder: JSONDecoder = {  
        let decoder = JSONDecoder()  
        decoder.keyDecodingStrategy =  
            .convertFromSnakeCase  
        return decoder  
    }()  
  
    func fetchCoins() async -> [Coin] {  
    }  
}
```

3. `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`. We will add two properties, and in the `body` variable, we'll add a `List` view to present the coins in a `CoinView`. Replace the struct declaration with the following code:

```
struct ContentView: View {  
    private let service = Service()  
    @State private 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 cryptocurrency:

```
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)) { image in
            image.resizable()
                .aspectRatio(contentMode: .fit)
                .frame(maxWidth: 40, maxHeight: 40)
        } placeholder: {
            ProgressView()
        }
    }
}
```

7. 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:

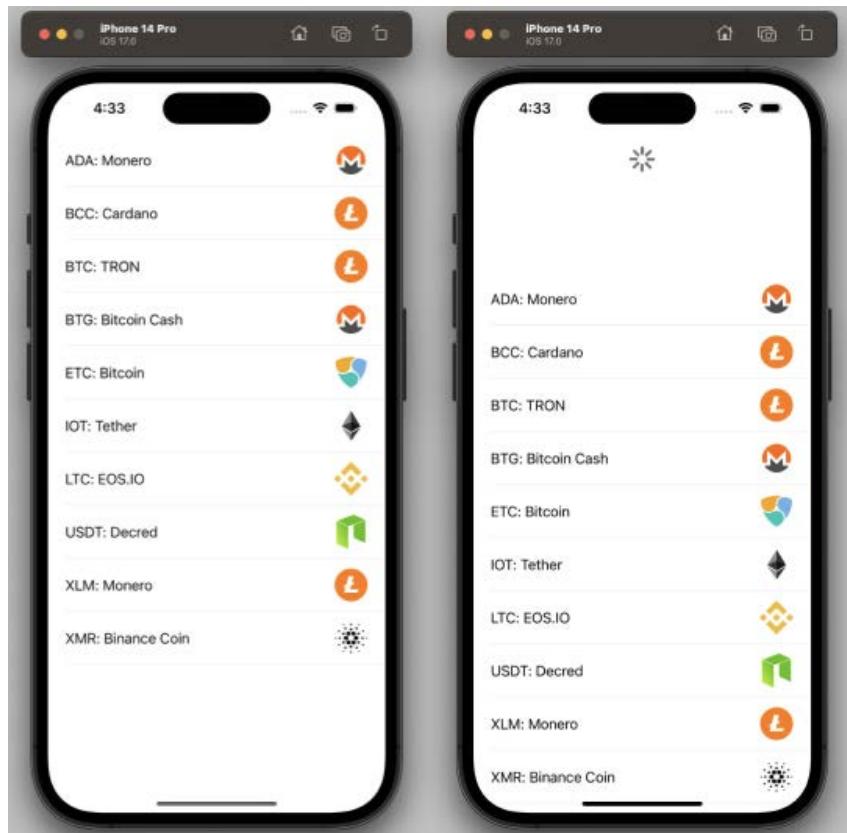


Figure 12.6: Pulling to refresh a list of crypto-currencies

How it works...

This is a 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)` asynchronous function, which is then called inside the `.task{}` modifier.

Finally, to add a pull-to-refresh control, we append the `.refreshable{}` modifier, which, 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 `@State` property `coins` 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 use, or even our legacy code, still has a completion block-based interface? Swift 5.5 introduced a simple way to convert block-based 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 to create a color theme for an app.

Getting ready

Create a SwiftUI app called `PaletteGenerator`.

From the Xcode menu, select the **File** option and then the **Add Package Dependencies** option to add the package from the following address: <https://github.com/gscalzo/Lorikeet>.

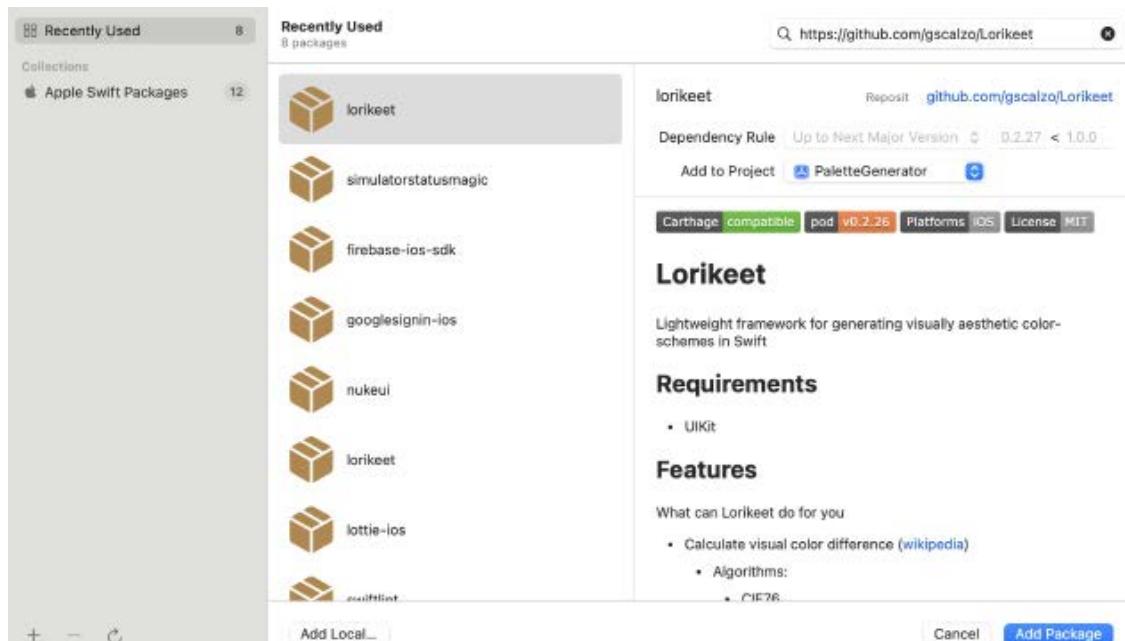


Figure 12.7: Importing the `Lorikeet` package

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 private 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 { description }  
}
```

- 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:

```
        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))
            }
        }
    }
}
```

- Finally, add a `.task{}` modifier to `ScrollView` to generate the colors when the view appears:

```
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 generated color scheme. It consists of 10 horizontal bars with different background colors. The first bar is the default blue color, and each of the other bars has a different color, as shown in the following image:

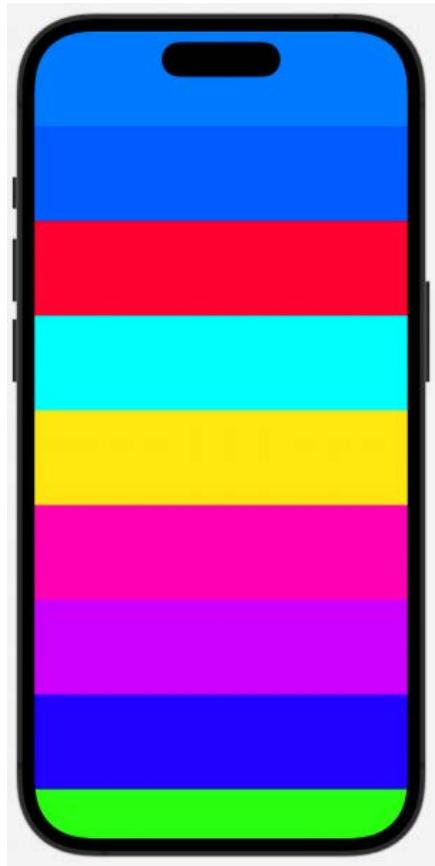


Figure 12.8: The generated color scheme

How it works...

Lorikeet provides a completion block function for generating a color scheme, with the following signature:

```
public func generateColorScheme(numberOfColors: Int,  
                                withRange range: HSVRange? = nil,  
                                using algorithm: Algorithm = .cie2000,  
                                completion: (([UIColor]) -> Void)? = nil)
```

If we used this completion block function in our code, the task modifier would have been:

```
.task {  
    Color.blue.lkt.generateColorScheme(numberOfColors: 10) { colors in  
        self.colors = colors.map(Color.init)  
    }  
}
```

We use the `withCheckedContinuation(function:_:)` function to transform the function provided by Lorikeet into an `async` function with the following signature:

```
func generateColorScheme(numberOfColors: Int,  
                        withRange: HSVRange,  
                        using: Algorithm) async -> [Color]
```

The `withCheckedContinuation` function suspends the current task, and then it calls the closure with a `CheckedContinuation` object.

Inside the closure, we call the completion block-based function, and when it finishes, we resume the execution of the task via the `CheckedContinuation` that `withCheckedContinuation` provided. 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 will be blocked forever. If we do it twice, the app will crash.

With this mechanism, we interface synchronous and asynchronous code. Thanks to this conversion, we used our custom `async` function in the task modifier:

```
.task {  
    colors = await Color.blue.lkt.generateColorScheme(numberOfColors: 10)  
}
```

In this example, we may not appreciate the advantage of using the asynchronous function over the completion block function. The real advantage comes when we have nested calls to complete block functions. In that case, we end up with what is known in computer programming as the *pyramid of doom*, like in this simplified example:

```
function1 {  
    function2 {  
        function3 {  
            function4 {  
                ...  
            }  
        }  
    }  
}
```

Using `async` functions, we convert the nested completion block functions in a sequence of calls to `async` functions:

```
await function1a  
await function2a  
await function3a  
await function4a  
...
```

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 `withCheckedContinuation`, as you can see in 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)
            }
        }
    }
}
```

In the trailing closure, we call the old function and, depending on its result, either we resume the continuation returning the value or resume the continuation throwing 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

And you can find out more about the pyramid of doom on Wikipedia here: [https://en.wikipedia.org/wiki/Pyramid_of_doom_\(programming\)](https://en.wikipedia.org/wiki/Pyramid_of_doom_(programming))

Implementing infinite scrolling with `async await`

Infinite scrolling is a technique that loads data when the scrolling reaches the end of a `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. We will use asynchronous functions to call the remote API with task modifiers on our views.

For this recipe, we will use a royalty-free image website called Pixabay (<https://pixabay.com/>). Pixabay provides a powerful RESTful API (<https://pixabay.com/api/docs/>), which we'll use to search and fetch a few images to present in a `List` view.

Getting ready

To use the Pixabay API, we need an API key:

1. First, sign up for an account on Pixabay. You can do this from the main page or from the API documentation page. Just click on the **Join** button and fill in the form and once the account is created, you'll be logged in.

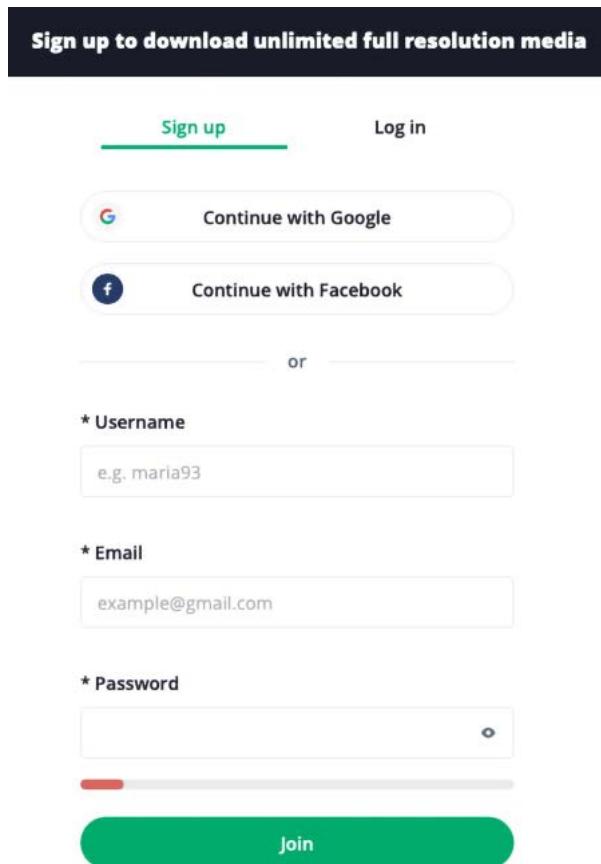


Figure 12.9: The Pixabay Sign up form

2. Navigate to the API documentation <https://pixabay.com/api/docs/>, where you can read about the API and get the API key. Scroll down to the section titled **Search Images**:

Search Images

<https://pixabay.com/api/> GET

Parameters

key (required)	str	Your API key: https://pixabay.com/api/
q	str	A URL encoded search term. If omitted, <i>all images</i> are returned. This value may not exceed 100 characters. Example: "yellow+flower"
lang	str	Language code of the language to be searched in. Accepted values: cs, da, de, en, es, fr, id, it, hu, nl, no, pl, pt, ro, sk, fi, sv, tr, vi, th, bg, ru, el, ja, ko, zh Default: "en"
id	str	Retrieve individual images by ID.
image_type	str	Filter results by image type. Accepted values: "all", "photo", "illustration", "vector" Default: "all"
orientation	str	Whether an image is wider than it is tall, or taller than it is wide. Accepted values: "all", "horizontal", "vertical" Default: "all"

Figure 12.10: The Pixabay API key

Once we have obtained the API key, we are ready to start working on the app. Create a new SwiftUI project called `ImageList`.

How to do it...

The app we are building fetches a set of images 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 images and the metadata returned by the API:

```
struct PixabayImage: Identifiable, Codable, Equatable {  
    let id: Int  
    let tags: String  
    let webformatURL: String  
    let user: String  
    var tagsList: [String] {  
        tags.components(separatedBy: " ", "  
    }  
}
```

```

struct PixabayResponse: Codable {
    let images: [PixabayImage]

    enum CodingKeys: String, CodingKey {
        case images = "hits"
    }
}

```

2. Add a `PixabayService` struct with a `private` function to build the request URL and an asynchronous function to fetch the images from the API. Replace the placeholder text `<YOUR_API_KEY>` with the API key you obtained from Pixabay:

```

struct PixabayService {
    private func searchImagesUrl(page: Int) -> URL? {
        let api = "https://pixabay.com/api/"
        let apiKey = "<YOUR_API_KEY>"
        let q = "waterfall"
        let query = "?key=\(apiKey)&image_"
        type=photo&orientation=horizontal&q=\(q)&per_page=10&page=\(page)"
        return URL(string: api + query)
    }

    func fetchImages(page: Int) async -> [PixabayImage] {
        guard let url = searchImagesUrl(page: page) else { return [] }
        do {
            let (data, _) = try await URLSession.shared.data(from: url)
            let response = try JSONDecoder().decode(PixabayResponse.self,
from: data)
            return response.images
        } catch {
            print(error)
            return []
        }
    }
}

```

3. Let's move to `ContentView`. Add a `List` view to present the images and some metadata:

```

struct ContentView: View {
    private let service = PixabayService()
    @State private var pixabayImages: [PixabayImage] = []
    @State private var page = 1

```

```
var body: some View {
    List(pixabayImages) { pixabayImage in
        VStack(alignment: .leading) {
            AsyncImage(url: URL(string: pixabayImage.webformatURL)) {
                image in
                image.resizable()
                    .aspectRatio(contentMode: .fit)
                    .clipShape(RoundedRectangle(cornerRadius:
                        CGSize(width: 10.0, height: 10.0)))
            } placeholder: {
                Image(systemName: "photo.artframe")
                    .resizable()
                    .foregroundStyle(.quaternary)
                    .aspectRatio(contentMode: .fit)
            }
        HStack {
            ForEach(pixabayImage.tagsList, id:\.self) { tag in
                Text(tag)
                    .font(.caption2)
                    .padding(7.0)
                    .foregroundStyle(.black)
                    .background(.yellow)
                    .clipShape(Capsule())
            }
            Spacer()
            Text("by \u2026\ufe0f(pixabayImage.user) via Pixabay")
                .font(.caption2)
                .italic()
        }
    }
    .task {
        if pixabayImage == pixabayImages.last {
            page += 1
            pixabayImages += await service.fetchImages(page:
page)
        }
    }
}
.listStyle(.plain)
```

```
        .task {
            pixabayImages = await service.fetchImages(page: page)
        }
    }
}
```

- When running the app, we can see the images, loading a page at a time:

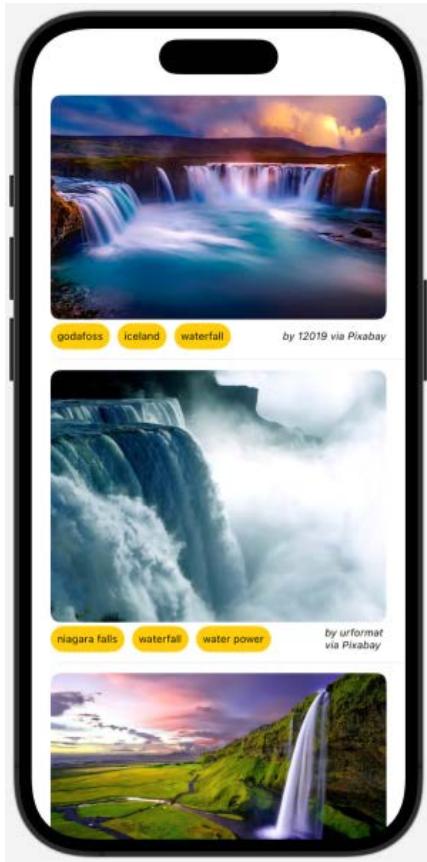


Figure 12.11: Images of waterfalls around the world

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 list row, but we trigger a call to the service only if the row is the last row of the current set of loaded images.

On the `searchImagesUrl(page:)` function modify the value of the `q` variable with a different search term and you'll get a different set of images. I chose to get images of waterfalls, but feel free to choose another word for the image search.

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>.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

<https://packt.link/swiftUI>



13

Handling Authentication and Firebase with SwiftUI

Since the creation of the mobile app market, one of the most important features that most apps utilize is authentication. The normal way in which app creators handle the authentication 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. 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 the password for that app has been used elsewhere, all 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, have 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 will still know that we are using that app and when we are logging 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.

Privacy is one of the core values of Apple. iOS provides the **Sign in with Apple** service, which authenticates a user in a secure way while respecting the user's privacy. 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 users, 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 leverage this nice and 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 will 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 that 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 users with Google Sign-In
- Implementing a Notes app with Firebase and SwiftUI

Technical requirements

The code in this chapter is based on Xcode 15.0 and iOS 17.0. You can download and install the latest version of Xcode from the App Store. You'll also need to be running macOS Ventura (13.5) or newer.

Simply search for Xcode in the App Store and select and download the latest version. Launch Xcode and follow any additional installation instructions that your system may prompt you with. Once Xcode has fully launched, you're ready to go.

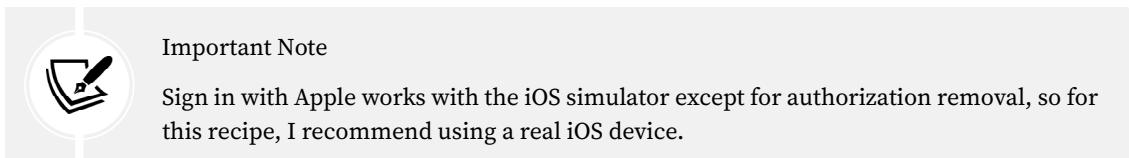
All the code examples for this chapter can be found on GitHub at <https://github.com/PacktPublishing/SwiftUI-Cookbook-3rd-Edition/tree/main/Chapter13-Handling-authentication-and-Firebase-with-SwiftUI>

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 works with the iOS simulator except for authorization removal, 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 `SwiftUI@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 instead.
- 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 in this recipe.

Now, let's start implementing 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:

- 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**, as shown in the following figure:

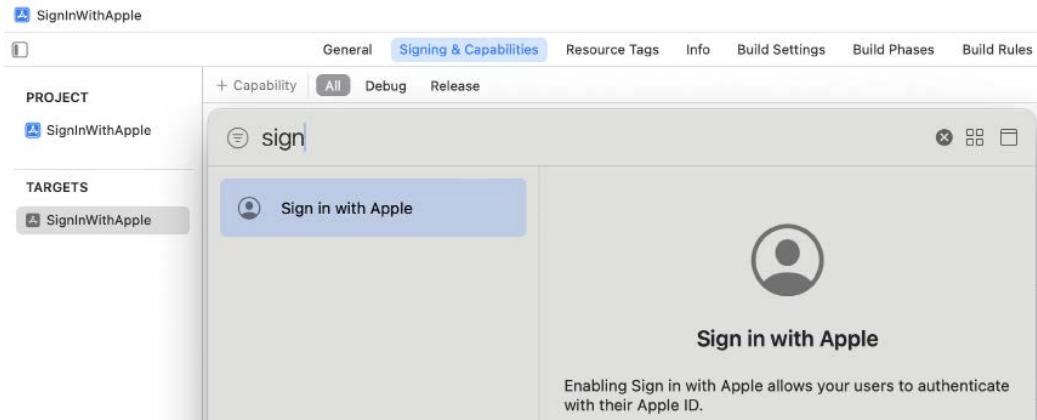


Figure 13.1: Searching for the Sign in with Apple capability

- After selecting **Sign in with Apple**, check that there is a new entry in the project, indicating the newly added capability, as shown in the following figure:



Figure 13.2: 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 has logged in, or the **SignInWithApple** button when the user starts the app for the first time:

```
import SwiftUI
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. We can use closures or functions for the callbacks. Add the button and two callback functions 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: ASAuthorizationAppleIDRequest) {  
        }  
  
        private func onCompletion(_ result: Result<ASAuthorization, Error>) {  
        }  
    }  
}
```

5. Implement the `onRequest` function, where we ask the framework to return the full name and email address of the user:

```
private func onRequest(_ request: ASAuthorizationAppleIDRequest) {
```

```
    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 three `@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?
                ASAAuthorizationAppleIDCredential
            else { return }
            storedName = credential.fullName?.givenName ?? ""
            storedEmail = credential.email ?? ""
            userID = credential.user
        case .failure (let error):
            print("Authorization failed: " + error.localizedDescription)
    }
}
```

8. 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 = ""  
    }  
}  
}
```

Use your iPhone to run the app and your Apple ID to sign in, as shown in the following figure:

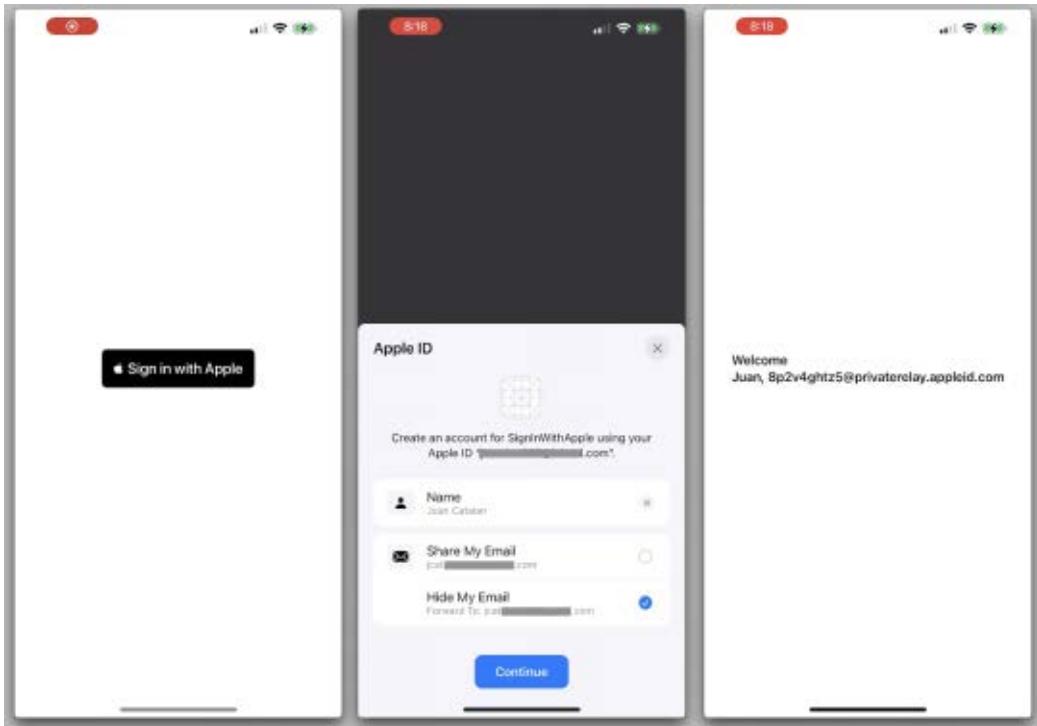


Figure 13.3: Sign in with Apple in action

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 whether 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 `ASAuthorizationAppleIDProvider()` 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 sign-out process can't be done inside the app. To sign out from our app, the user must disable the credentials provided in the iOS **Settings** app, in the **Sign-In & Security** section of their Apple ID profile, as shown in the following screenshot (iOS 17 version):

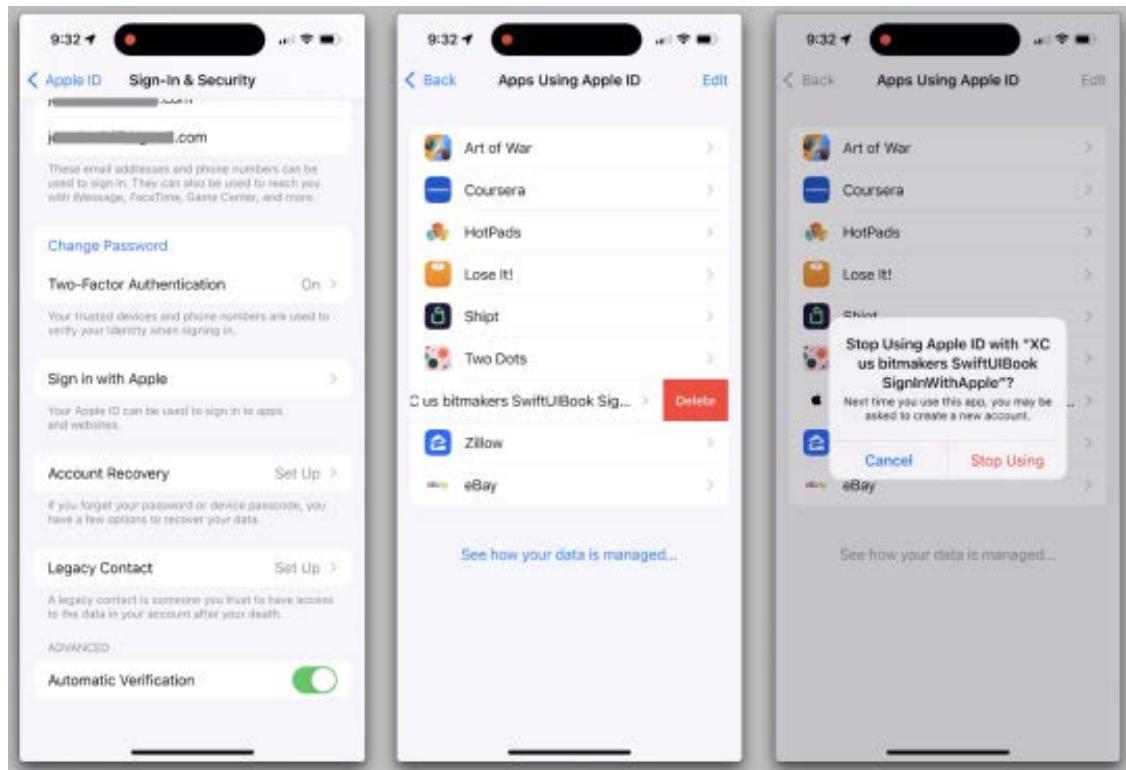


Figure 13.4: Removing credentials for Apple ID sign in

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 cloud services needed to power their app.

Firebase has a native SDK for iOS and because it has been around for quite some time, is largely written in Objective-C, although it has some code written in Swift and even in SwiftUI. Google is making efforts to update the SDK and newer APIs are written in Swift and SwiftUI. Google even announced that starting on January 1, 2024, they will remove Objective-C code snippets from the official website and focus on Swift code snippets. At the time of this writing, the Firebase iOS SDK supports Objective-C, Swift, and SwiftUI. SwiftUI has some known issues documented on the official Firebase website, but with the correct configuration, it will work with SwiftUI. Firebase is a sophisticated service and you'll need to invest some time in learning its features and configurations. However, it is easier to use Firebase in your apps rather than implementing a backend service from scratch.

We'll start our exploration of Firebase in SwiftUI by integrating its Remote Config product. Think of Remote Config as a UserDefaults service in the cloud. By storing default values in Remote Config and retrieving them when the app starts, we can customize our app without having to release an update to the App Store. The SDK even allows us to get real-time updates from the Firebase service as soon as changes are published.

In this recipe, we are going to implement two sample main screens, so that we can configure remotely which one to present to the users. We will use one key-value pair in Remote Config to achieve our result.

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:

1. Firstly, add to the project a new **Swift Package Manager (SPM)** package. Use the **Add Package Dependencies** option from Xcode's **File** menu. When the modal screen appears, 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, as shown in the following figure:

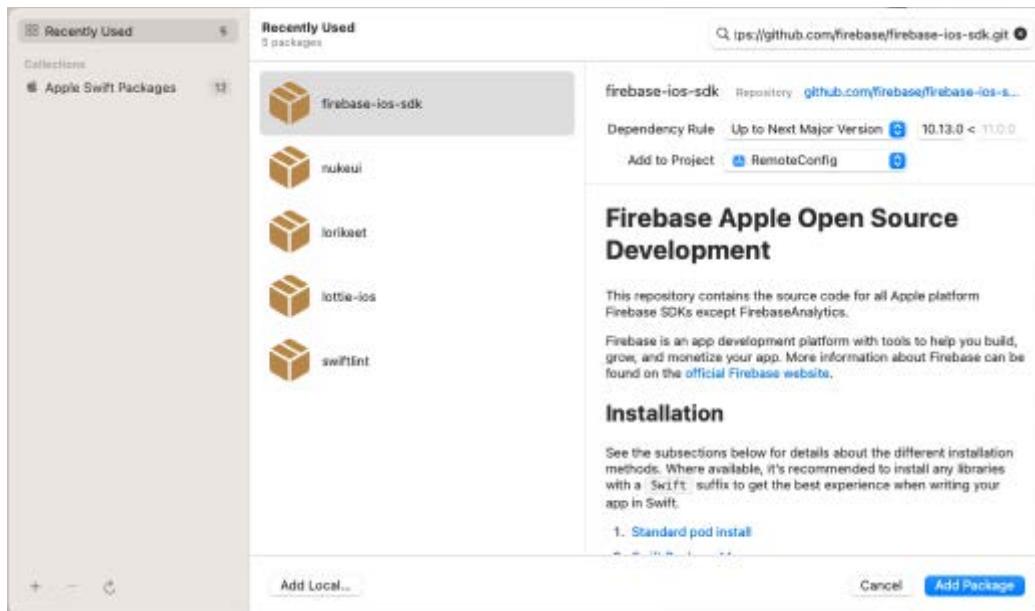


Figure 13.5: Adding the Firebase package

2. In the next step, click on the Add Package button to add Firebase to the project:

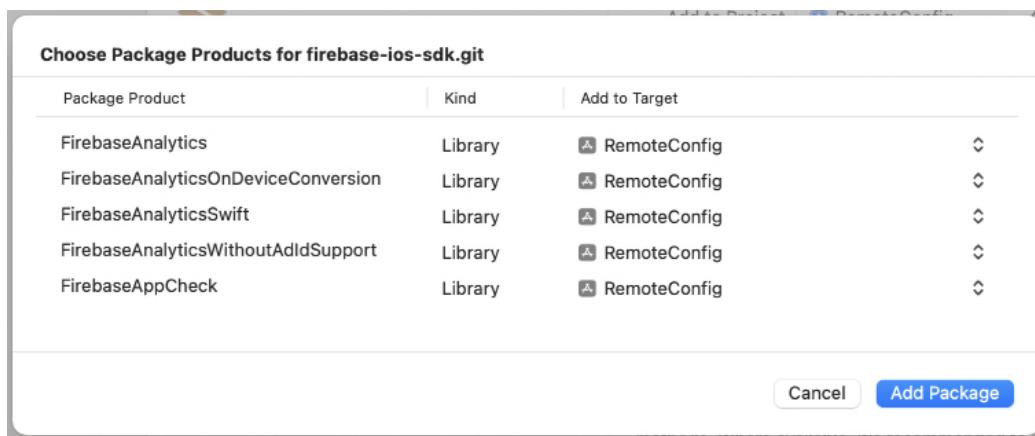


Figure 13.6: Selecting the two Firebase sub-packages

3. After adding the package, our project should look like the one in the following figure:

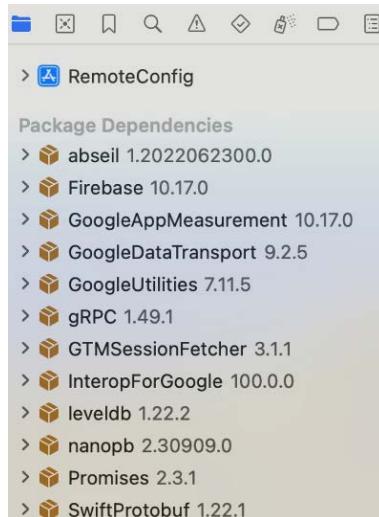


Figure 13.7: The RemoteConfig project with the Firebase packages

How to do it...

In this recipe, we'll encounter more configuration steps 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 official Firebase website, <https://firebase.google.com/>. After logging into the website with a valid Google account, click on the **Get started** button to be taken to the console, as shown in the following figure:

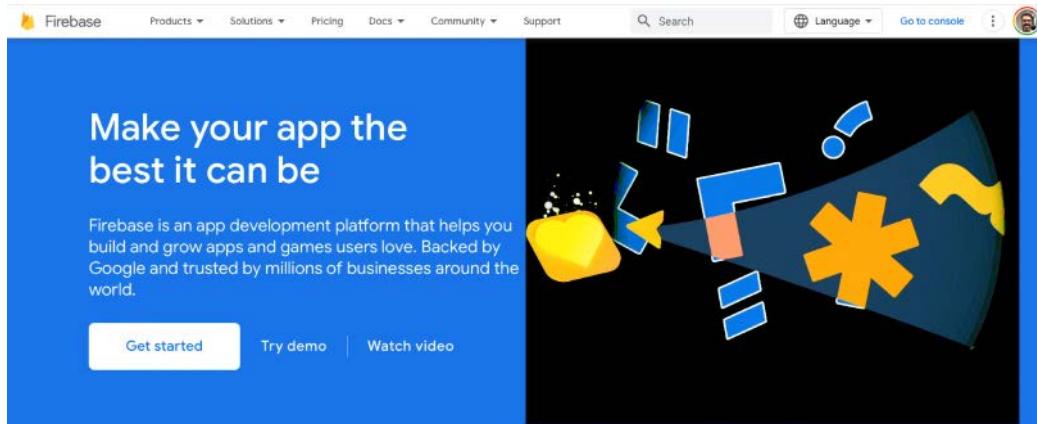


Figure 13.8: The official Firebase website

2. From the console, create a new project by selecting **Add project**:

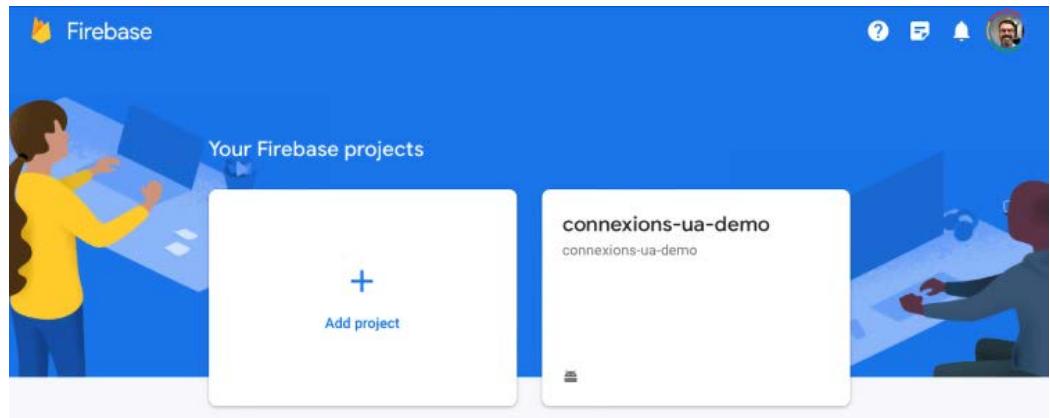


Figure 13.9: The Firebase console

3. Choose a name for the project, for example, `FirebaseRemoteConfigApp`:

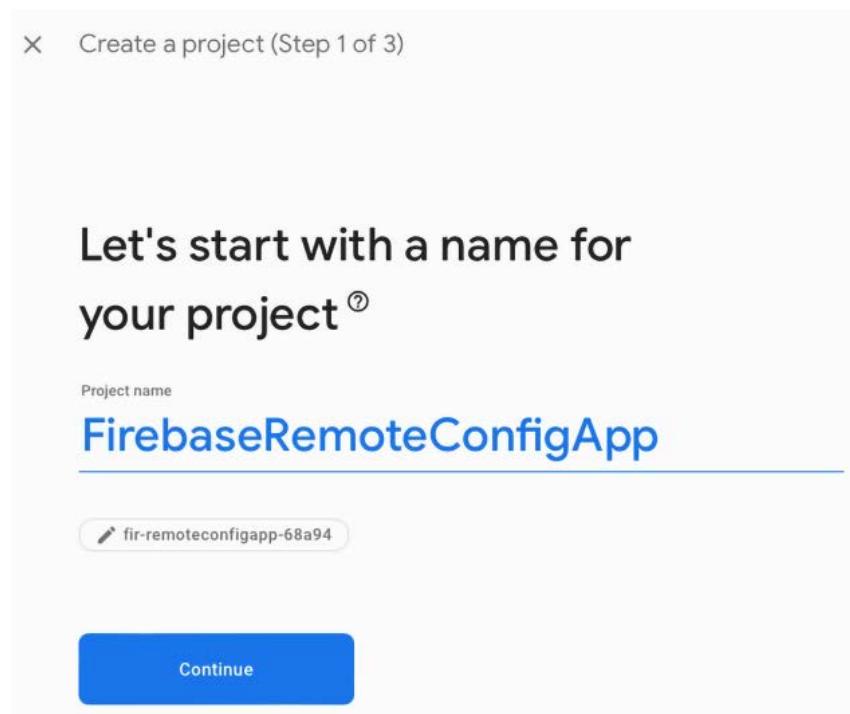


Figure 13.10: Creating a new Firebase project

4. On the next screen, disable *Google Analytics* since we don't need it for this recipe:

×

Create a project (Step 2 of 2)

Google Analytics for your Firebase project

Google Analytics is a free and unlimited analytics solution that enables targeting, reporting, and more in Firebase Crashlytics, Cloud Messaging, In-App Messaging, Remote Config, A/B Testing, and Cloud Functions.

Google Analytics enables:

- ×
- A/B testing ⓘ
- ×
- Event-based Cloud Functions triggers ⓘ
- ×
- User segmentation & targeting across ⓘ
- Firebase products
- ×
- Free unlimited reporting ⓘ
- Crash-free users ⓘ

Enable Google Analytics for this project
Recommended

Previous

Create project

Figure 13.11: Disabling Google Analytics

5. Once you have created the project, it is time to configure the **Remote Config** service. Expand the **Engage** section, select the **Remote Config** option, and click on the **Create configuration** button. Then, add a parameter named `screenType`, of type `String`, whose default value is `screenA`, as shown in the following figure:

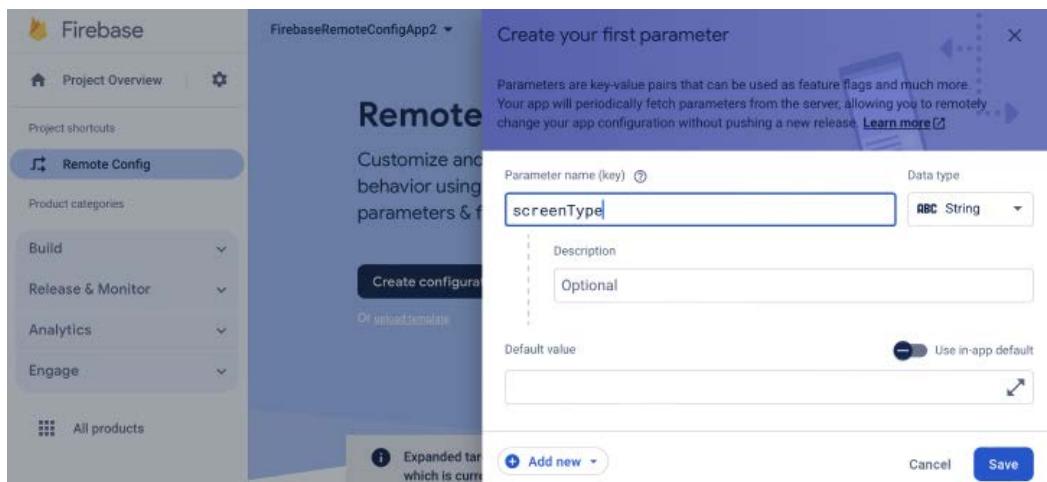


Figure 13.12: Adding a Remote Config parameter

- After saving the parameter, we can publish it as follows:

The screenshot shows the Firebase Remote Config dashboard. It displays a published parameter named 'screenType' of type 'String' with a value 'screenA'. The dashboard also includes sections for 'Fetches - Last 24 Hours', 'A/B Testing', and 'Personalization'.

Name	Condition	Value	Fetch %	Last published
screenType		screenA		

Figure 13.13: Remote Config parameter added

7. Publishing these changes makes them immediately available to our users:

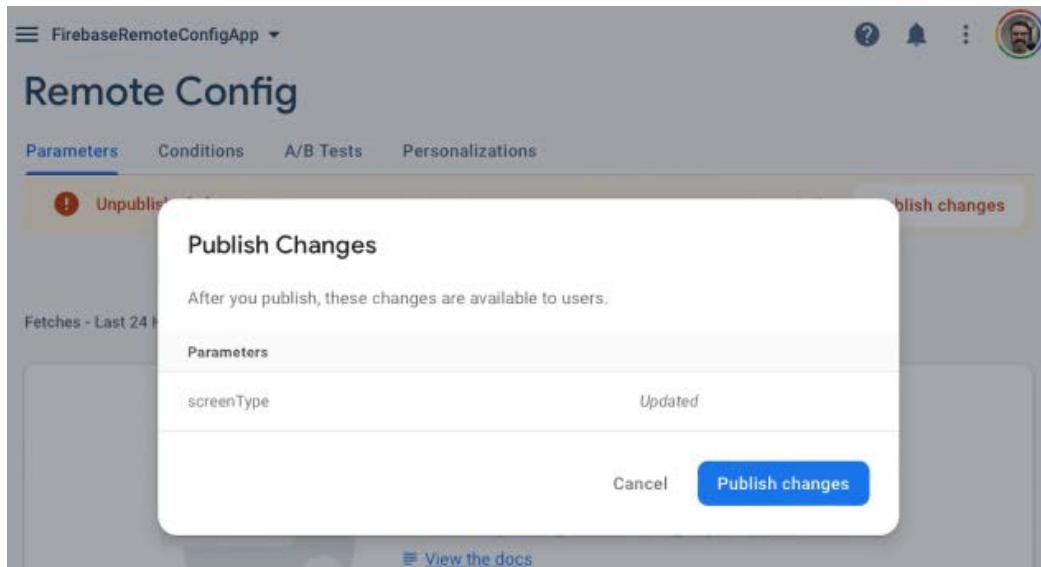


Figure 13.14: Publishing our parameter

8. 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 13.15: Configuring iOS for Firebase

9. Register the app using our Apple Bundle ID as shown in the following figure:

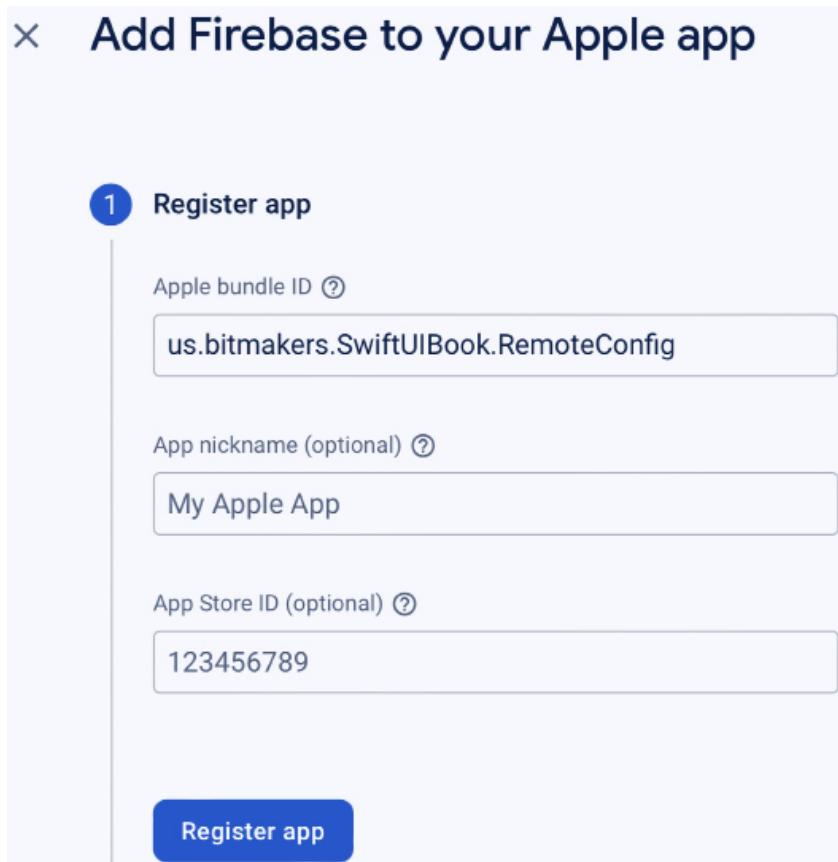


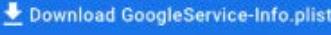
Figure 13.16: Registering the Apple bundle ID

10. The registration of the app with Firebase generates a configuration file, `GoogleService-Info.plist`, which must be imported into the Xcode project. To do this, first, download the file using the button:

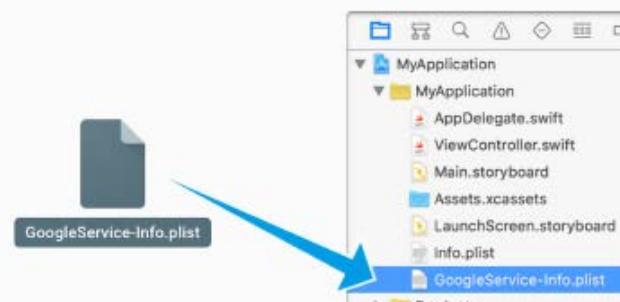
x Add Firebase to your Apple app

✓ Register app
Apple bundle ID: us.bitmakersSwiftUIBook.RemoteConfig

2 Download config file Instructions for Xcode below | Unity C++  

 Download GoogleService-Info.plist

Move the `GoogleService-Info.plist` file you just downloaded into the root of your Xcode project and add it to all targets.



Next

3 Add Firebase SDK

Figure 13.17: `GoogleService-Info.plist` file

11. Once you've downloaded the file, drag it into the project, taking care to select **Copy items if needed** as depicted in the following figure:

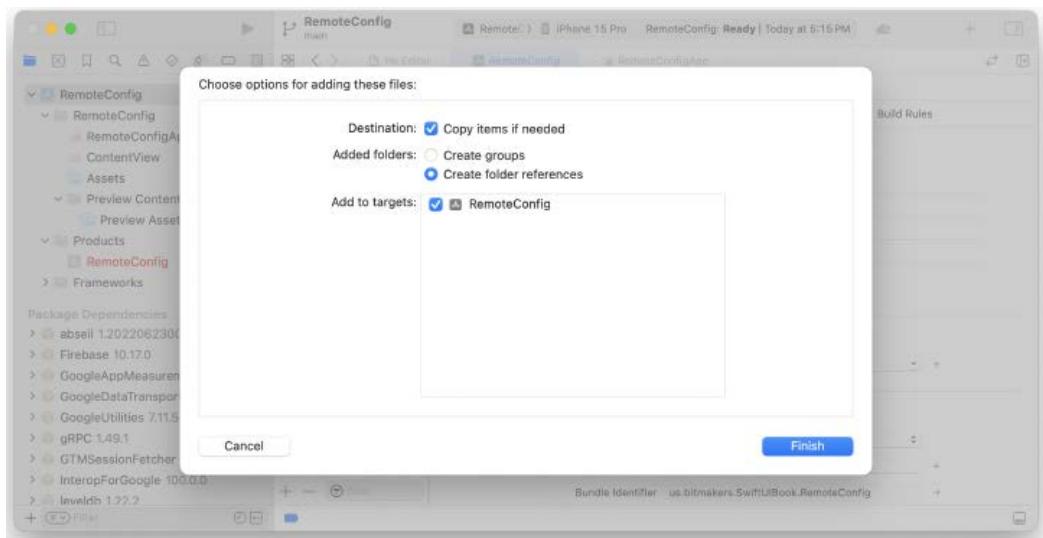


Figure 13.18: Importing the GoogleService-Info.plist file

12. This is how our project should appear in Xcode:

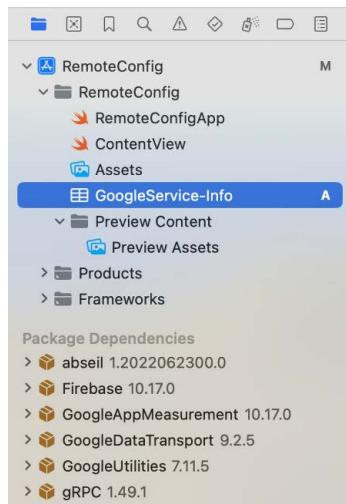


Figure 13.19: Imported GoogleService-Info.plist file

13. Now, let's work on the code. The first thing we must do is to configure Firebase in the **RemoteConfigApp** main struct:

```
import SwiftUI
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 the `Remote Config` parameter:

```
import SwiftUI
import Firebase
import FirebaseRemoteConfigSwift

struct ContentView: View {
    private var config = RemoteConfiguration()
    @RemoteConfigProperty(key: "screenType", fallback: nil) var screenType: String?

    private var image: (name: String, color: Color) {
        if screenType == "screenA" {
            ("a.square", .green)
        } else if screenType == "screenB" {
            ("b.square", .blue)
        } else {
            ("questionmark.square", .red)
        }
    }

    var body: some View {
        VStack {
            if screenType != nil {
                Image(systemName: image.name)
                    .foregroundStyle(image.color)
                    .font(.system(size: 250))
            } else {
                ProgressView()
                    .controlSize(.large)
            }
        }
    }
}
```

15. Adding the `RemoteConfiguration` class after the `ContentView` struct closing brace is a custom class, which is a simple wrapper around the Firebase Remote Config service:

```
class RemoteConfiguration {  
    private var remoteConfig = Firebase.RemoteConfig.remoteConfig()  
}
```

16. In the `init()` function, enable *Developer Mode* for the Firebase Remote Config service:

```
init() {  
    // Enable developer mode  
    let settings = RemoteConfigSettings()  
    settings.minimumFetchInterval = 0  
    remoteConfig.configSettings = settings  
}
```

17. Add a `private` function, `activate()`, which calls the `activate(completion:)` function of the Firebase module, which updates the local values with the remote values fetched from the Remote Config service. The Firebase module offers two functions, one asynchronous and the one that we are using here, which has a completion block:

```
private func activate() {  
    remoteConfig.activate { changed, error in  
        guard error == nil else {  
            print("Error activating Remote Config: \(error!.localizedDescription)")  
            return  
        }  
        print("Default values were \(changed ? "" : "NOT ")updated from  
        Remote Config")  
    }  
}
```

18. Add a `fetchFromServer()` `async` function to fetch the key-value pairs from the Remote Config service. In this case, we use the corresponding `async` function of the Firebase module. It is very important to note that we need to call the `activate()` function to update the local values with the remote values, otherwise, nothing will change:

```
func fetchFromServer() async {  
    guard let status = try? await remoteConfig.fetch(), status ==  
.success else {  
        print("Couldn't fetch Remote Config")  
        return  
    }  
    print("Remote Config successfully fetched")
```

```
        activate()  
    }
```

19. Add a `registerForRealtimeUpdates()` function to register so as to receive real-time updates when the remote config changes. Unfortunately, the Firebase module only provides an Objective-C function with a completion block. Notice how, after new remote values have been fetched, we again call the `activate()` function to update the local values:

```
func registerForRealtimeUpdates() {  
    print("Registering for Remote Config realtime updates")  
    remoteConfig.addOnConfigUpdateListener { [self] update, error in  
        guard let update, error == nil else {  
            print("Error listening for Remote Config realtime updates: \(error!.localizedDescription)")  
            return  
        }  
        print("Updated keys in realtime: \(update.updatedKeys)")  
        activate()  
    }  
}
```

20. Finally, add a `.task()` modifier to the `ContentView` view, and call the two public methods of our custom class, one to fetch the initial values from the Remote Config service and the other to register for real-time updates:

```
struct ContentView: View {  
    // ...  
    var body: some View {  
        VStack {  
            //...  
        }  
        .task {  
            await config.fetchFromServer()  
            config.registerForRealtimeUpdates()  
        }  
    }  
}
```

If you go to the Firebase console to change the value of the parameter and publish these changes, this will be reflected in the app. If you experiment with the real-time updates, make sure the service is activated in your Firebase console, as explained in the Firebase docs. Use the live preview on the canvas or run the app in the iOS simulator. The results should be similar to the following screenshots:



Figure 13.20: Selecting the View to show depending on the Remote Config value

How it works...

Firebase is a 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 `fetchFromServer()` function in the `.task()` modifier of the main view. This function retrieves the default values from the remote server and then with the call to the `activate()` function, the local values get updated. The update on the local values changes the `screenType` property. We use a property wrapper included in the `FirebaseRemoteConfigSwift` module:

```
@RemoteConfigProperty(key: "screenType", fallback: nil) var screenType: String?
```

This property wrapper is an example of a Firebase API, which is written in Swift and uses SwiftUI. Under the hood, the property wrapper includes a `@Published` variable, where the value received from the Remote Config server is stored. Any change to the wrapped value will be published, and since we use the variable in the body of our `ContentView` view, SwiftUI will trigger a new rendering automatically.

There's more...

Firebase Remote Config is very powerful, and we have covered some of its features in this recipe. I encourage you to explore the Firebase Remote Config documentation on the official Firebase website and learn how and when to use this powerful service in your apps.

See also

- Firebase support for SwiftUI: <https://firebase.google.com/docs/ios/learn-more#swiftui>
- Firebase announcement about removing Objective-C code snippets on January 1, 2024: https://firebase.google.com/docs/ios/learn-more#ongoing_support_for_objective-c
- Firebase Remote Config: <https://firebase.google.com/docs/remote-config>
- Firebase iOS SDK: <https://github.com/firebase/firebase-ios-sdk>

Using Firebase to sign in users with Google Sign-In

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 sign in users to a Firebase app. Instead of handling our own user names and passwords, we will integrate a social login with Google. Once our user is authenticated with Google, we will then sign the user in to our Firebase app using a token provided by Google. This is a very secure way to sign in our users, respecting their privacy as well. Once the user is logged in, we will be able to provide a personalized user experience.



Important Note

If we implement any social login in our app, we must implement Sign in with Apple too. This is a requirement from Apple, and it will be reviewed by Apple before the app is approved for App Store distribution.

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 using the `https://github.com/firebase/firebase-ios-sdk.git` URL, as shown in the figure:

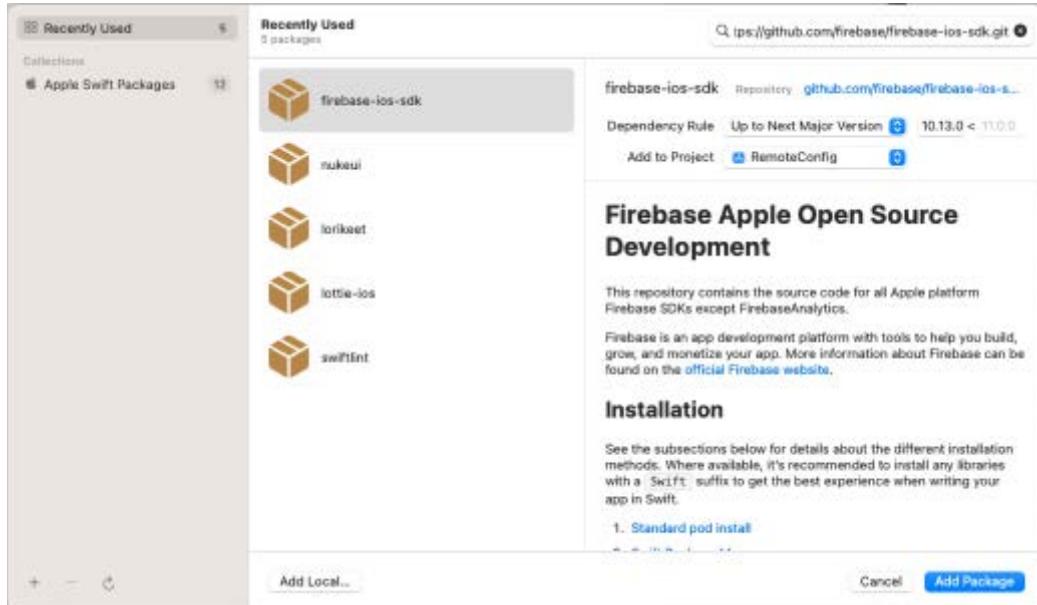


Figure 13.21: Adding the Firebase package

2. Then select the `FirebaseAuth` sub-package, as shown:

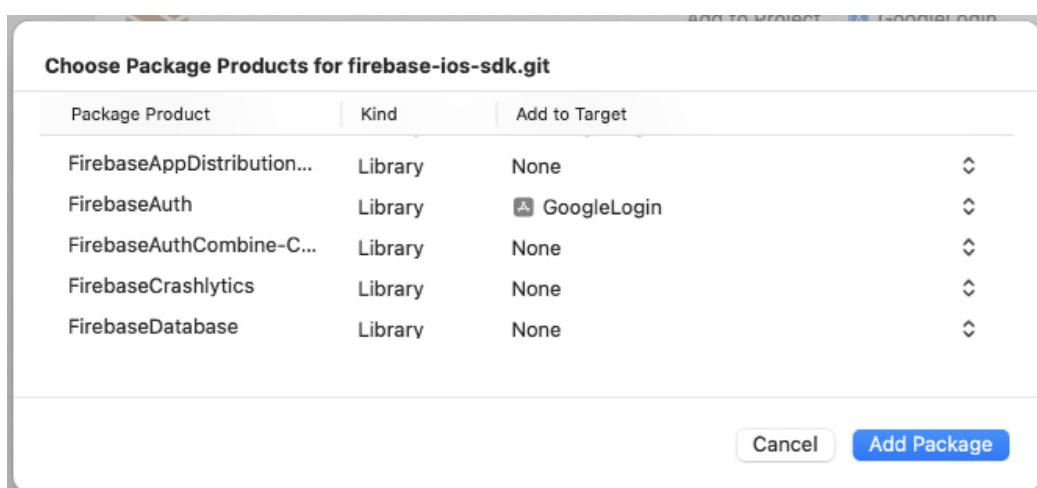


Figure 13.22: Adding the FirebaseAuth sub-package

3. Add the GoogleSignIn-iOS package, and set the URL to <https://github.com/google/GoogleSignIn-iOS>:

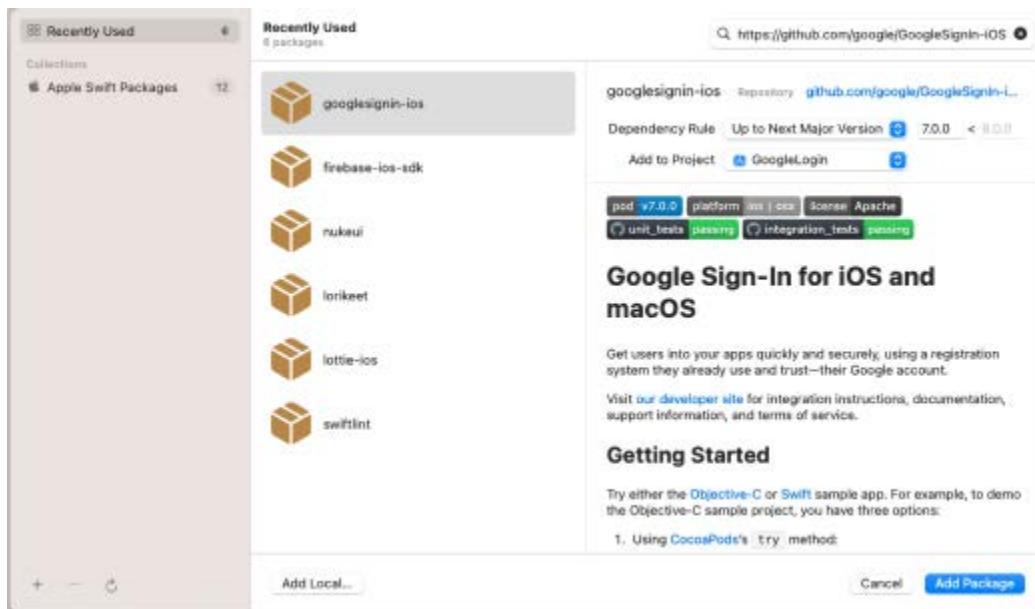


Figure 13.23: Adding the GoogleSignIn SPM package

4. After this, select both sub-packages:

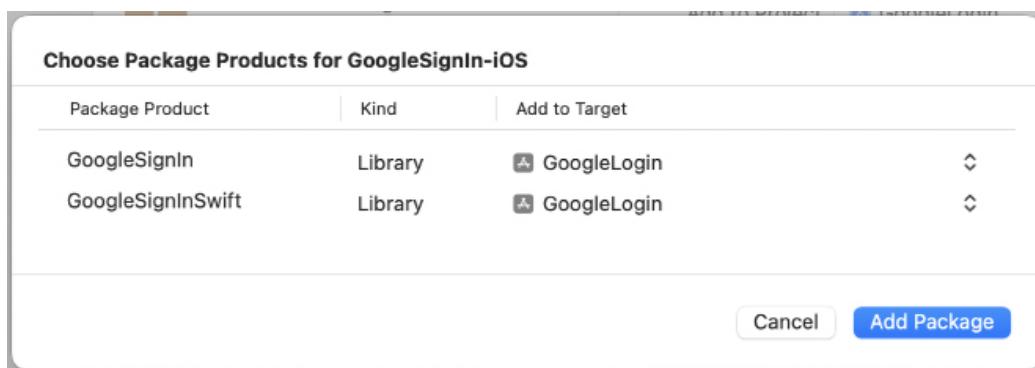


Figure 13.24: Adding the FirebaseAuth sub-package

5. The FirebaseAuth module uses App Delegate swizzling. Due to a known issue with swizzling, SwiftUI applications must disable Method Swizzling for Firebase. We can disable swizzling in our app by adding to the `Info.plist` file the `FirebaseAppDelegateProxyEnabled` entry of type Boolean with a value of NO. To add this entry, select the project file in Xcode, then the app target, and the `Info` tab, as in the following figure:

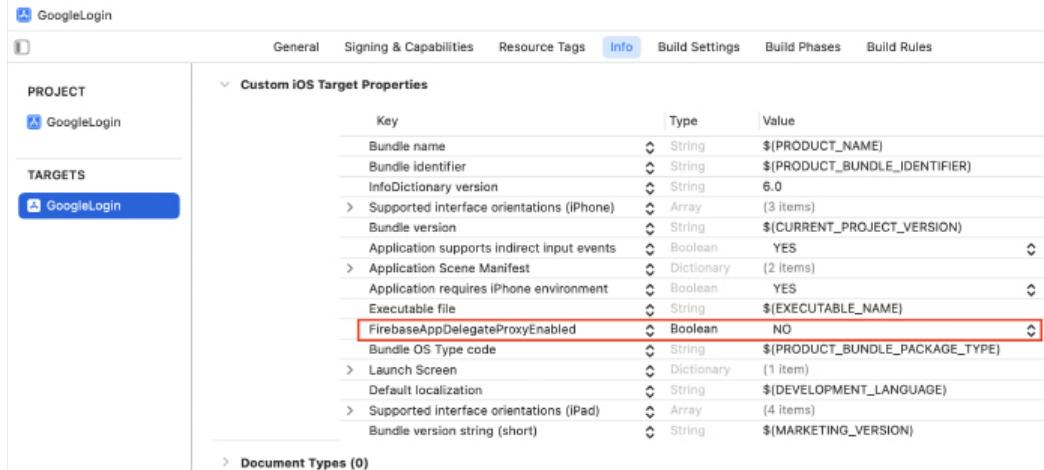


Figure 13.25: Disabling the Firebase method swizzling

After configuring the Xcode project, create a `FirebaseGoogleSignInApp` in Firebase, following the steps in *Integrating Firebase into a SwiftUI project*.

How to do it...

Firebase provides a library for using several sign-in methods, including Google Sign-In. Thanks to the Google Sign-In for iOS package, we will be able to allow our users to authenticate with their Google credentials in a secure and private way. This package provides two libraries, `GoogleSignIn`, which is based in UIKit, to authenticate the user with Google, and `GoogleSignInSwift`, which is in SwiftUI, and provides a `Sign in with Google` button.

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:

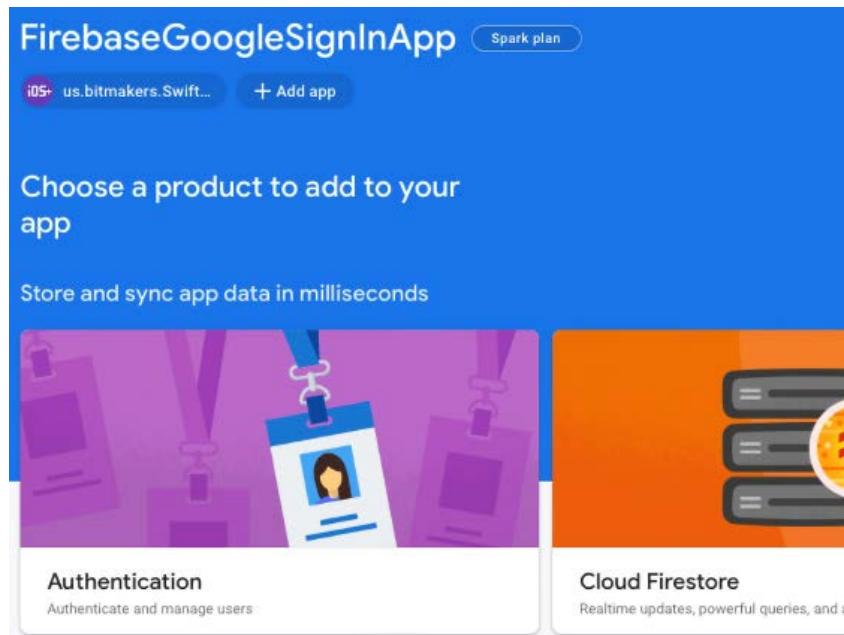


Figure 13.26: Firebase Authentication section

2. In the **Authentication** section, click on the **Get started** button and you'll be taken to the **Sign-in method** tab as shown in *Figure 13.27*:

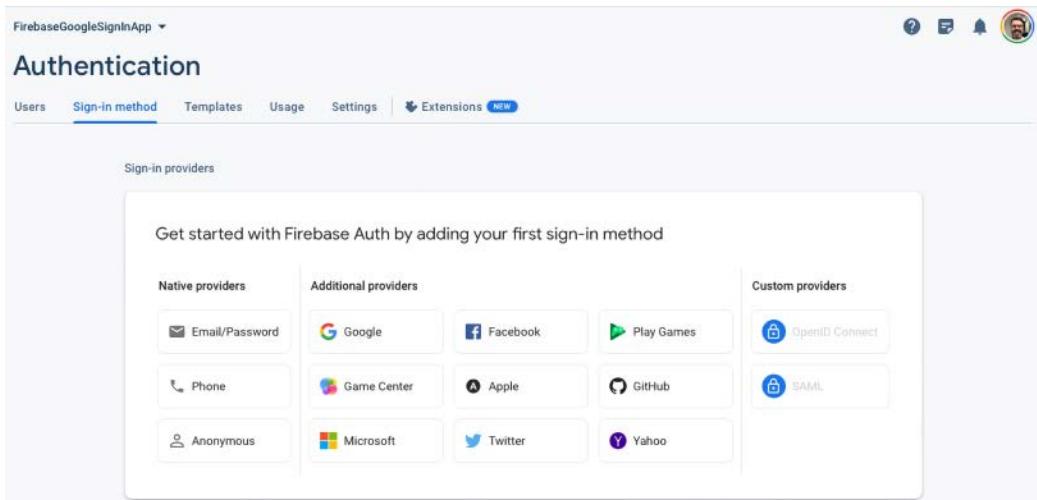


Figure 13.27: Firebase project authentication page, Sign-in method tab

3. From the list of sign-in methods, choose **Google**. On the next screen, click on the **Enable** toggle, and in the **Support email for project** dropdown, choose an email. When you are done, click on the **Save** button to save these changes:

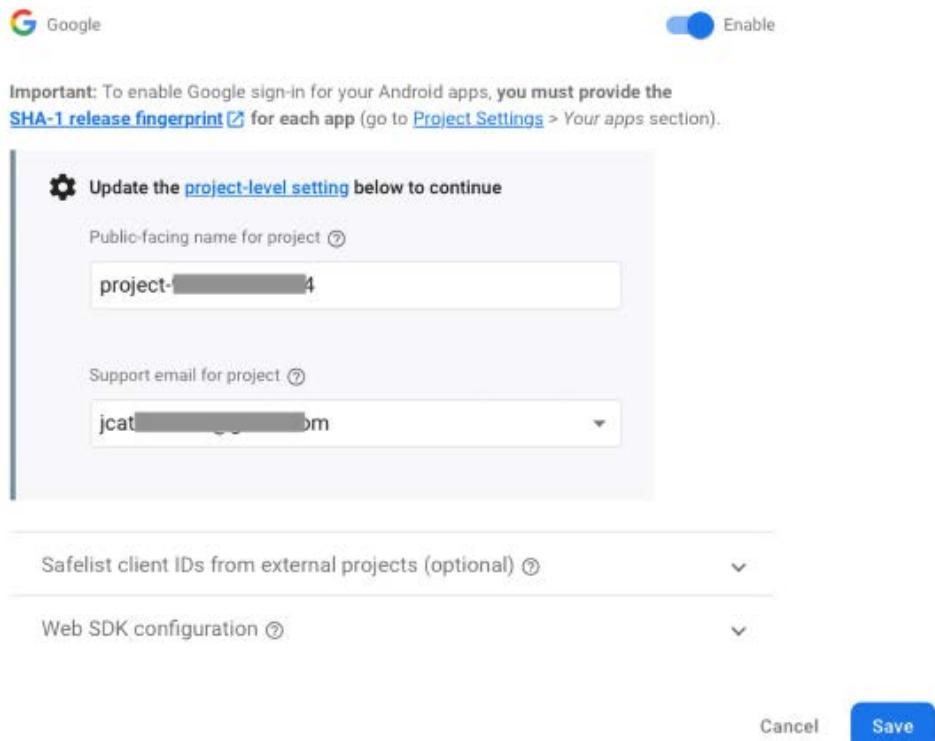


Figure 13.28: Google Sign-In configuration

4. After adding the Google provider, if you haven't done it before, go to the iOS configuration, add the bundle ID of your iOS app and download the `GoogleService-Info.plist` configuration file to your Mac, and add the file to your Xcode project. Each project has its own configuration file, so this file is different than the configuration file for the previous recipe.
5. 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.

6. Select the app from the TARGETS section, then select the Info tab, click the plus sign button to add an entry, and expand the URL Types section. Paste the copied value into the URL Schemes box as shown in the following figure:

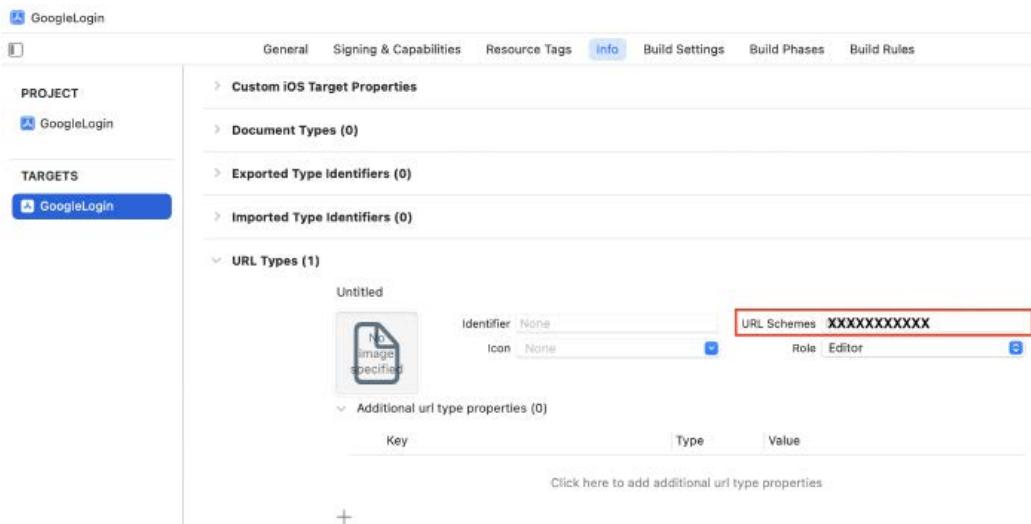


Figure 13.29: Configuring URL schemes

7. Let's work on the code now. Firstly, configure Firebase and Google Sign In in the GoogleLoginApp main struct:

```
import SwiftUI
import Firebase
import GoogleSignIn

class AppDelegate: NSObject, UIApplicationDelegate {
    func application(_ application: UIApplication,
didFinishLaunchingWithOptions launchOptions: [UIApplication.
LaunchOptionsKey : Any]? = nil) -> Bool {
        FirebaseApp.configure()
        return true
    }

    func application(_ app: UIApplication, open url: URL, options:
[UIApplication.OpenURLOptionsKey: Any] = [:]) -> Bool {
        GIDSignIn.sharedInstance.handle(url)
    }
}

@main
```

```
struct GoogleLoginApp: App {
    @UIApplicationDelegateAdaptor(AppDelegate.self) var delegate
    var body: some Scene {
        WindowGroup {
            ContentView()
        }
    }
}
```

8. In the `ContentView` struct, let's create a class named `AuthenticationViewModel` to handle our authentication with Google and Firebase. Since Google Sign-In for iOS uses UIKit, we will also add an extension to `UIApplication` to get the root view controller:

```
import SwiftUI
import Firebase
import GoogleSignIn
import GoogleSignInSwift

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
    }
}

@Observable final class AuthenticationViewModel {
    enum State {
        case busy
        case signedIn
        case signedOut
    }

    var state: State = .busy
    private var authResult: AuthDataResult? = nil
    var username: String { authResult?.user.displayName ?? "" }
    var email: String { authResult?.user.email ?? "" }
}
```

```
var photoURL: URL? { authResult?.user.photoURL }
```

```
func logout() {
    GIDSignIn.sharedInstance.signOut()
    GIDSignIn.sharedInstance.disconnect()
    try? Auth.auth().signOut()
    authResult = nil
    state = .signedOut
}
```

```
func restorePreviousSignIn() {
    GIDSignIn.sharedInstance.restorePreviousSignIn { user, error in
        if let error { print("Error: \(error.localizedDescription)") }
    }
    Task {
        await self.signIn(user: user)
    }
}
```

```
func login() {
    state = .busy
    guard let clientID = FirebaseApp.app()?.options.clientID,
          let rootViewController = UIApplication.
currentRootViewController else {
        return
    }
    let configuration = GIDConfiguration(clientID: clientID)
    GIDSignIn.sharedInstance.configuration = configuration
    GIDSignIn.sharedInstance.signIn(withPresenting:
rootViewController, hint: nil) { result, error in
        if let error { print("Error: \(error.localizedDescription)") }
    }
    Task {
        await self.signIn(user: result?.user)
    }
}
```

```
private func signIn(user: GIDGoogleUser?) async {
    guard let user, let idToken = user.idToken else {
```

```
        state = .signedOut
        return
    }
    let credential = GoogleAuthProvider.credential(withIDToken:
idToken.tokenString,
                                                accessToken: user.
accessToken.tokenString)
    do {
        authResult = try await Auth.auth().signIn(with: credential)
        state = .signedIn
    } catch {
        state = .signedOut
        print("Error: \(error.localizedDescription)")
    }
}
}
```

9. Let's create a `ProfileView` view to display the user profile, which we will show after the user is authenticated with Firebase using the Google credentials:

```
struct ProfileView: View {
    var authenticationViewModel: AuthenticationViewModel
    var body: some View {
        VStack(spacing: 10) {
            AsyncImage(url: authenticationViewModel.photoURL) { image in
                image
                    .resizable()
                    .clipShape(Circle())
            } placeholder: {
                ProgressView()
            }
            .frame(width: 100, height: 100)
            Text("User: \(authenticationViewModel.username)")
                .foregroundColor(.black)
            Text("Email: \(authenticationViewModel.email)")
                .foregroundColor(.black)
            Button("Logout") {
                authenticationViewModel.logout()
            }
            .buttonStyle(.borderedProminent)
            .frame(width: 200, height: 30, alignment: .center)
        }
    }
}
```

```
        .padding(.top, 10)
    }
}
```

10. Complete the app working on the main view of the app, `ContentView`. This view will present the user profile if the user is authenticated or a `Sign In with Google` button if the user is not authenticated. Additionally, it will show a progress view while the authentication is being handled:

```
struct ContentView: View {
    @State var authenticationViewModel = AuthenticationViewModel()
    var body: some View {
        ZStack {
            Color.white.edgesIgnoringSafeArea(.all)
            switch authenticationViewModel.state {
                case .busy:
                    ProgressView()
                case .signedIn:
                    ProfileView(authenticationViewModel:
                        authenticationViewModel)
                case .signedOut:
                    GoogleSignInButton(action: authenticationViewModel.login)
                        .frame(width: 150, height: 30, alignment: .center)
            }
        }
        .task {
            authenticationViewModel.restorePreviousSignIn()
        }
    }
}
```

Finally, take a few minutes to run the app on the iOS simulator or your iPhone, and get experience with how the authentication with your Google account works. The workflow should be similar to the one in the following screenshots:

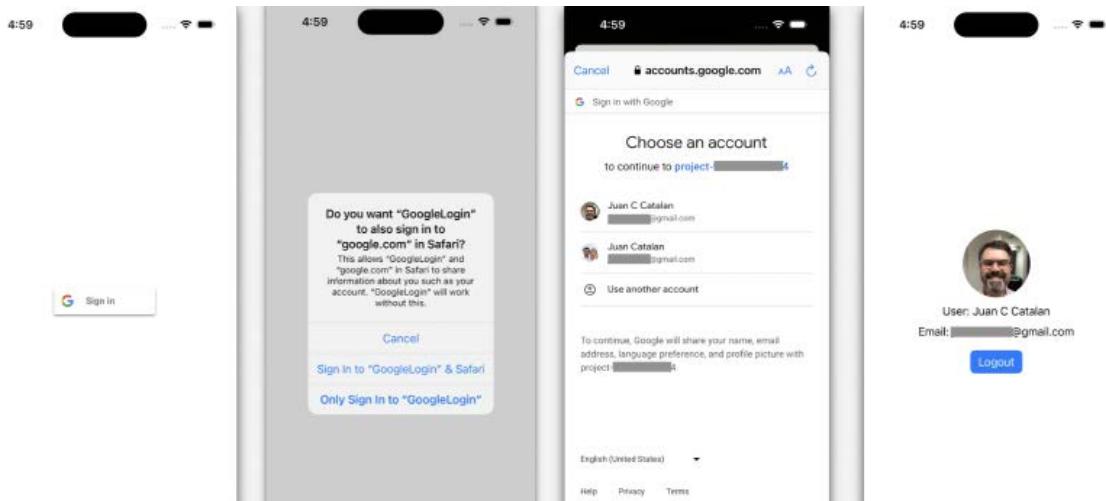


Figure 13.30: Signing in with a Google account

How it works...

That was another long recipe with quite a few configuration steps. But in the end, we have created an authentication component that can be useful in other apps too.

The authentication process is a sequence of two operations:

- Authenticate with Google and get the user credentials
- Authenticate with Firebase with the credentials obtained from Google

While the Firebase SDK fully embraces the `async await` pattern, that isn't the case for the Google Sign In framework yet. For example, in our `login()` function, we make two calls, one to the `signIn(withPresenting:hint:)` function to authenticate the user with their Google account, which uses a completion block, and another call to our custom `signIn(user:)` function, which is asynchronous, and in turn calls the Firebase `signIn(with:)` asynchronous function. Because our `login` function is not asynchronous, we must wrap the call to the asynchronous function in a `Task {}` struct.

Another thing to note is that the Google Sign In framework depends on UIKit. It requires a `UIViewController` to present the Google authentication views. In this recipe, we had to resort to UIKit to get the current root view controller, needed by the `signIn(withPresenting:hint:)` function:

```
extension UIApplication {  
    static var currentRootViewController: UIViewController? {  
        //...  
    }  
}
```

However, this solution isn't a future-proof way of providing a `UIViewController` to present the Google Login view, since Apple can change how the view hierarchy of an iOS app is created.

It is important to mention that to use FirebaseAuth with SwiftUI, we need to implement a `UIApplicationDelegate` class and initialize Firebase from this class, instead of from the `App` struct. Additionally, we also need to disable swizzling for FirebaseAuth. We implemented the initialization code recommended in the official Firebase docs and on the official Firebase YouTube channel. Links to these resources are provided in the next section.

Lastly, the `Sign-In with Google` button is a SwiftUI view provided by the `GoogleSignInSwift` module. This native SwiftUI component provided by Google minimized the use of UIKit in our app to just the extension to `UIApplication` mentioned earlier.

If you go to the Firebase console, choose **Authentication**, and then select the **Users** tab, you should see all the users who signed in to your app. The result should be like the following screenshot:

The screenshot shows the Firebase Authentication console with the 'Users' tab selected. There are two entries listed:

Identifier	Providers	Created	Signed In	User UID
xxxxxxxxxx@gmail.com	G	Aug 20, 2023	Aug 20, 2023	xxxxxxxxxxxxxxxxxxxx
xxxxxxxxxx@gmail.com	G	Aug 20, 2023	Aug 20, 2023	xxxxxxxxxxxxxxxxxxxx

Below the table, there are pagination controls: 'Rows per page: 50', '1 - 2 of 2', and navigation arrows.

Figure 13.31: Users who signed in to the app using their Google account

See also

- Getting started with Firebase on Apple platforms: https://youtu.be/F9Gs_pfT3hs
- SwiftUI and Objective-C inter-op with Swizzling: <https://github.com/firebase/firebase-ios-sdk/issues/10417>
- App Delegate swizzling: https://firebase.google.com/docs/ios/learn-more#app_delegate_swizzling
- Authenticate Using Google Sign-In on Apple Platforms: <https://firebase.google.com/docs/auth/ios/google-signin>

Implementing a 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 legacy database.
- Cloud Firestore, which is a newer 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 the repository 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 them in 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 as shown in the following figure:

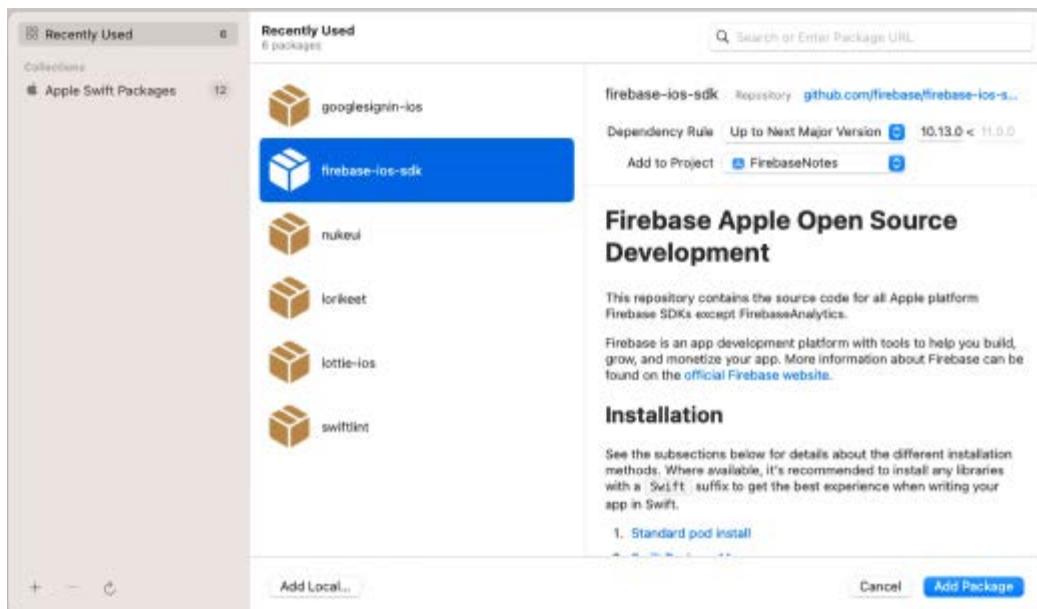


Figure 13.32: Adding the main Firebase package

2. Then select the **FirebaseFirestore** and **FirebaseFirestoreSwift** sub-packages as shown:

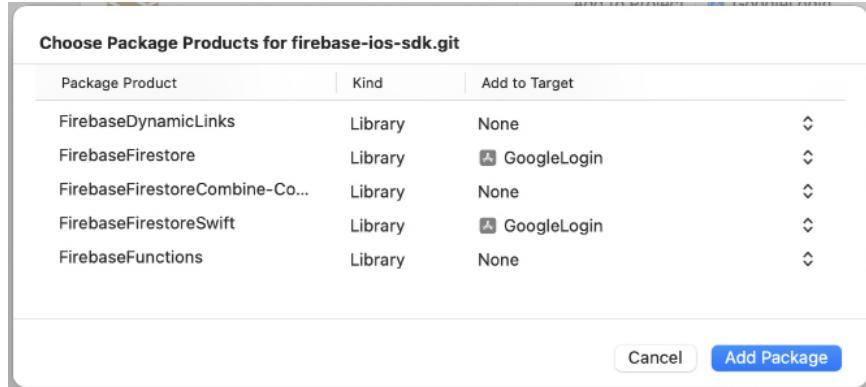


Figure 12.33: Adding the Firebase sub-packages

After configuring the project, create a **FirebaseNotesApp** in Firebase by following the steps in the *Integrating Firebase into a SwiftUI project recipe*. Don't forget to register the iOS app with the bundle ID. Download the **GoogleService-Info.plist** and add it to the Xcode project.

How to do it...

The app we are going to implement will have two main parts: the UI and the repository service. The repository service will be implemented using the Firestore SDK and the UI will be the SwiftUI code.

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** as shown:

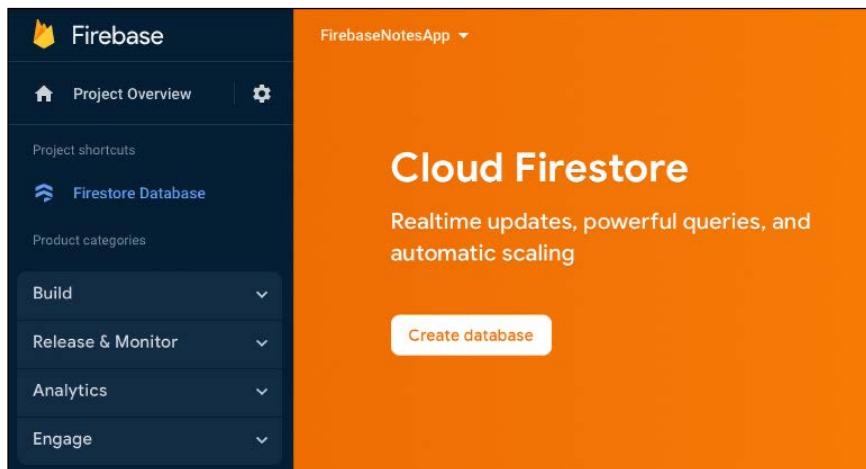


Figure 13.34: Creating a database

- Once you've clicked the **Create database** button, 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:

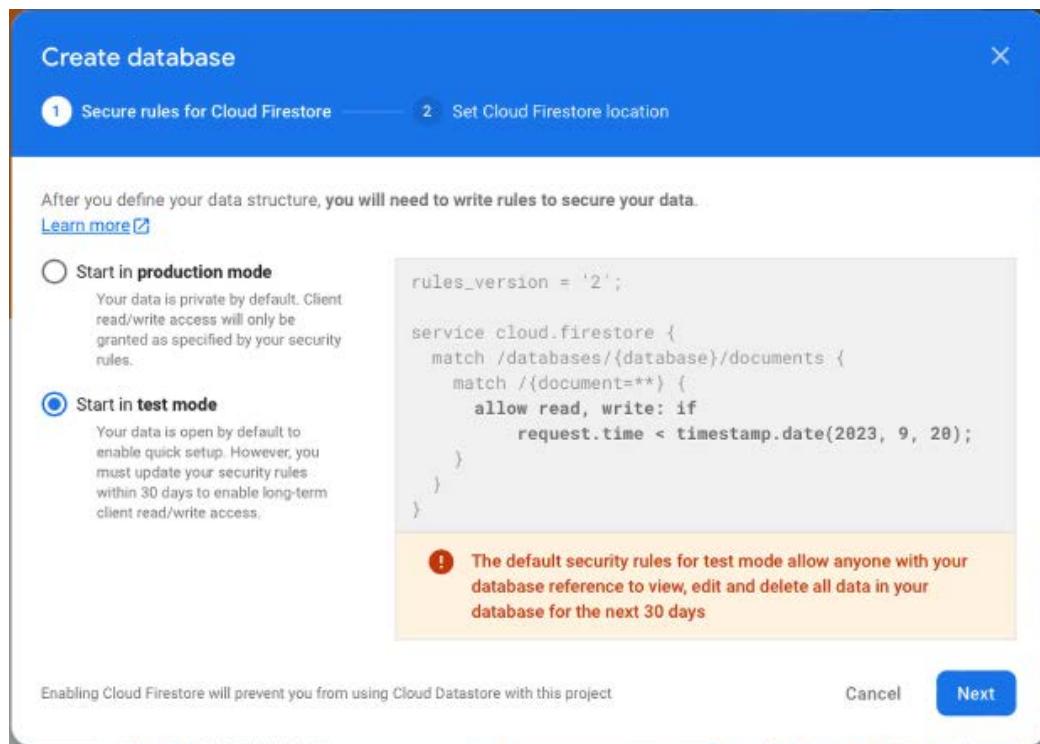


Figure 13.35: Database security rules

- Now, we must select the region where we're creating the database. Select the one closest to your area, as shown in the following figure:

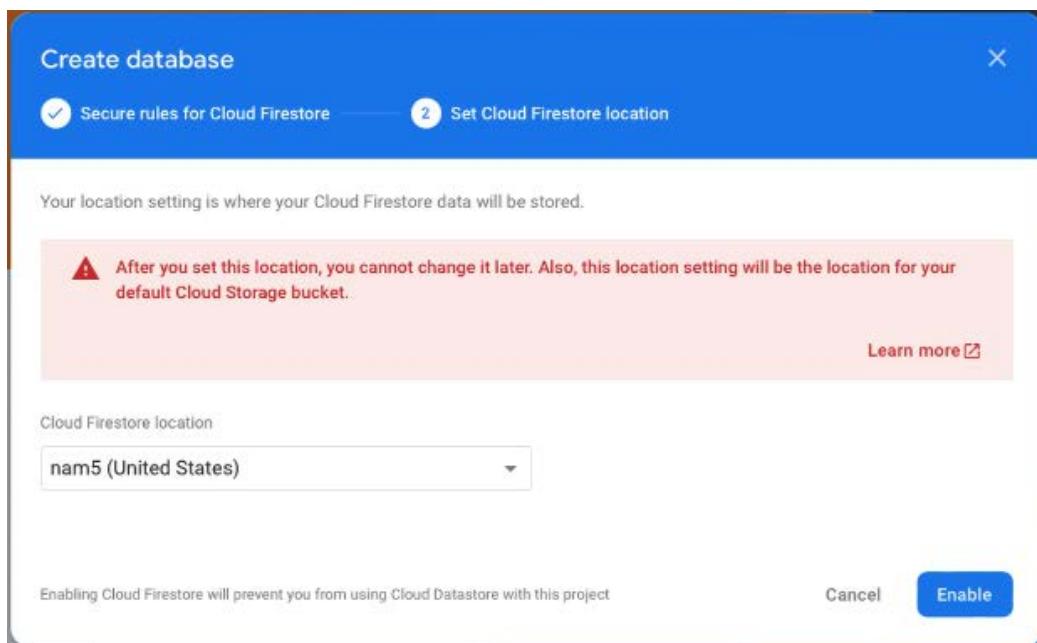


Figure 13.36: Cloud Firestore location

- By doing this, we have created the database. Verify this in the Firebase console:

The screenshot shows the 'Cloud Firestore' overview page for the project 'FirebaseNotesApp'. The top navigation bar includes tabs for 'Data' (which is active), 'Rules', 'Indexes', 'Usage', and 'Extensions (NEW)'. Below the tabs, there's a 'Protect your Cloud Firestore resources from abuse, such as billing fraud or phishing' button. Underneath, there are two buttons: 'Panel view' and 'Query builder'. The main content area features a large 'fir-notesapp-def94' collection entry with a document icon and a 'Start collection' button. There's also a small house icon.

Figure 13.37: Database overview on Firebase

5. Move to the code now. Configure Firebase in the `FirebaseNotesApp` main view. Since Firestore does not require a `UIApplicationDelegate` class to work, we will configure Firebase in the `App` struct:

```
import SwiftUI
import Firebase
@main
struct FirebaseNotesApp: App {
    init() {
        FirebaseApp.configure()
    }
    var body: some Scene {
        WindowGroup {
            ContentView()
        }
    }
}
```

6. Let's implement our model type. The `Note` we are going to implement is a simple `struct` with a `title`, a `body`, and its creation date visible. We will make it conform to `Identifiable` so as to use instances in a `List` view, conform to `Hashable` so as to use it in a `NavigationLink`, and conform to `Codable` so as to use it with Firestore. Add the following code in a new file in the project:

```
struct Note: Identifiable, Codable, Hashable {
    let id: String
    let title: String
    let date: Date
    let body: String
}
```

7. Let's create some sample data so we can use it in our previews. We will add an extension to our `Note` struct and define two instances, one with a short body and another one with a long body to test scrolling. We have omitted the text in the long body here. You have the complete code in the GitHub repository, and you could use your own text:

```
extension Note {
    static let sample = Note(id: UUID().uuidString,
                            title: "Sample Title",
                            date: Date(),
                            body: "Sample Body")

    // body text generated with: https://loremipsum.io
}
```

```
static let sampleWithLongBody = Note(id: UUID().uuidString,
                                      title: "Sample Title",
                                      date: Date(),
                                      body:
                                      """
                                      Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do
                                      eiusmod tempor incididunt ut
                                      ...
                                      quam vulputate dignissim suspendisse in est ante in nibh.
                                      """
)
}
```

8. In a separate file, create a `NotesService` class, which will handle our interactions with Firestore and store the notes in an array, which will then be used by our views. Import the Firebase libraries and annotate the class with the `@Observable` macro:

```
import Firebase
import FirebaseFirestoreSwift

@Observable class NotesService {
    var notes: [Note]

    init(notes: [Note] = []) {
        self.notes = notes
    }

    func fetch() {}

    func addNote(title: String, date: Date, body: String) {}

    private func startRealtimeUpdates() {}

    private func updateNotes(snapshot: QuerySnapshot) {}
}
```

9. Add a reference to a collection called "notes" in the Firestore instance, and another variable to store a listener for real-time updates:

```
@Observable class NotesService {  
    var notes: [Note]  
    private let dbCollection = Firestore.firestore().collection("notes")  
    private var listener: ListenerRegistration?  
    //...  
}
```

10. Implement the `fetch()` function:

```
func fetch() {  
    guard listener == nil else { return }  
    dbCollection.getDocuments { [self] querySnapshot, error in  
        guard let snapshot = querySnapshot else {  
            print("Error fetching snapshots: \(error!)")  
            return  
        }  
        updateNotes(snapshot: snapshot)  
    }  
}
```

11. The `addNote()` function appends a new note to the collection in Firestore and then refreshes the local array of notes:

```
func addNote(title: String, date: Date, body: String) {  
    let note = Note(id: UUID().uuidString, title: title, date: date, body:  
body)  
    _ = try? dbCollection.addDocument(from: note)  
    fetch()  
}
```

12. Implement the `updateNotes(snapshot:)` function, which will take the collection returned by Firestore and convert it to notes instances, which we will assign to the `notes` array. Thanks to the `Observable` macro, any changes to the `notes` array will be broadcast to the views:

```
private func updateNotes(snapshot: QuerySnapshot) {  
    let notes: [Note] = snapshot.documents.compactMap { document in  
        try? document.data(as: Note.self)  
    }  
    self.notes = notes.sorted {  
        $0.date < $1.date  
    }  
}
```

13. Implement the `startRealtimeUpdates()` function to subscribe to Firestore real-time updates:

```
private func startRealtimeUpdates() {
    listener = dbCollection.addSnapshotListener { [self] querySnapshot,
    error in
        guard let snapshot = querySnapshot else {
            print("Error fetching snapshots: \(error!)")
            return
        }
        snapshot.documentChanges.forEach { diff in
            if (diff.type == .added) {
                print("New note: \(diff.document.data())")
            }
            if (diff.type == .modified) {
                print("Modified note: \(diff.document.data())")
            }
            if (diff.type == .removed) {
                print("Removed note: \(diff.document.data())")
            }
        }
        updateNotes(snapshot: snapshot)
    }
}
```

14. Modify the `init()` method to call the `startRealtimeUpdates()` function:

```
init(notes: [Note] = []) {
    self.notes = notes
    startRealtimeUpdates()
}
```

15. Now that we have finished with the service, let's concentrate our efforts on visualizing the notes. We are going to implement four views: `ContentView` to show the notes in a list, `NoteSummaryView` to represent an entry in the list, `AddNoteView` to create a new note, and `NoteDetailView` to visualize a note.

16. Let's start with the custom views to visualize notes. Create a new SwiftUI file named `NoteSummaryView.swift` and add the corresponding view:

```
struct NoteSummaryView: View {
    private let format: Date.FormatStyle =
        .dateTime.month(.wide).day().year()
    var note: Note
    var body: some View {
        VStack(alignment: .leading) {
```

```
        Text(note.title)
            .font(.headline)
            .fontWeight(.bold)
        Text("\(note.date, format: format)")
            .font(.subheadline)
    }
}
}

#Preview {
List(0 ..< 5) { item in
    NoteSummaryView(note: Note.sample)
}
}
```

If everything goes well, the preview should look like this:

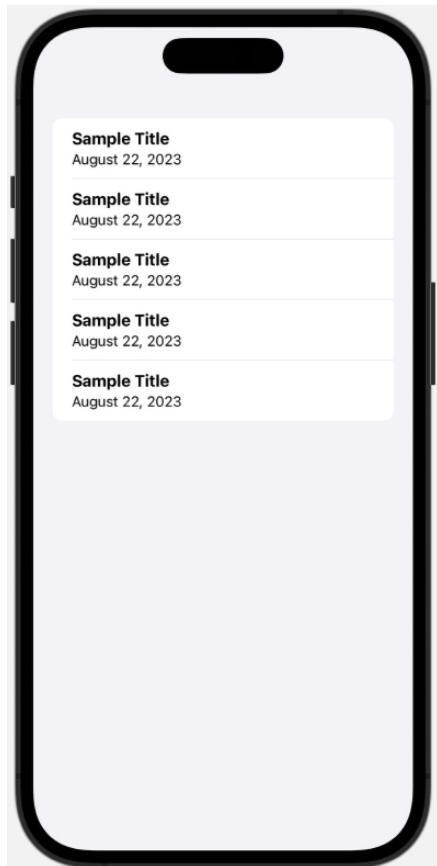


Figure 13.38: Preview of our summary view in a list

17. Create a new SwiftUI file named `NoteDetailView.swift` and add the following view:

```
struct NoteDetailView: View {
    let note: Note
    var body: some View {
        VStack(spacing: 12) {
            Text(note.title)
                .font(.headline)
                .fontWeight(.bold)
            TextEditor(text: .constant(note.body))
                .border(.gray)
        }
        .padding(24)
    }
}

#Preview {
    NoteDetailView(note: Note.sampleWithLongBody)
}
```

The preview should look like this:



Figure 13.39: Preview of our detail view

18. Finally, create another SwiftUI file named `AddNoteView.swift`. This view will have some local state to temporarily hold the properties of the note, a button to save the note to Firestore, and a binding variable of type Boolean to drive the visibility of the view when presented modally from our `ContentView`:

```
struct AddNoteView: View {
    @State private var title: String = ""
    @State private var bodyText: String = ""
    @Environment(\.dismiss) var dismiss

    var service: NotesService?

    var body: some View {
        NavigationStack {
            VStack(spacing: 12) {
                TextField("Title", text: $title)
                    .padding(4)
                    .border(.gray)
                TextEditor(text: $bodyText)
                    .border(.gray)
            }
            .padding(32)
            .navigationTitle("New Note")
            .navigationBarTitleDisplayMode(.inline)
            .toolbar {
                ToolbarItem(placement: .navigationBarTrailing) {
                    Button {
                        service?.addNote(
                            title: title,
                            date: Date(),
                            body: bodyText
                        )
                        dismiss()
                    } label: {
                        Image(systemName: "checkmark")
                            .font(.headline)
                    }
                    .disabled(title.isEmpty)
                }
            }
        }
    }
}
```

```
}
```

```
#Preview {
    AddNoteView(service: nil)
}
```

And the preview should look like this:

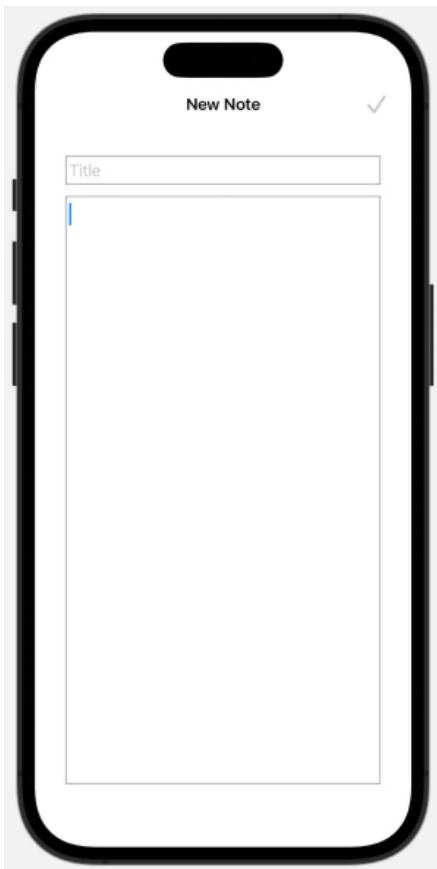


Figure 13.40: Preview of the add note view

19. Let's switch to `ContentView` and finish our work with the views. We will start adding the `@State` properties:

```
struct ContentView: View {
    @State private var service: NotesService = NotesService()
    @State private var isNewNotePresented = false

    var body: some View {
        //...
    }
}
```

20. In the `body`, add a `NavigationStack` with a `List` to display the notes in a summarized way. If the user clicks on the row, they will be redirected to a new view that shows the selected note in full detail:

```
var body: some View {
    NavigationStack {
        List(service.notes) { note in
            NavigationLink(value: note) {
                NoteSummaryView(note: note)
            }
        }
        .navigationDestination(for: Note.self) { note in
            NoteDetailView(note: note)
        }
        .navigationTitle("FireNotes")
        .navigationBarTitleDisplayMode(.inline)
        // ...
    }
}
```

21. To create a new note, add a "plus" button to the navigation bar that toggles the `isNewNotePresented` `@State` variable:

```
var body: some View {
    NavigationStack {
        //...
        .navigationBarTitleDisplayMode(.inline)
        .toolbar {
            Button {
                isNewNotePresented.toggle()
            } label: {
```

```
        Image(systemName: "plus")
            .font(.headline)
        }
    }
    // ...
}
}
```

22. The `AddNoteView` will be presented modally using the `.sheet()` modifier:

```
var body: some View {
    NavigationStack {
        //...
        .toolbar {
            //...
        }
        .sheet(isPresented: $isNewNotePresented) {
            AddNoteView(service: service)
        }
    }
}
```

23. Finally, add a `.task()` modifier to fetch the notes:

```
var body: some View {
    NavigationStack {
        //...
        .sheet(isPresented: $isNewNotePresented) {
            AddNoteView(service: service)
        }
        .task {
            service.fetch()
        }
    }
}
```

Done! The app is now complete. We can try it out by adding notes to it, as shown in the following screenshot:



Figure 13.41: Notes app in action

How it works...

The interface between the Firestore library and SwiftUI is native, and it works seamlessly with SwiftUI. Firestore is a so-called document store, where the data is saved in the form of a collection of documents. Think of a document as a dictionary with key-value pairs.

If you have some 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 new attributes, we will 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.

The Firestore service 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 have created in this recipe looks like:

The screenshot shows the Cloud Firestore console interface. At the top, there's a navigation bar with tabs for Data, Rules, Indexes, Usage, and Extensions. Below the navigation bar, there are two buttons: Panel view and Query builder. The main area displays a hierarchical tree structure under the 'notes' collection. The first node is 'tfCxxF6zsZ61Ly...', which is expanded to show five child documents: 'XxU83UHNieKY1FNv4edv', 'k31I1i3X1ELswY2hThSa', 'nSk6JAY3eg11ka19FYR0', 'srk0Piq98HoxpuUGCSiyN', and 'tfCxxF6zsZ61LyYOH2H8'. The document 'tfCxxF6zsZ61LyYOH2H8' is selected and its details are shown in the right-hand panel. The right-hand panel includes a 'More in Google Cloud' dropdown and a 'Start collection' button. The document details are as follows:

body: "Body 5"
date: August 22, 2023 at 3:02:36 PM UTC-5
id: "3EFD5B2F-B35D-414A-A90A-7DC7693B77CF"
title: "Note 5"

Figure 13.42: Firestore panel view of our data

Try for yourself and update any notes using the Firestore console. You'll see that, thanks to the real-time updates, our app will update the notes immediately without having to retrigger or refresh manually.

There's more...

As we have seen, using Firestore is easy and fits very well with SwiftUI. From here, we can add more features.

An obvious one would be to add the possibility of editing a note and another one to delete a note.

I encourage you to explore all the capabilities of Firestore and add the two features mentioned before. You could test the app running in two simulators to see the changes in real time.

I'm pretty sure you'll be amazed by the simplicity of coding these additional features, thus making our basic notes app close to the official Notes app provided by Apple with every iPhone.

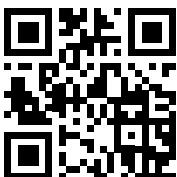
See also

- Get data with Cloud Firestore: <https://firebase.google.com/docs/firestore/query-data/get-data>
- Get real-time updates with Cloud Firestore: <https://firebase.google.com/docs/firestore/query-data/listen>
- Add data to Cloud Firestore: <https://firebase.google.com/docs/firestore/manage-data/add-data>
- Delete data from Cloud Firestore: <https://firebase.google.com/docs/firestore/manage-data/delete-data>

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

<https://packt.link/swiftUI>



14

Persistence in SwiftUI with Core Data and SwiftData

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 view. Also, `NSManagedObjectContext` is injected into the environment of the view hierarchy so that SwiftUI views 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 select `Core Data` in the `Storage` drop-down menu. By doing so, Xcode will 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 with SwiftUI.

Apple introduced SwiftData during the WWDC23 conference. Apple describes SwiftData as a new persistence framework combining Core Data's proven persistence technology and Swift's modern concurrency features. When creating a new project in Xcode 15, and selecting the iOS app template, we can select `SwiftData` in the `Storage` drop-down menu to generate the necessary code to use SwiftData in our project. However, following the approach of the *Integrating Core Data with SwiftUI* recipe in this chapter, we will implement SwiftData manually to better understand how it works with SwiftUI.

In this chapter, we are going to learn how to implement persistence with Core Data and SwiftData in SwiftUI.

We'll explore the basics of Core Data and SwiftData 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`
- Getting started with SwiftData

Technical requirements

The code in this chapter is based on Xcode 15.0 and iOS 17.0. You can download and install the latest version of Xcode from the App Store. You'll also need to be running macOS Ventura (13.5) or newer.

Simply search for Xcode in the App Store and select and download the latest version. Launch Xcode and follow any additional installation instructions that your system may prompt you with. Once Xcode has fully launched, you're ready to go.

All the code examples for this chapter can be found on GitHub at <https://github.com/PacktPublishing/SwiftUI-Cookbook-3rd-Edition/tree/main/Chapter14-Handling-Core-Data-in-SwiftUI>.

Integrating Core Data with SwiftUI

If you're familiar with iOS development, 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, before iOS 14, provided Xcode templates that created the Core Data containers in the `AppDelegate`. Other developers prefer to wrap Core Data inside custom classes, abstracting Core Data entirely, while encapsulating the whole Core Data Stack and its managed objects in a module. By abstracting the persistence code, it will be easy to move to another persistence solution, such as `Realm` (<https://realm.io>), if needed.

SwiftUI's integration, however, points firmly in one direction: create the container when the app starts, inject it into the 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 interactions with Core Data without being overwhelmed by the complexity of its initial setup code.

Getting ready

Let's create a SwiftUI app called `SwiftUICoreDataStack`, ensuring that we choose `None` for the Storage option, as seen in Figure 14.1:

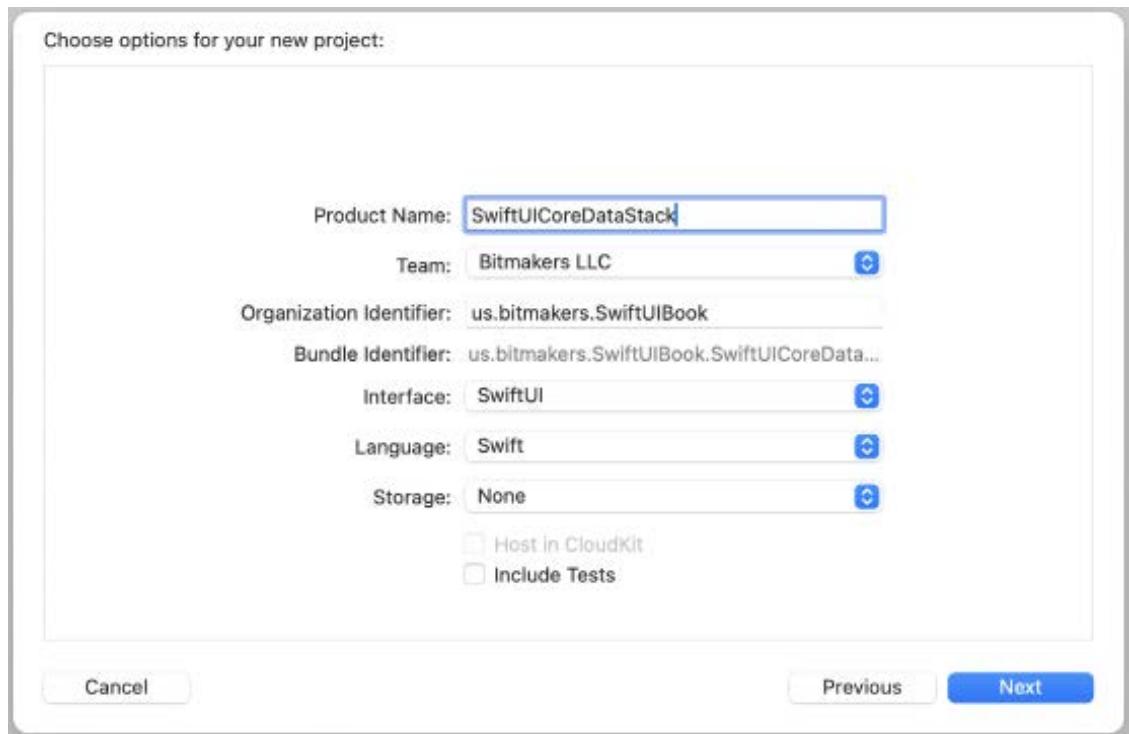


Figure 14.1: A SwiftUI app without Core Data storage

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 App struct, which will handle its life cycle.

Let's get started:

1. Core Data needs a model file with entities to describe our data. We will cover these in detail in the *Showing Core Data objects with @FetchRequest* recipe of this chapter. In Xcode, add a Core Data model called `ContactsModel`, as shown in *Figure 14.2*.

We should add some entities to the model file, but for the moment, let's leave it empty.

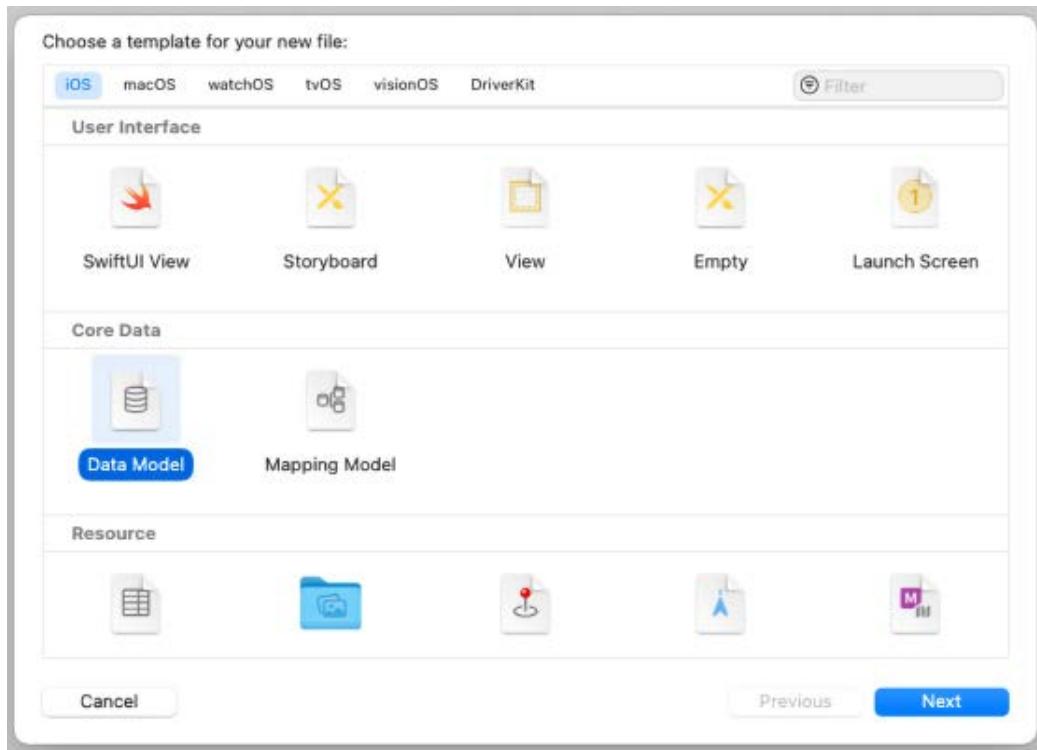


Figure 14.2: Adding a Core Data model

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 a reference to an instance of `NSPersistentContainer`, which encapsulates the Core Data stack in our app:

```
import CoreData
class CoreDataStack: ObservableObject {
    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 to the Core Data persistent storage:

```
class CoreDataStack {
//...
func save () {
    guard managedObjectContext.hasChanges else { return }
    do {
        try managedObjectContext.save()
    } catch {
        print(error)
    }
}
}
```

4. Create a unique instance of our `CoreDataStack` class in the `App` struct. Inject instances of our class and the `managedObjectContext` into the `Environment`:

```
@main
struct SwiftUICoreDataStackApp: App {
    private let coreDataStack = CoreDataStack(modelName: "ContactsModel")

    var body: some Scene {
        WindowGroup {
            ContentView()
                .environmentObject(coreDataStack)
                .environment(\.managedObjectContext, coreDataStack.
managedObjectContext)
        }
    }
}
```

- 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 as shown in *Figure 14.3*:



Figure 14.3: The shared instance of NSManagedObjectContext in SwiftUI

How it works...

In this recipe, we learned how to create a container for the Core Data stack. 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 the `NSPersistentContainer`'s view context 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 request. We will look at this in more detail in the *Showing Core Data objects with `@FetchRequest`* recipe.

There's more...

To better understand the code shown in this recipe, I encourage you to create another SwiftUI project. In this new project, add `Core Data` as the `Storage` option while creating the project, so that Xcode will use Apple's template. By doing this, you can compare the 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 is 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 database containing the clothes for the new season is created and released.

Given the importance of having this feature, 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 the `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 start-up, 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 App struct, create an instance of our CoreDataStack class with a model named `ContactsModel`. Then inject the `managedObjectContext` instance into the Environment:

```
@main
struct FetchContactsApp: App {
    private let coreDataStack = CoreDataStack(modelName: "ContactsModel")

    var body: some Scene {
        WindowGroup {
            ContentView()
                .environment(\.managedObjectContext, coreDataStack.
managedObjectContext)
        }
    }
}
```

3. Add a Core Data model called **ContactsModel**. Open the model file and do the following:
1. Add a new entity called **Contact**.
 2. Add three string attributes called **firstName**, **lastName**, and **phoneNumber**.
 3. Check that **Codegen** is set to **Class Definition**.
 4. The following screenshot shows the steps you must follow to update the model:

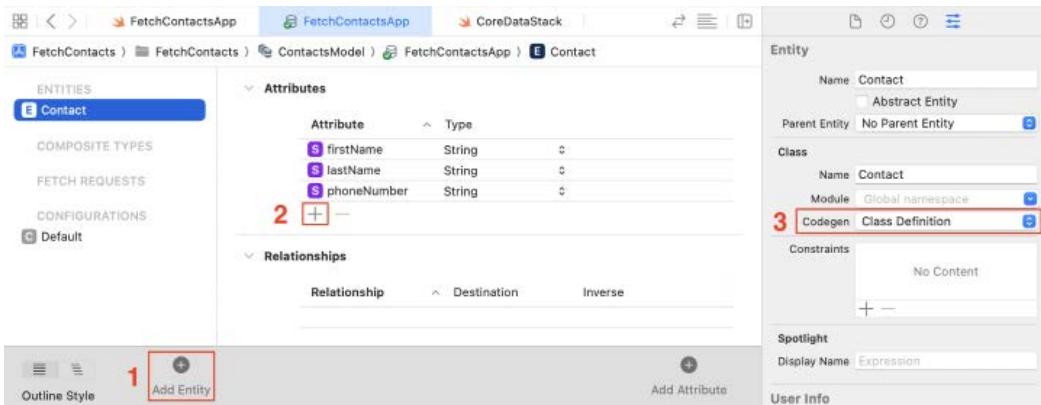


Figure 14.4: Creating the Core Data model

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.
5. Build the app by pressing *Command* (⌘) + *B*, so that Xcode generates the class for our **Contact** entity.
6. Switch to the **CoreDataStack** class and add a class extension with an **insertContact()** function. This function will be used to insert a new contact in the managed object context:

```
extension CoreDataStack {
    func insertContact(firstName: String, lastName: String, phoneNumber: String) {
        let contact = Contact(context: managedObjectContext)
        contact.firstName = firstName
        contact.lastName = lastName
        contact.phoneNumber = phoneNumber
    }
}
```

7. Now, let's create a function that will insert a bunch of contacts. Create another extension to the **CoreDataStack** class with the **addContacts()** function:

```
extension CoreDataStack {
    func addContacts() {
        [("Daenerys", "Targaryen", "02079460803"),
         ("Bran", "Stark", "02079460071"),
```

```

        ("Jon", "Snow", "02079460874"),
        ("Theon", "Greyjoy", "02890180771"),
        ("Ned", "Stark", "011774960111"),
        ("Tyrion", "Lannister", "02079460695"),
        ("Arya", "Stark", "02079460878"),
        ("Stannis", "Baratheon", "02079460367"),
        ("Samwell", "Tarly", "011774960104"),
        ("Jaime", "Lannister", "02890180239"),
        ("Jorah", "Mormont", "02079460025"),
        ("Jeor", "Mormont", "02079460127"),
        ("Robb", "Stark", "011774960384"),
        ("Joffrey", "Baratheon", "02079460963"),
        ("Tywin", "Lannister", "02890180899"),
        ("Margaery", "Tyrell", "011774960635"),
        ("Catelyn", "Stark", "02890180301"),
        ("Viserys", "Targaryen", "02079460220"),
        ("Cersei", "Lannister", "02890180492"),
        ("Davos", "Seaworth", "02079460848"),
        ("Sansa", "Stark", "02890180764")]
    .forEach { (firstName, lastName, phoneNumber) in
        insertContact(firstName: firstName, lastName: lastName,
        phoneNumber: phoneNumber)
    }
    save()
}
}

```

8. We will call the function `addContacts()` only the first time we run 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. Switch to the `App` struct and add a `@AppStorage` property wrapper to declare a `appHasData` Boolean variable with a default value of `false`:

```

@main
struct FetchContactsApp: App {
    private let coreDataStack = CoreDataStack(modelName: "ContactsModel")
    @AppStorage("appHasData") var appHasData = false
    // ...
}

```

9. Then in the body of the `App` struct, add an `onAppear(perform:)` modifier to add the contacts if `appHasData` is `false`:

```

var body: some Scene {

```

```
WindowGroup {
    ContentView()
        .environment(\.managedObjectContext, coreDataStack.
managedObjectContext)
        .onAppear {
            if !appHasData {
                coreDataStack.addContacts()
                appHasData = true
            }
        }
    }
}
```

10. Now, let's move on to the code for visualizing the contacts. In `ContentView.swift`, create a `ContactView` view:

```
struct ContactView: View {
    let contact: Contact
    var body: some View {
        HStack {
            Text(contact.firstName ?? "-")
            Text(contact.lastName ?? "-")
            Spacer()
            Text(contact.phoneNumber ?? "-")
        }
    }
}
```

11. 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>
}
```

12. Finally, the body variable will simply render the list of contacts. Replace the contents of the body variable with 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 as shown in the following screenshot:

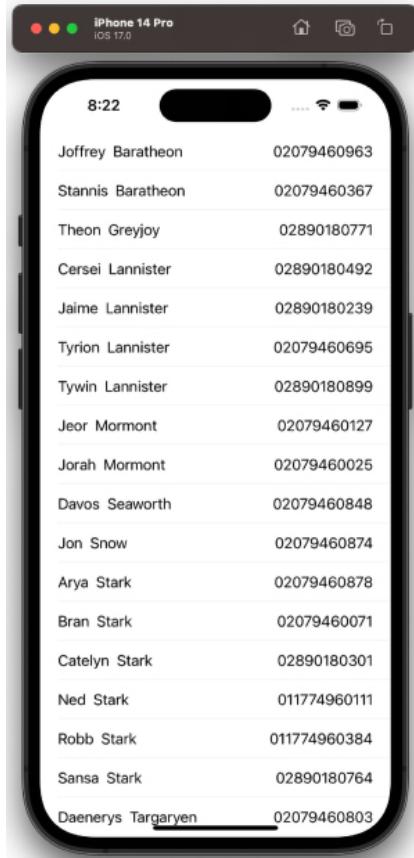


Figure 14.5: List of contacts from Core Data

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 a `managedObjectContext` and expects it in `@Environment(\.managedObjectContext)`. As you may recall, in the `App` struct, we are injecting `managedObjectContext` into `@Environment (\.managedObjectContext)` so that our code works without needing any further configuration.

Adding Core Data objects from a SwiftUI view

A storage 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 from a SwiftUI view.

In this recipe, we are going to implement a simple Contacts app where we can add storable contact profiles to Core Data for persistent storage.

Getting ready

Create a SwiftUI app called `AddContacts`.

Before you start this recipe, make sure you have completed steps 1 to 6 of the *Showing Core Data objects with @FetchRequest* recipe. Then, you can work on 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 contact, we are going to implement a simple modal view, presented as a sheet, with three text fields: two for the 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 contact list and presents each profile. The list is embedded in a `NavigationStack`. Add the following code to achieve this:

```

struct ContentView: View {
    //....
    var body: some View {
        NavigationStack {
            List(contacts, id: \.self) {
                ContactView(contact: $0)
            }
            .listStyle(.plain)
            .navigationTitle("Contacts")
            .navigationBarTitleDisplayMode(.inline)
        }
    }
}

```

4. To present the modal view 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 {
        NavigationStack {

```

```
//...
.toolbar {
    Button {
        isAddContactPresented.toggle()
    } label: {
        Image(systemName: "plus")
            .font(.headline)
    }
}
```

5. Switch to CoreDataStack and add conformance to the ObservableObject protocol. After this change, the class declaration should look like:

```
class CoreDataStack: ObservableObject { ... }
```

6. Switch to the App struct and inject the instance of CoreDataStack into the Environment. After this change, the body variable should look as follows:

```
var body: some Scene {
    WindowGroup {
        ContentView()
            .environment(\.managedObjectContext, coreDataStack.
managedObjectContext)
            .environmentObject(coreDataStack)
    }
}
```

7. If the isAddContactPresented property is true, we will present our custom view so that we can add the necessary contact details. Add a .sheet modifier to present the view modally:

```
struct ContentView: View {
//...
@EnvironmentObject var coreDataStack: CoreDataStack
var body: some View {
    NavigationStack {
//...
.toolbar {
// ...
}
.sheet(isPresented: $isAddContactPresented) {
    AddNewContact()
        .environmentObject(coreDataStack)
}
```

```
        }
    }
}
```

8. The `AddNewContact` view has three text fields for the contact details. These are for the first name, last name, and phone number:

```
struct AddNewContact: View {
    @EnvironmentObject var coreDataStack: CoreDataStack
    @Environment(\.dismiss) private var dismiss
    @State var firstName = ""
    @State var lastName = ""
    @State var phoneNumber = ""
}
```

9. The `body` function renders the text fields in a vertical stack, embedded in `NavigationStack`. Add the following code:

```
struct AddNewContact: View {
    // ...
    var body: some View {
        NavigationStack {
            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")
        }
    }
}
```

10. Inside `NavigationStack`, add a button to the navigation bar when we have finished filling in the contact details:

```
var body: some View {
    NavigationStack {
        // ...
        .toolbar {
            Button(action: saveContact) {
                Image(systemName: "checkmark")
                    .font(.headline)
            }
            .disabled(isDisabled)
        }
    }
}
```

11. This button is only enabled if all the fields have been filled in. After the `body` variable add this private variable:

```
private var isDisabled: Bool {
    firstName.isEmpty || lastName.isEmpty || phoneNumber.isEmpty
}
```

12. When this button is tapped, we will insert a new contact into the persistent storage and save it, thanks to our `CoreDataStack` class. Achieve this by adding the following code:

```
private func saveContact() {
    coreDataStack.insertContact(firstName: firstName, lastName: lastName,
    phoneNumber: phoneNumber)
    coreDataStack.save()
    dismiss()
}
```

Now, we can run the app, add contacts, and see them visualized in the list:

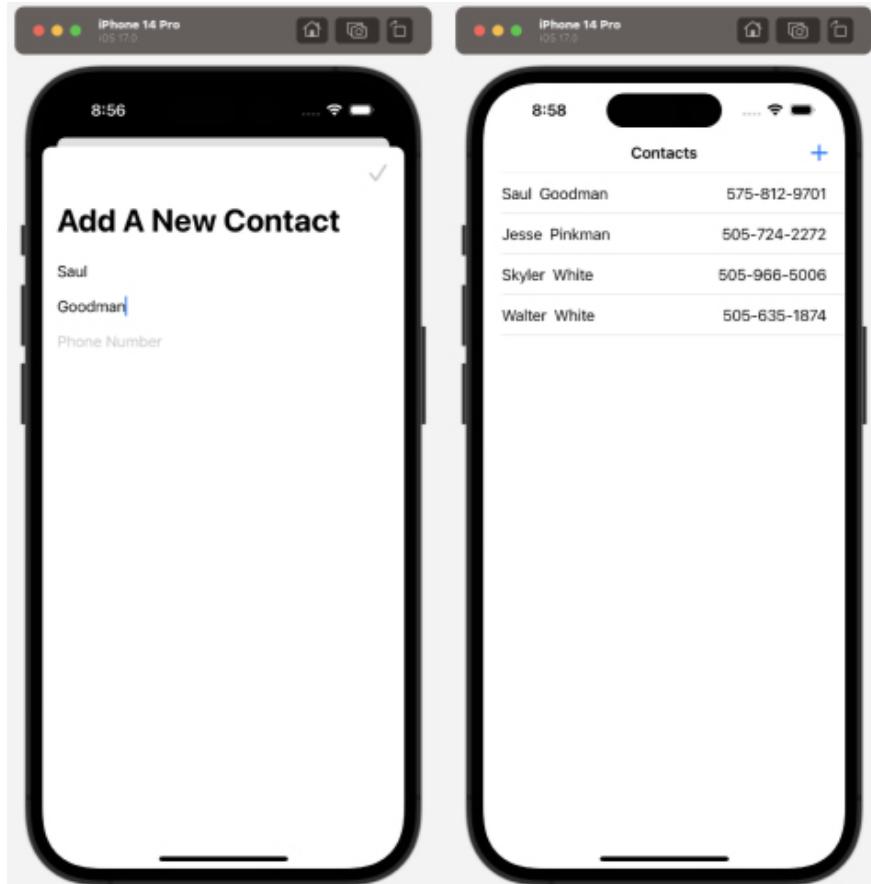


Figure 14.6: Adding a contact to the list

How it works...

This recipe is built on the foundation of two other recipes: the *Integrating Core Data with SwiftUI recipe*, which introduced the `CoreDataStack` class and injected the managed object context into the Environment, and *Showing Core Data objects with @FetchRequest recipe*, where we learned how to fetch objects from the 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 whether 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 data that matches a filter is retrieved from the repository and transformed into actual Swift 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 that it must be created at the beginning. It cannot change during the life cycle of the view because of a search text field.

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 you start this recipe, ensure you have completed *steps 1 to 10* 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 insights into it.

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 a parameter, and it builds 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[c] %@", 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 `FilteredContacts` view we created in *steps 1 and 2* in a `NavigationStack` in the body of the `ContentView` struct:

```
struct ContentView: View {  
    @State private var searchText : String = ""  
  
    var body: some View {  
        NavigationStack {  
            FilteredContacts(filter: searchText)  
                .navigationTitle("Contacts")  
                .navigationBarTitleDisplayMode(.inline)  
        }  
    }  
}
```

```
        }
        .searchable(text: $searchText)
    }
}
```

Upon running the app, we will see that changing the text in `SearchBar` changes the list of visualized contacts accordingly:

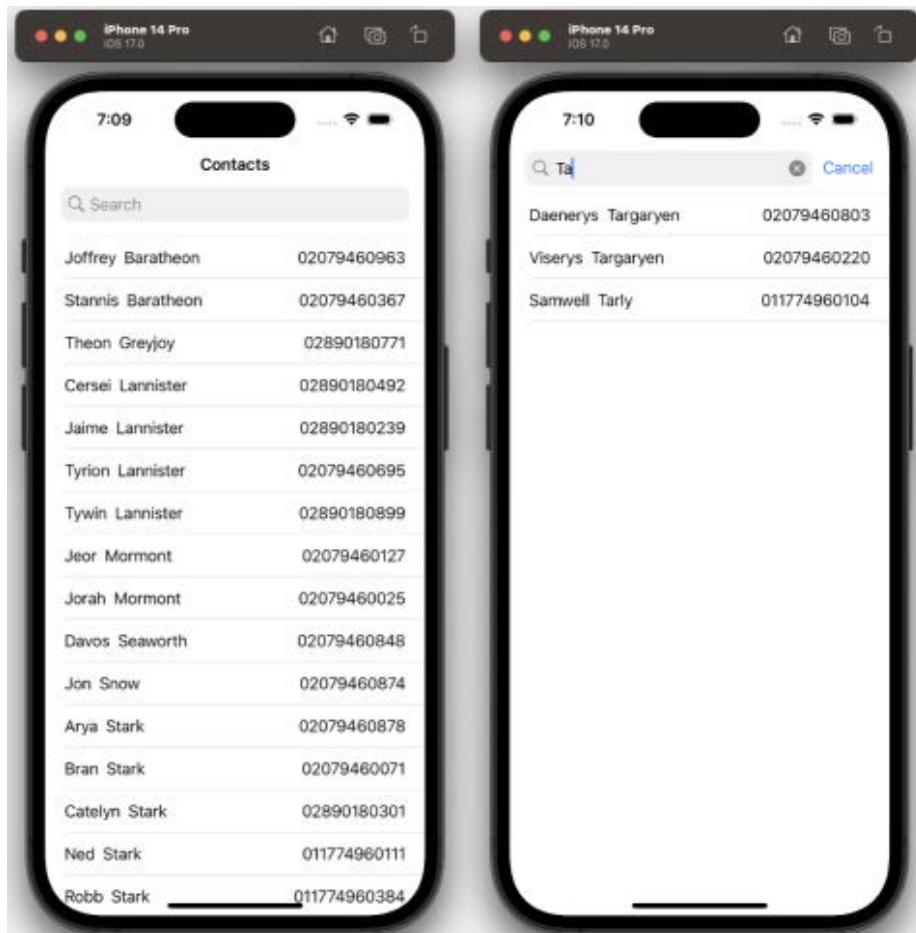


Figure 14.7: Core Data dynamic filtering

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[c] %@", 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 instantiated the `FetchRequest` with a predicate constructed with the `filter` String passed as a parameter to `init()`.

The `FetchRequest` property wrapper, via `managedObjectContext`, retrieves the objects and stores them as a `FetchedResults` type in the wrapped property. We can reach the results by accessing the wrapped value of the property. The results can be iterated 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 solved, 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 which when added to `NavigationStack`, adds a `SearchBar` component to the top of the view contained inside the `NavigationStack`. 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, make sure you have completed *steps 1 to 10* 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. Switch to the App struct and inject the instance of CoreDataStack into the Environment. After this change, the body variable should be as follows:

```
var body: some Scene {  
    WindowGroup {  
        ContentView()  
            .environment(\.managedObjectContext, coreDataStack.  
managedObjectContext)  
            .environmentObject(coreDataStack)  
            .onAppear {  
                if !appHasData {  
                    coreDataStack.addContacts()  
                    appHasData = true  
                }  
            }  
    }  
}
```

2. Go to CoreDataStack and in the first extension, add a new function to delete a contact:

```
extension CoreDataStack {  
    //...  
  
    func deleteContact(_ contact: Contact) {  
        managedObjectContext.delete(contact)  
    }  
}
```

3. 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),
    ]
)
private var contacts: FetchedResults<Contact>
}
```

4. Switch to `ContentView` and add a new `@EnvironmentObject` variable to receive the `CoreDataStack` instance from the Environment:

```
@EnvironmentObject var coreDataStack: CoreDataStack
```

5. The `body` function will simply render the list of contacts with the `.onDelete()` modifier, which will call a `deleteContact()` function that we will define in the next step. Add the following code:

```
struct ContentView: View {
    //...
    var body: some View {
        List {
            ForEach(contacts, id: \.self) {
                ContactView(contact: $0)
            }
            .onDelete(perform: deleteContact)
        }
        .listStyle(.plain)
    }
}
```

6. Add the `deleteContact()` function, which simply fetches the `Contact` object from the list and calls the `deleteContact()` function created in *step 2*:

```
private func deleteContact(at offsets: IndexSet) {
    guard let index = offsets.first else {
        return
    }
    let contact = contacts[index]
    coreDataStack.deleteContact(contact)
    coreDataStack.save()
}
```

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, as shown in the following screenshot:

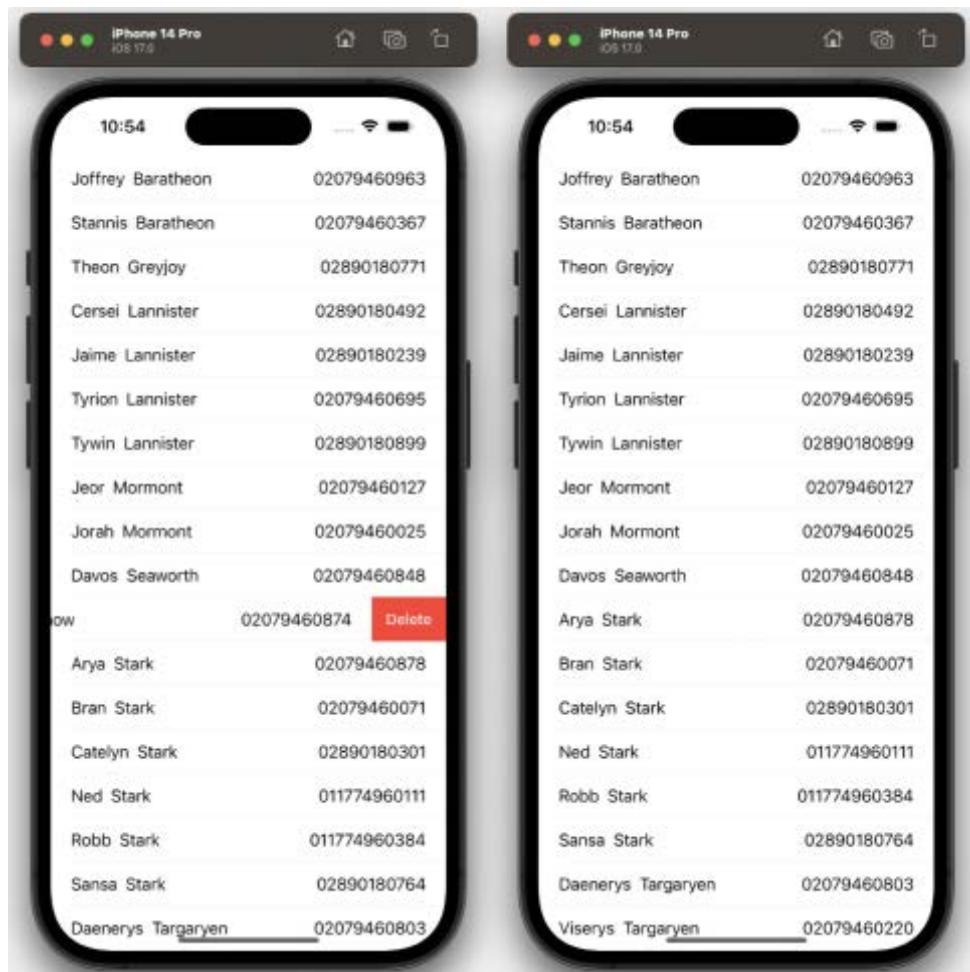


Figure 14.8: Deleting a Core Data-backed object in SwiftUI

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. After the delete operation, we save the changes to the persistent storage.

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 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`.

Keep 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 plain `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 it in sections. Think, for example, of a settings list where the settings are aggregated into types of settings, or a list of contacts, where the contacts are aggregated into sections by the first characters of surnames.

SwiftUI provides a mechanism for fetching objects from Core Data storage that's already been aggregated into sections, allowing it to be easily presented in a sectioned `List` view.

In this recipe, we'll implement a list of contacts, where the contacts are grouped into sections determined by the first character of their surname.

Getting ready

Create a SwiftUI app called `SectionedContacts`.

Before we start this recipe, make sure to complete *steps 1 to 10* 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 determined by the first characters of their surnames.

The steps are as follows:

1. Switch to `ContentView.swift`. 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) ?? "")  
        }  
    }  
}
```

2. In the `ContentView` component, add the `@SectionedFetchRequest` property wrapper to fetch the contacts using the `KeyPath` to the property created in the previous step as a section identifier:

```
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  
}
```

3. Finally, in the body function of the component, present the `Contacts` grouped into sections:

```
var body: some View {  
    NavigationStack {  
        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 that the contacts are grouped by the first character of the surname in a sectioned List view:

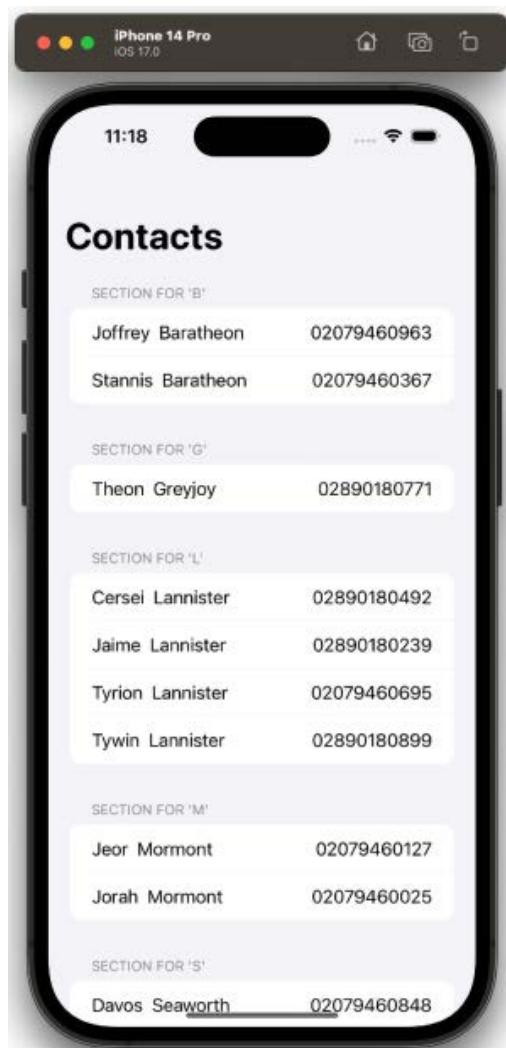


Figure 14.9: Sectioned contacts List view

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 like 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 a `keyPath`, which is passed and identifies the grouping 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 Objective-C runtime to do so, we must decorate its definition with `@objc`. Objective-C was the primary language that was used to build iOS before Swift. Some system frameworks still rely on Objective-C system libraries and conventions, such as Core Data.

Getting started with SwiftData

With the introduction of SwiftData in iOS 17, Apple provided a new framework for persisting data in our apps. With minimal code and no external dependencies, we can declare our model classes in plain Swift, using macros, while the framework handles the underlying storage for us.

At the time of writing this book, iOS 17 beta 7 has just been released, and SwiftData still remains beta software. The framework has changed with every beta release of iOS 17, and some features available in earlier versions, such as dynamic queries in iOS 17 beta 2, are no longer available. Due to these changing requirements, we chose to focus on an introduction of SwiftData and its basic features, leaving the more advanced features for a future edition of the book.

When creating a new project with Xcode 15, and choosing the iOS app template, we can select SwiftData from the Storage drop-down menu and Xcode will generate SwiftData code with a sample model class. We can inspect the code to understand how to use SwiftData in our app and then modify it to fit our needs.

In this recipe, we'll introduce SwiftData manually in a new SwiftUI iOS project, along with a minimum set of functionalities, using the contact list app developed in the previous recipes.

Getting ready

Let's create a SwiftUI app called `SwiftDataBasics`, ensuring that we choose `None` for the `Storage` option:

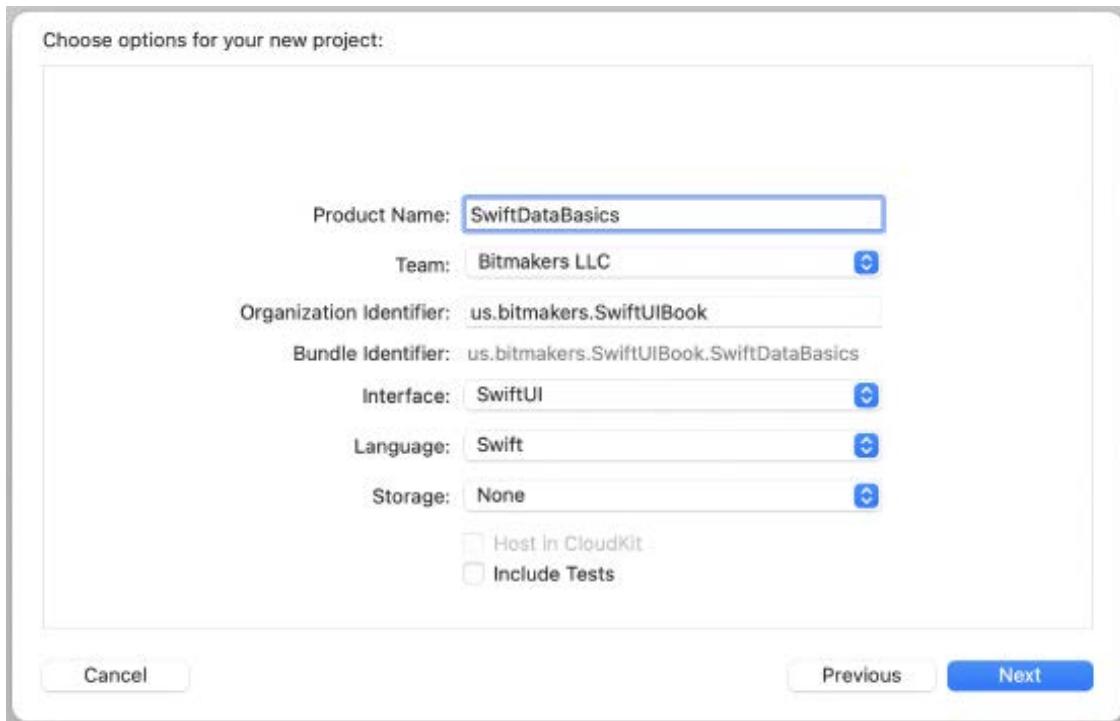


Figure 14.10: A SwiftUI app without storage

How to do it...

In this recipe, we are going to implement the contacts app used in the previous recipes, but using SwiftData as the persistence framework. The app will load a bunch of contacts only in the first launch, display the contacts in a list with swipe-to-delete functionality, and have a button in the toolbar to add new contacts. Contacts will be persisted between launches of the app, thanks to SwiftData.

Let's get started:

1. First, let's create our model. Create a new Swift file named `Contact.swift` and declare the following class:

```
import Foundation

final class Contact {
    var firstName: String
    var lastName: String
```

```
var phoneNumber: String

var fullName: String { firstName + " " + lastName }

init(firstName: String, lastName: String, phoneNumber: String) {
    self.firstName = firstName
    self.lastName = lastName
    self.phoneNumber = phoneNumber
}

}
```

2. Now let's add an extension to our class with some sample data to use in our app:

```
extension Contact {
    static var samples: [Contact] = [
        ("Daenerys", "Targaryen", "02079460803"),
        ("Bran", "Stark", "02079460071"),
        ("Jon", "Snow", "02079460874"),
        ("Theon", "Greyjoy", "02890180771"),
        ("Ned", "Stark", "011774960111"),
        ("Tyrion", "Lannister", "02079460695"),
        ("Arya", "Stark", "02079460878"),
        ("Stannis", "Baratheon", "02079460367"),
        ("Samwell", "Tarly", "011774960104"),
        ("Jaime", "Lannister", "02890180239"),
        ("Jorah", "Mormont", "02079460025"),
        ("Jeor", "Mormont", "02079460127"),
        ("Robb", "Stark", "011774960384"),
        ("Joffrey", "Baratheon", "02079460963"),
        ("Tywin", "Lannister", "02890180899"),
        ("Margaery", "Tyrell", "011774960635"),
        ("Catelyn", "Stark", "02890180301"),
        ("Viserys", "Targaryen", "02079460220"),
        ("Cersei", "Lannister", "02890180492"),
        ("Davos", "Seaworth", "02079460848"),
        ("Sansa", "Stark", "02890180764")
    ].map { firstName, lastName, phoneNumber in
        Contact(firstName: firstName, lastName: lastName, phoneNumber:
            phoneNumber)
    }
}
```

- Now let's transform our class to use SwiftData as the persistent storage framework. We will accomplish this very easily. Just import SwiftData and annotate the class declaration with the `Model()` macro:

```
import Foundation
import SwiftData

@Model final class Contact {
    // ...
}
```

- Now let's switch to the App struct and initialize the SwiftData persistent storage. You'll notice how simple it is. Replace the content of `SwiftDataBasicsApp.swift` with the following:

```
import SwiftUI
import SwiftData

@main
struct SwiftDataBasicsApp: App {
    var body: some Scene {
        WindowGroup {
            ContentView()
                .modelContainer(for: Contact.self)
        }
    }
}
```

- Now let's instruct SwiftData to load the persistent storage with our sample data, but only the first time the app launches. We will use `@AppStorage` to store a Boolean value to flag the first launch of the app. In the `App` struct, replace the code with the following:

```
        appHasData = true
    }
    case .failure(let error):
        print(error.localizedDescription)
    }
}
}
```

6. Now let's work on our views. We will start with the `ContentView` view. The code is very similar to the code used in the *Adding Core Data objects from a SwiftUI view* recipe so refer to that recipe for a deeper examination if required. We have just replaced the Core Data stack with `SwiftData`. The new code is in **Bold** typeface:

```
import SwiftUI
import SwiftData

struct ContentView: View {
    @State private var isAddNewContactPresented = false
    @Environment(\.modelContext) private var modelContext
    @Query(sort: [
        SortDescriptor(\Contact.lastName, order: .forward),
        SortDescriptor(\Contact.firstName, order: .forward)
    ])
    private var contacts: [Contact]

    var body: some View {
        NavigationStack {
            List {
                ForEach(contacts, id: \.self) { contact in
                    LabeledContent(contact.fullName) {
                        Text(contact.phoneNumber)
                            .monospacedDigit()
                            .foregroundStyle(.primary)
                    }
                }
                .onDelete(perform: deleteContact)
            }
            .navigationTitle("Contacts")
            .navigationBarTitleDisplayMode(.inline)
        }
    }
}
```

```
.toolbar {
    Button {
        isAddNewContactPresented.toggle()
    } label: {
        Image(systemName: "plus")
            .font(.headline)
    }
}

.sheet(isPresented: $isAddNewContactPresented) {
    AddNewContactView(isPresented:isAddNewContactPresented)
}
```

```
}

private func deleteContact(at offsets: IndexSet) {
    withAnimation {
        for index in offsets {
            modelContext.delete(contacts[index])
        }
    }
}
```

7. Still in `ContentView.swift`, add the following view at the end of the file. The code is very similar to the code used in the *Adding Core Data objects from a SwiftUI view* recipe so refer to that recipe for more detail if needed. We have just replaced the Core Data stack with SwiftData. The new code is in **Bold** typeface:

```
struct AddNewContactView: View {
    @Environment(\.modelContext) private var modelContext
    @Environment(\.dismiss) private var dismiss
    @State private var firstName = ""
    @State private var lastName = ""
    @State private var phoneNumber = ""

    var body: some View {
        NavigationStack {
```

```
    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")
    .toolbar {
        Button(action: saveContact) {
            Image(systemName: "checkmark")
                .font(.headline)
        }
        .disabled(isButtonDisabled)
    }
}

private var isButtonDisabled: Bool {
    firstName.isEmpty || lastName.isEmpty || phoneNumber.isEmpty
}

private func saveContact() {
    let newContact = Contact(firstName: firstName, lastName:
lastName, phoneNumber: phoneNumber)
    modelContext.insert(newContact)
    dismiss()
}
}
```

At the time of writing this chapter, there is a known issue in Xcode previews and the SwiftData app, so we will use the Simulator to test the app instead. Run the app in the iOS Simulator and play with it. Add contacts, delete contacts, and observe how the contacts persist between app launches, thanks to SwiftData.

Screenshots of these features are shown in *Figures 14.11* and *14.12*:

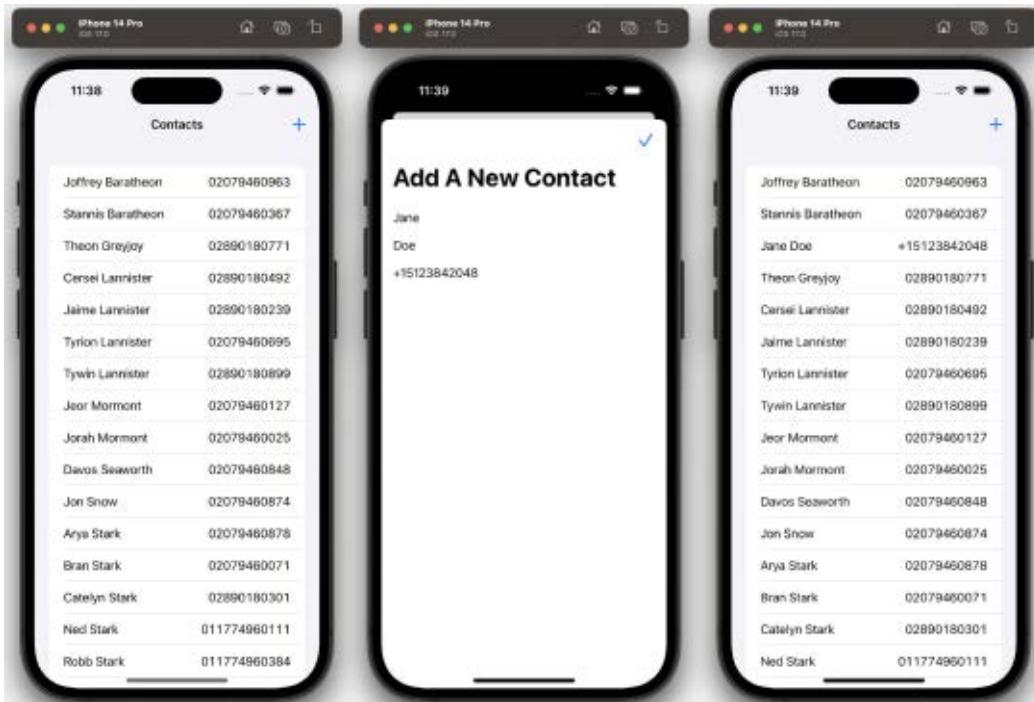


Figure 14.11: Adding a contact in our SwiftData app

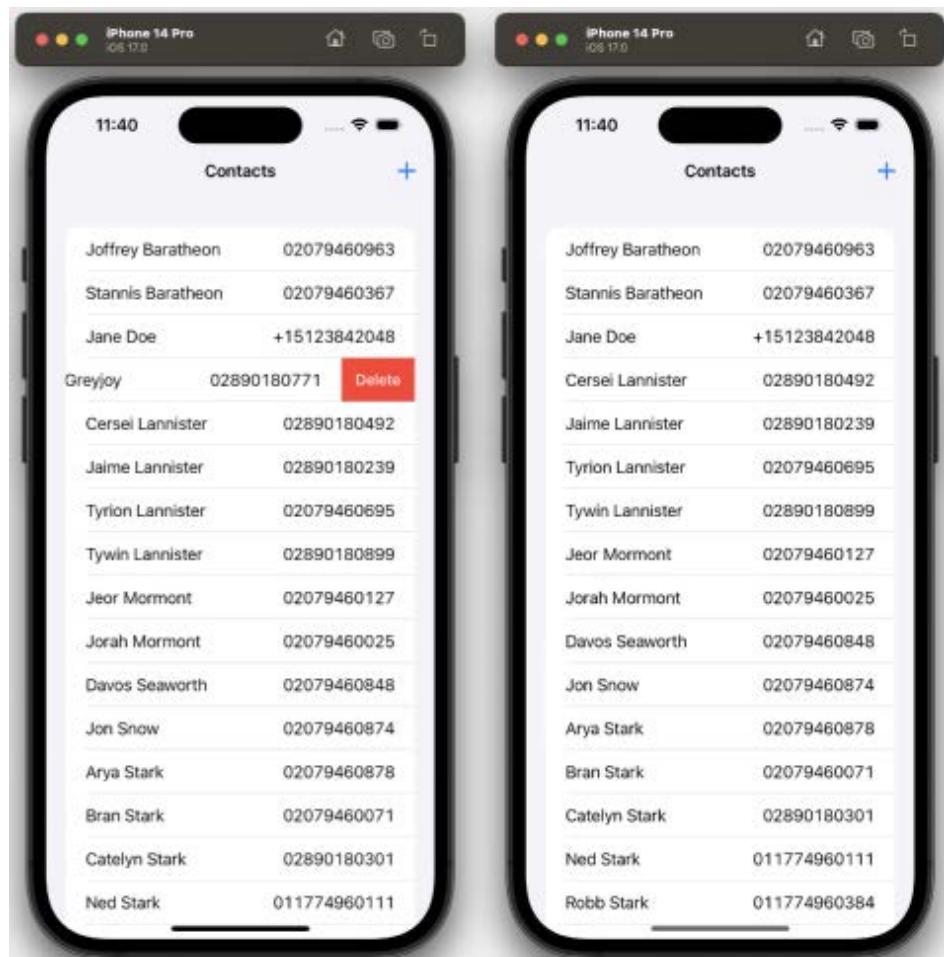


Figure 14.12: Deleting a contact in our SwiftData app

How it works...

Making our model types persistable with SwiftData is as simple as attaching the `Model()` macro to the declaration:

```
@Model final class Contact
```

You have noticed that we use a class instead of a struct for our model, as it is a requirement of SwiftData. Then, in the App struct, we use the `modelContainer(for:inMemory:isAutosaveEnabled:isUndoEnabled:onSetup:)` modifier to declare which classes need to be persisted. In its simplest form, we only need to pass the `for` parameter, which accepts a type or an array, with the types we want to persist:

```
.modelContainer(for: Contact.self)
```

In our case, we used the `onSetup` closure, which is a callback invoked after the creation of the persistent container, to introduce some initialization code and store sample data in the persistent storage:

```
.modelContainer(for: Contact.self) { result in
    switch result {
        case .success(let container):
            if !appHasData {
                for contact in Contact.samples {
                    container.mainContext.insert(contact)
                }
                appHasData = true
            }
        case .failure(let error):
            print(error.localizedDescription)
    }
}
```

To use the persistent storage in our views, we use the `Environment` value `modelContext`, which is injected in the environment by the `modelContainer()` modifier:

```
@Environment(\.modelContext) private var modelContext
```

Then, we can use the methods in `modelContext` to add or delete items from the persistent storage:

```
modelContext.insert(newContact)
modelContext.delete(contacts[index])
```

If we want to query the persistent storage, we use the `Query()` macro, which has several parameters. In our case, we used some sort descriptors to retrieve the contacts in a specific order:

```
@Query(sort: [
    SortDescriptor(\Contact.lastName, order: .forward),
    SortDescriptor(\Contact.firstName, order: .forward)
])
private var contacts: [Contact]
```

Observe how with SwiftData, we didn't use any of the Objective-C classes, thanks to the help of macros.

There's more...

SwiftData is a very powerful framework and I believe once it is fully released and improved, it could completely replace the Core Data code in our apps. Moreover, since it uses the Core Data persistent storage, it can coexist with Core Data and provide a path to a gradual adoption of the framework, like what happened with the introduction of Swift and its interoperability with Objective-C, or with the introduction of SwiftUI and its compatibility with UIKit.

SwiftData allows for automatic and manual schema migrations, automatic synchronization with iCloud, relationships between model classes with cascade rules, in-memory persistent containers, ideal for previews, and powerful filtering thanks to the `Predicate(_:)` macro, which allows us to use standard Swift to define the filtering conditions.

See also

- Apple's official documentation: <https://developer.apple.com/documentation/swiftdata>
- Preserving your app's model data across launches: <https://developer.apple.com/documentation/swiftdata/preserving-your-apps-model-data-across-launches/>
- Adopting SwiftData for a Core Data app: https://developer.apple.com/documentation/coredata/adopting_swiftdata_for_a_core_data_app
- Predicate documentation: <https://developer.apple.com/documentation/foundation/predicate>

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

<https://packt.link/swiftUI>



15

Data Visualization with Swift Charts

Data visualization is the graphical representation of information and data using visual elements like charts, graphs, and maps. It is an excellent tool to convey information to our users and present data to non-technical users. With data visualization, it is easier to spot trends, patterns, and outliers in our dataset.

When iOS 16 was announced at **Worldwide Developer Conference (WWDC) 22**, Apple introduced Swift Charts, which is a SwiftUI framework for data visualization. With Swift Charts, we can build customizable charts with little extra code, using the familiar SwiftUI syntax that we all know and love.

Building charts and graphs is easily accomplished by combining just a few building blocks: marks, scales, axes, and legends. And of course, in a native SwiftUI way, we can use view modifiers to further customize these building blocks. Swift Charts provides a native solution for data visualization built-in in SwiftUI, with multiplatform support. We no longer need to use third-party frameworks for our charts, similar to what happened when Combine was introduced and there was no longer the need to use RxSwift for asynchronous programming.

If you're new to Swift Charts, we recommend you read the first three recipes before you read the interactive charts recipes at the end of the chapter. In the first three recipes, you'll learn the building blocks in detail, and it will be easier to follow the other two recipes with the acquired knowledge.

In this chapter, we will cover the following recipes:

- Understanding the basics of Swift Charts
- Customizing charts: axes, annotations, and rules
- Different types of charts: marks and mark configuration
- Histograms with data bins
- Pie charts and donut charts

- Interactive charts: selection
- Interactive charts: scrollable content

Technical requirements

The code in this chapter is based on Xcode 15.0 and iOS 17.0. You can download and install the latest version of Xcode from the App Store. You'll also need to be running macOS Ventura (13.5) or newer.

Simply search for Xcode in the App Store, and select and download the latest version. Launch Xcode, and follow any additional installation instructions that your system may prompt you with. Once Xcode has fully launched, you're ready to go.

All the code examples for this chapter can be found on GitHub at <https://github.com/PacktPublishing/SwiftUI-Cookbook-3rd-Edition/tree/main/Chapter15-Data-Visualization-with-Swift-Charts>.

Understanding the basics of Swift Charts

In this recipe, we are going to introduce the basic building blocks of Swift Charts. We will create some charts to display the results of a hypothetical survey, conducted with a questionnaire, in a graphical way. We will start with a simple chart displaying the number of answers for each question, and then add a couple of detailed charts, to see the different answers to each question.

Getting ready

Create a new SwiftUI iOS app named `SwiftChartsBasics`.

How to do it...

Our app will have just one view, which will include the charts. We will also have a data layer in a separate file. Let's start working on the data first, and then switch to the view. These are the steps:

1. Create a new Swift file named `SurveyEntry.swift`. This file will contain our data layer. Start by creating a couple of `String` enumerations with implicitly assigned raw values, one to model our questions and one to model the answers:

```
enum Question: String {
    case first
    case second
    case third
}

enum Answer: String {
    case yes
    case no
    case maybe
    case declined
}
```

- Now, let's define a type to model our survey entries. This type will include the question, the answer, and the number of answers received. We will also declare conformance to the `Identifiable` protocol so that we can use the type in a `ForEach` view:

```
struct SurveyEntry: Identifiable {
    var id = UUID()
    var question: Question
    var answer: Answer
    var count: Int
}
```

- To finish our data layer, define the `static` variable with some sample data representing the survey results, in an extension to our type:

```
extension SurveyEntry {
    static let sampleSurvey: [SurveyEntry] = [
        .init(question: .first, answer: .yes, count: 12),
        .init(question: .first, answer: .no, count: 7),
        .init(question: .first, answer: .maybe, count: 5),
        .init(question: .first, answer: .declined, count: 1),
        .init(question: .second, answer: .yes, count: 7),
        .init(question: .second, answer: .no, count: 4),
        .init(question: .second, answer: .maybe, count: 9),
        .init(question: .second, answer: .declined, count: 0),
        .init(question: .third, answer: .yes, count: 4),
        .init(question: .third, answer: .no, count: 8),
        .init(question: .third, answer: .maybe, count: 0),
        .init(question: .third, answer: .declined, count: 3)
    ]
}
```

- Switch to `ContentView.swift` to start working on our views. First, since we will be working with the `Charts` framework, add an import statement at the top of the file. Second, after the `ContentView` struct declaration, add a private variable to store the data that we are going to use to plot the charts:

```
import Charts

struct ContentView: View {
    private var data: [SurveyEntry] = SurveyEntry.sampleSurvey
    // ...
}
```

5. The starting code for our view is going to be a `VStack`, with a `Text` and an empty `Chart`. The `Chart` view by default will fill the space of its parent view. To change this behavior, we will use an `.aspectRatio()` modifier. Replace the body variable with the following code:

```
var body: some View {
    VStack {
        Text("Swift Charts Basics")
            .foregroundStyle(Color.accentColor)
            .font(.title)
        Chart {
        }
        .aspectRatio(contentMode: .fit)
    }
    .padding(.horizontal)
}
```

6. We will draw a bar chart to display how many answers we got for each of the questions in the survey. We will use the horizontal axis, or *x-axis*, to display the questions and the vertical axis, or *y-axis*, to display the number of answers we received. Our `Chart` view is empty for now, so let's replace it with the following:

```
Chart {
    ForEach(data) { entry in
        BarMark(
            x: .value("Question", entry.question),
            y: .value("Count", entry.count)
        )
    }
}
```

7. Xcode complains of a couple of errors. To fix these errors, the data used in the chart needs to conform to the `Plottable` protocol. We use the `question` and `count` properties of our data entries in the `BarMark`. If you look at the two properties, the `count` property is of type `Int`, and the `question` property is of the `Question` type. `Int` already conforms to the protocol, and `Question` is an enum with implicitly assigned `String` values. Since `String` already conforms to `Plottable`, we can declare the conformance of `Question` to the protocol without implementing any extra code. Include this code at the top of the file, before the `struct ContentView`:

```
extension Question: Plottable {}
```

8. After implementing this conformance, Xcode should not complain anymore, and we should see a chart in the live preview of the canvas. The preview should look like this:



Figure 15.1: Our basic chart preview

9. With the basic chart working, let's implement a couple more charts that will display the breakdown of the answers received for each question. First, we are going to create an enumeration to manage which of the charts will be displayed. Add the following code at the top of the file, *before* the conformance of `Question` to `Plottable`:

```
enum BarChartType {  
    case aggregate  
    case stacked  
    case sideBySide  
}
```

10. Create a `@State` variable to keep track of the chart chosen by the user. Add the following code right after the variable that stores our data:

```
@State private var typeOfChart: BarChartType = .aggregate
```

11. Add a picker inside the VStack, between the Text and the Chart views, so that the user can choose the type of chart displayed:

```
Picker("Type of Chart", selection: $typeOfChart) {
    Text("Aggregate").tag(BarChartType.aggregate)
    Text("Stacked").tag(BarChartType.stacked)
    Text("Side-by-side").tag(BarChartType.sideBySide)
}
.pickerStyle(.segmented)
.padding(.bottom)
```

12. Modify the chart to display a different chart, depending on the user's choice. We will implement a switch statement on the value of the variable `typeOfChart` to accomplish this. The new chart should look like the following code:

```
Chart {
    ForEach(data) { entry in
        let mark = BarMark(
            x: .value("Question", entry.question),
            y: .value("Count", entry.count)
        )
        switch typeOfChart {
        case .aggregate:
            mark
        case .stacked:
            mark
                .foregroundStyle(by: .value("Answer", entry.answer))
        case .sideBySide:
            mark
                .foregroundStyle(by: .value("Answer", entry.answer))
                .position(by: .value("Answer", entry.answer))
        }
    }
}
.aspectRatio(contentMode: .fit)
```

13. Xcode will complain, and the app won't build. The reason for this, despite the cryptic error message, is because we are using `entry.answer` of type `Answer` in the chart, and `Answer` does not conform to the `Plottable` protocol. Like we did with `Question`, add the conformance to the protocol at the top of the file:

```
extension Question: Plottable {}
extension Answer: Plottable {}
```

14. We have finally finished our app. Play with the app in the live preview of the canvas, and click on the picker to change the displayed chart. If everything went well, then the preview should look like the following figure:



Figure 15.2 Preview of the final app

How it works...

When using charts for data visualization, we must follow two fundamental steps:

1. Prepare our data so that it can be easily used in a chart.
2. Use the Swift Charts framework building blocks to accomplish the desired visualization.

Preparing data for visualization in a chart is a discipline by itself, and we won't explain it in this chapter, since it falls outside of iOS development. However, we will explain how to get the data ready for Swift Charts.

As part of this first step, to represent the data of our survey, we created the `struct SurveyEntry`. We made this `struct` conform to `Identifiable` so that we could use instances in a `ForEach` view, embedded in the `Chart` view. Additionally, for the properties of `SurveyEntry` used in the chart, we implemented the conformance to the `Plottable` protocol. Since fundamental types already conform to this protocol, we declared conformance for the couple of custom `enum` types used. However, since these `enum` store `String` raw values, and `String` already conforms to `Plottable`, we got the conformance to the protocol automatically.

For the second step, in our survey data, we had three questions, so we decided to use the horizontal axis, also known as the *x-axis*, to plot the questions. For the vertical axis, also known as *y-axis*, we chose to display the number of answers to the specific question. We used `BarMark` instances to plot a bar chart. `BarMark` is one of the many graphical marks that Swift Charts provides. Marks are the fundamental building block of a chart, and just by using a mark, we get a nice-looking chart. The framework automatically takes care of the scales, axes, axis labels, and colors for the marks. For example, when we used:

```
Chart {
    ForEach(data) { entry in
        BarMark(
            x: .value("Question", entry.question),
            y: .value("Count", entry.count)
        )
    }
}
```

we just specified four things:

- A dataset
- A mark in the form of a bar chart
- Plotting the questions in the x-axis
- Plotting the count of the answers in the y-axis

And the `Chart` view drew the chart in Figure 15.3. We have added some notes in the figure to explain different parts of the chart:

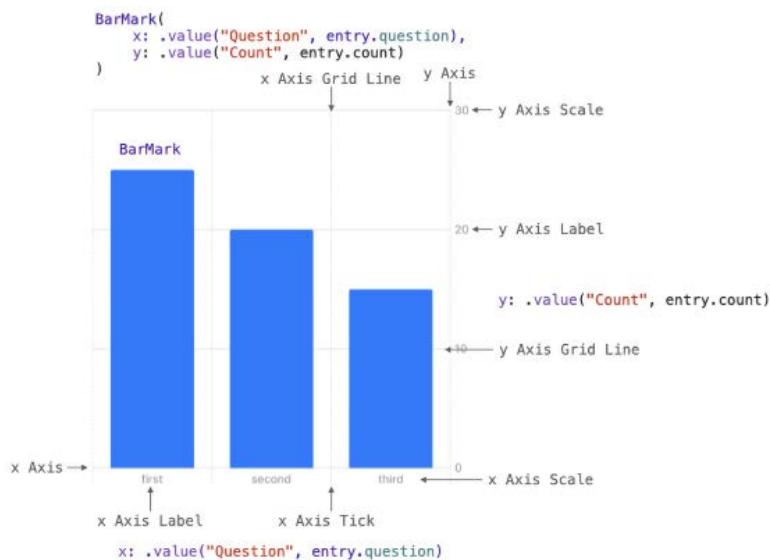


Figure 15.3: Our basic chart with the notes about the different building blocks

Let's break down how it works, with the help of the preceding figure.

1. Our dataset comes from `private var data: [SurveyEntry] = SurveyEntry.sampleSurvey`. If we look at the sample data, we'll see that it is an array of 12 entries. We have 3 questions, and each question has 4 possible answers, so that gives us the 12 samples. How did we get 3 bars?
2. We got 3 bars because we used `BarMark` and we instructed it to plot the property `question` for the *x-axis*. Since we have 3 questions, we got 3 bars. For the *y-axis*, we wanted to plot the `count` property. Swift Charts takes care of the aggregate count for us and adds all the data points for a specific question, so we get the total count of the answers to the specific question, regardless of the type of answer. For example, by inspecting our sample data, we see that the second question has a total of 20 answers; 7 are *yes*, 4 are *no*, 9 are *maybe*, and none are *declined*. If we look at the chart, we can see that the bar for the second question has a height of 20, precisely the total count. The same reasoning applies to the other two questions. Since we are representing the same value for each bar, the `Chart` view picked the same color for all the bars.
3. Since we used the *x-axis* for the questions and `question` is an `enum` with `String` raw values, the raw value is what is used for the axis. Inspecting our data, we have 3 cases in the `enum`, and each one has the following raw values: *first*, *second*, *third*. This is why we automatically got the 3 labels on the axis. Additionally, we got ticks and grid lines for free.
4. For the *y-axis*, we use the `count` property of type `Int`, so the `Chart` calculated the properties of the *y-axis* automatically for us. We got a scale, which goes from 0 to 30, and we got grid lines and labels for every 10 units: 0, 10, 20, 30. All this automatically!

Now that we understand how a `Chart` view works, let's see how we managed to display the other two charts, where for each question we have a breakdown of its answers.

First, inside the content closure of the `Chart` view, we used a `switch` statement on the `enum` variable `typeOfChart` to create three different `BarMark` instances, one for each case of the `enum`. Since the difference among the three marks is the applied view modifiers, we started by storing the `BarMark` instance in a local variable named `mark`. This allowed us to reuse the code for the mark without having to repeat it.

For the *Aggregate* chart, we just used the `mark`, unmodified. We have explained this chart extensively in this section.

For the *Stacked* chart, we applied the `.foregroundStyle(by: .value("Answer", entry.answer))` modifier to the `mark`. Since our answer is an `enum` with 4 cases, and each case stores a `String` raw value, we modify the foreground style of the mark, depending on the value of the answer. The `Chart` view automatically chooses a color for each of the answers, takes care of the color of the bar, and adds a legend, below the *x-axis*, with the equivalence between the colors and the values of the answers. You can see this chart in the center of *Figure 15.2*.

For the *Side-by-side* chart, we applied two modifiers to the `mark`: the `.foregroundStyle(by: .value("Answer", entry.answer))` and the `.position(by: .value("Answer", entry.answer))`. The second modifier tells the `Chart` view to position the bar on the *x-axis*, depending on the value of the answer. Thanks to this second modifier, we plotted the bars side by side instead of stacking them. You can see this chart on the right side of *Figure 15.2*.

As you can see, Swift Charts is a very powerful framework, and with a few lines of code, we plotted three beautiful bar charts, as shown in *Figure 15.2*. The framework took care of automatically calculating the scale for both axes, the labels and grid lines, the coloring of the bars, and even the color-coded legend in the detailed bar charts.

See also

- Chart view documentation: <https://developer.apple.com/documentation/charts/chart>
- ChartContent protocol documentation: <https://developer.apple.com/documentation/charts/chartcontent>
- ChartContentBuilder result builder documentation: <https://developer.apple.com/documentation/charts/chartcontentbuilder>
- To learn more about bar charts check the BarMark documentation: <https://developer.apple.com/documentation/charts/barmark>

Customizing charts: axes, annotations, and rules

In this recipe, we are going to learn how to customize a chart to make it look more appealing. We will use a bar chart to plot the average rainfall for the city of Austin, TX, for each month of the year 2022. We will use the x-axis to plot the month, and the y-axis to plot the quantity. We will customize the axis scales, labels, and gridlines, add custom annotations to the bars, and add a rule with custom text. To make it more convenient for the user, we will introduce a control to switch between European units (millimeters) and US units (inches).

Getting ready

Create a new SwiftUI iOS app named `ChartCustomizations`.

How to do it...

Our app will have just one view, which will include the charts. We will also have a data layer in a separate file. Let's start working on the data first, and then switch to the view. These are the steps:

1. Drag and drop the `climate_data.json` file from the book's Resources folder at GitHub (<https://github.com/PacktPublishing/SwiftUI-Cookbook-3rd-Edition/tree/main/Resources/Chapter15/Recipe02>) into the project's main folder. Make sure the file is added to the target.
2. Inspect the JSON file to understand how our data is organized. We can see that the top-level object is an array of three objects, each representing the climate of a city. The object has two properties; the first, named `city`, contains the name of the city, and the second, named `climate`, is an array of custom objects representing the climate data for a specific month. We will model our data with two types, one representing the cities in the top-level array, and the other representing the data for each month of the year:

```
[  
 {  
   "city": "Austin, TX",  
 }
```

```
"climate": [
  {
    "month": "Jan-22",
    "rainfall": 56,
    "high": 16,
    "low": 5
  },
  ...
]
},
...
]
```

3. Create a new Swift file named `CityClimate.swift`. This file will contain our data layer. To represent the climate data, we will use the `Measurement` structure, which includes unit conversion functions that we will use in our app. Start by creating a `struct ClimateData` to model the climate data for a month:

```
struct ClimateData {
    let date: Date
    let rainfall: Measurement<UnitLength>
    let maxTemperature: Measurement<UnitTemperature>
    let minTemperature: Measurement<UnitTemperature>
}
```

4. Since we are going to decode the data for `ClimateData` from a JSON file, let's create an extension to declare the conformance to the `Decodable` protocol. We will need to implement custom decoding because our property names are different than the ones in the file, and because the file contains numbers, but we must decode the `Measurement` instances:

```
extension ClimateData: Decodable {
    enum CodingKeys: String, CodingKey {
        case date = "month"
        case rainfall
        case maxTemperature = "high"
        case minTemperature = "low"
    }

    init(from decoder: Decoder) throws {
        let container = try decoder.container(keyedBy: CodingKeys.self)
        date = try container.decode(Date.self, forKey: .date)
        rainfall = try Measurement(
            value: container.decode(Double.self, forKey: .rainfall),
            unit: UnitLength.meters)
        maxTemperature = try Measurement(
            value: container.decode(Double.self, forKey: .high),
            unit: UnitTemperature.degreesCelsius)
        minTemperature = try Measurement(
            value: container.decode(Double.self, forKey: .low),
            unit: UnitTemperature.degreesCelsius)
    }
}
```

```

        unit: .millimeters
    )
maxTemperature = try Measurement(
    value: container.decode(Double.self, forKey:
.maxTemperature),
    unit: .celsius
)
minTemperature = try Measurement(
    value: container.decode(Double.self, forKey:
.minTemperature),
    unit: .celsius
)
}
}
}

```

5. For our city, we will create the `struct CityClimate` and declare conformance to the `Decodable` protocol:

```

struct CityClimate: Decodable {
    let city: String
    let climate: [ClimateData]
}

```

6. We are ready now to read the sample data from the JSON file and decode it into our own struct instances. A couple of things to mention are that we are getting the file from the main bundle, and we need a custom `DateFormatter` to decode the dates. The code should be this:

```

extension CityClimate {
    static var sample: [CityClimate] = {
        guard let url = Bundle.main.url(forResource: "climate_data",
withExtension: "json"),
            let data = try? Data(contentsOf: url)
        else {
            return []
        }
        let dateFormatter = DateFormatter()
        dateFormatter.dateFormat = "MMM-yy"
        let decoder = JSONDecoder()
        decoder.dateDecodingStrategy = .formatted(dateFormatter)
        let array = try? decoder.decode([CityClimate].self, from: data)
        return array ?? []
    }()
}

```

7. To finish with our model layer, let's add a couple of computed properties that will be used for our chart. The first one will return the year of the dataset, and the second one will calculate the average rainfall for the year. Implement the following code:

```
extension CityClimate {  
    var year: Int {  
        Calendar.current.dateComponents([.year], from: climate[0].date).  
        year!  
    }  
    var averageRainfall: Measurement<UnitLength> {  
        let initial = Measurement(value: 0, unit: UnitLength.meters)  
        let average = climate.map {  
            $0.rainfall.converted(to: .meters)  
        }.reduce(initial, +) / Double(climate.count)  
        return average  
    }  
}
```

8. We want to draw a bar chart to display the rainfall for the city of Austin, TX, in 2022. By the inspection of our dataset, we can see that we have monthly data for three cities and that Austin is the first city in the array. Let's go to `ContentView.swift` and declare three variables. The first one is to store our sample data. The second will be used in a `Toggle` control, indicating a selection between US measurement units and international units. The third will be a computed variable to return the unit of length used to measure the rainfall, according to the user's preferences. The code should look like this:

```
struct ContentView: View {  
    private var city = CityClimate.sample[0]  
    @State private var isUS = true  
    private var unit: UnitLength { isUS ? .inches : .millimeters }  
    var body: some View {  
        // ...  
    }  
}
```

9. Now, let's work on the `ContentView` view. We will use a `VStack` and embed a couple of `Text` views at the top, then the `Chart` view at the middle, and a `Toggle` view at the bottom to switch between US and international units. For the chart, we will use `BarMark` and plot the months in the x-axis and the rainfall in the y-axis. Replace the content of the `body` property with the following:

```
var body: some View {  
    VStack {  
        Text("Average Rainfall (\(unit.symbol))")
```

```
        .font(.title)
Text("for \(city.city) in \(String(format: "%d", city.year))")
Chart {
    ForEach(city.climate, id:\.date) { monthlyData in
        let rainfall = monthlyData.rainfall.converted(to: unit)
        BarMark(x: .value("Month", monthlyData.date, unit:
.month),
            y: .value("Rain", rainfall.value))
    }
}
.aspectRatio(contentMode: .fit)
HStack {
    Text("EU") // EU flag
    Toggle("", isOn: $isUS)
        .labelsHidden()
    Text("US") // US flag
}
.font(.largeTitle)
}
.padding()
}
```

10. Use the live preview on the canvas to interact with the app. Notice how when we click on the toggle, the title at the top changes to display the unit chosen, and the scale and labels of the y-axis change too. The preview should look like this:



Figure 15.4: Preview with the two different unit choices

11. With the basic chart working, we will now customize the bars. First, we will change the foreground style of the bars, giving them a gradient color and rounded corners. We can accomplish this with two view modifiers applied to the BarMark. It should look like the following code after the changes:

```
BarMark(x: .value("Month", monthlyData.date, unit: .month),
        y: .value("Rain", rainfall.value))
    .foregroundStyle(Gradient(colors: [.blue, .blue.opacity(0.4)]))
    .cornerRadius(4)
```

12. Now, we would like to display the value of the rainfall for each month inside the corresponding bar. To achieve this, we will use an annotation. Add the following code right after the `cornerRadius` modifier:

```
.annotation(alignment: .center, spacing: -20) {
    Text(String(format: isUS ? "%.1f" : "%.0f", rainfall.value))
        .font(.caption2)
        .foregroundStyle(.white)
}
```

13. Use the live preview in the canvas to test the app. Notice how changing the value of the toggle changes the value of the annotations. It should look like the following screenshots:



Figure 15.5: Previews after the bar customization

14. Now, we will continue by adding a horizontal rule with the average rainfall for the year, so we can see, in a graphical way, which months are above or below the average. We will use a new type of mark for this purpose, a `RuleMark`. Since this rule applies to the chart content, but not to the individual bars, the code will be outside the `ForEach` struct. It should look like the following:

```
ForEach(city.climate, id:\.date) { monthlyData in
    //...
```

```
    }
    let averageRainfall = city.averageRainfall.converted(to: unit)
    RuleMark(y: .value("Average", averageRainfall.value))
        .foregroundStyle(.gray)
        .lineStyle(.init(dash: [4]))
```

15. As we saw with the bars, marks can be customized further with annotation modifiers. Let's add an annotation to the rule to display the average rainfall value too. Add the following modifier after the `lineStyle` modifier used in the previous step:

```
.annotation(alignment: .topLeading, spacing: 0) {
    Text("Average = " + String(format: "%.1f", averageRainfall.value))
        .font(.caption)
        .foregroundStyle(.gray)
        .background(.white.opacity(0.2))
}
```

16. Use the live preview in the canvas to test our changes. It should look like the following screenshots:



Figure 15.6: Previews after adding the horizontal rule

17. It's time to work on the customization of the horizontal axis. Basically, we will remove the gridlines and ticks and provide a label for each month. Due to space constraints, we will format the month to use the first initial for the labels. To customize the x-axis, we will use a `chartXAxis(content:)` modifier applied to the `Chart` view. The code should be like the following:

```
Chart {  
    //...  
}  
.aspectRatio(contentMode: .fit)  
.chartXAxis {  
    AxisMarks(values: city.climate.map { $0.date } ) { _ in  
        AxisValueLabel(format: .dateTime.month(.narrow),  
horizontalSpacing: 11)  
    }  
}
```

18. To see the possibilities of the axis label formatting, let's add another `AxisValueLabel` instance to our `AxisMarks` content:

```
AxisValueLabel(format: .dateTime.month(.abbreviated), orientation:  
.vertical, horizontalSpacing: 8, verticalSpacing: -25)  
.foregroundStyle(.black)
```

19. Use the live preview in the canvas to test our changes. It should look like the following screenshot:



Figure 15.7: Previews after customizing the x-axis

20. Comment out or remove the code from *step 18* so that we only have one set of labels for the x-axis. Now, we will start the customization of the y-axis. We will choose the range of values for the rainfall quantity and which values we want to display a label for. Since we want these values to be different depending on the unit chosen by the user, we will use computed variables.

Add the following code right after the `chartXAxis` modifier:

```
.chartXAxis {
    //...
}

.chartYAxis {
    var y: Double { isUS ? 5 : 120 }
    var step: Double { isUS ? 1 : 20 }
    let format = Decimal.FormatStyle.number.precision(.fractionLength(1))
    AxisMarks(values: Array(stride(from: 0, through: y, by: step))) { _ in
        AxisGridLine()
        AxisTick()
        AxisValueLabel(format: format)
    }
}
```

21. To show how powerful the axis customization in Swift Charts is, let's add a second set of `AxisMarks` to the y-axis, displaying the gridlines between the labels. The code should look like the following:

```
.chartYAxis {
    //...
    AxisMarks(values: Array(stride(from: 0, through: y, by: step))) { _ in
        //...
    }
    AxisMarks(values: Array(stride(from: 0, through: y, by: step/5))) { value in
        if let rainfall = value.as(Double.self), fmod(rainfall, step) != 0 {
            AxisGridLine(
                stroke: StrokeStyle(lineWidth: 0.5, dash: [5, 2.5],
                dashPhase: 0)
            )
        }
    }
}
```

22. Use the live preview in the canvas to test our y-axis customizations. It should look like the following screenshots:



Figure 15.8: Previews after customizing the y-axis

23. To finish the recipe, we will customize the background of the chart. If we apply the `background` modifier to the `Chart` view, this will include the chart, the axis, and the axis labels. However, if we want to modify the background of the plot area, we need to use the `chartPlotStyle(content:)` modifier. Let's add the following code after the `chartYAxis` modifier:

```
.chartYAxis {  
    // ...  
}  
.chartPlotStyle { content in  
    content  
        .background {  
            Color.yellow.opacity(0.2)  
        }  
        .border(.brown, width: 0.5)  
}
```

24. Now, we have finally finished our app. Play with the app in the live preview of the canvas and click on the toggle to change the units. The previews should look like this:



Figure 15.9: Final previews of the app

How it works...

Remember the two-step process when using charts for data visualization: prepare the data so that it can be easily used in a chart, and then use the Swift Charts framework to accomplish the desired visualization.

At the start of the recipe, we prepared our data so that it could be easily used in our chart. We started from a text file, with our dataset encoded in the JSON format. Usually, in real-world apps, the JSON content is obtained by making a call to a remote service, but in our case, for simplicity, we added a JSON file to the project.

As part of preparing the data for visualization in a chart, we created two `struct` types, `ClimateData` and `CityClimate`. Of relevance is the use of `Measurement` structs to represent our data for the rainfall and temperatures.

Because of this use, we had to implement custom code for the conformance of `ClimateData` to the `Decodable` protocol, while for `CityClimate`, we used automatic conformance. Check the link in the next section to learn more about decoding custom types in Swift.

We added a couple of computed variables to `CityClimate`: `year` to obtain the year of the climate data-set, and `averageRainfall` to calculate the average of the rainfall data. We used the `year` value in the second `Text` view at the top of the chart, and the `averageRainfall` value to draw the horizontal rule and display its value as an annotation. It is a good practice to perform calculations in the data layer and keep the view layer just for presentation code.

Our root view layout was achieved with a `VStack` with two text views at the top, the chart in the middle, and an `HStack` with two icons and a `Toggle` control at the bottom. This control used a binding to the `@State` variable `isUS`, which, when set to `true`, used US units for the chart, or international units when set to `false`.

Our chart used a `ForEach` view, iterating over the set of climate data for the city of Austin. For the chart, we used `BarMark` to plot the month on the x-axis and the rainfall quantity on the y-axis. Note how when we wanted to use the value of the rainfall, we used:

```
let rainfall = monthlyData.rainfall.converted(to: unit)
```

to convert the measurement to the correct unit, thanks to the `Measurement` structure. The computed variable `unit` provided the correct unit, depending on the measurement system chosen by the user:

```
private var unit: UnitLength { isUS ? .inches : .millimeters }
```

In our marks, for the `Plottable` value for the y-axis of the chart, we used the `value` property of the `rainfall`, which returned a `Double` that already conforms to `Plottable`:

```
.value("Rain", rainfall.value)
```

For the `Chart` view customization, we started by customizing the `BarMark` with a `foregroundStyle` modifier to use a gradient of blue, and a `cornerRadius` modifier to set the corner radius. We used an `annotation` modifier to display the value at the top of each bar:

```
.annotation(alignment: .center, spacing: -20)
```

The `alignment` parameter centered the annotation horizontally, and the `-20` value for the `spacing` parameter moved the annotation down, inside the bar. For the `@ViewBuilder` parameter's content, which is initialized in the trailing closure, we used a `Text` view with some modifiers:

```
Text(String(format: isUS ? "%.1f" : "%.0f", rainfall.value))
    .font(.caption2)
    .foregroundStyle(.white)
```

We could have used any type of view for the annotation content. This is the power of SwiftUI's `ViewBuilder` struct: to build any type of view from a closure.

We continued our customization and used a `RuleMark` to draw a horizontal line with the value of the average rainfall for the year:

```
RuleMark(y: .value("Average", averageRainfall.value))
```

The rule provided good visual support for our chart, as the user could easily see if a month's rainfall was above or below the average rainfall for the year. We also added an annotation to the rule. Its configuration is like the configuration used for the annotations on the bars and has already been explained.

For the x-axis customization, we used the `chartXAxis` modifier, with custom `AxisMarks` that overrode the default markers for the x-axis:

```
AxisMarks(values: city.climate.map { $0.date } )
```

The values used in the x-axis are of type `Date`, so we transformed the climate array in an array of `Date` instances, which we passed to the `values` parameter of the `AxisMarks` initializer. This allowed us to have exactly 12 elements in the array, each one representing the date of the first day of the month. For the content parameter of the `AxisMarks` initializer, which was used in the form of a trailing closure, we only used `AxisValueLabel` for custom labels, but we omitted `AxisGridLine` and `AxisTick`, so the gridlines and ticks were not displayed:

```
AxisValueLabel(format: .dateTime.month(.narrow), horizontalSpacing: 11)
```

We used the `format` parameter of the `AxisValueLabel` initializer to format the 12 dates in the `values` array as the narrow form of a month, which resulted in using the first letter of each month.

For the y-axis, we used two `AxisMarks` instances. The first one provided custom labels, ticks, and gridlines:

```
AxisMarks(values: Array(stride(from: 0, through: y, by: step))) { _ in
    AxisGridLine()
    AxisTick()
    AxisValueLabel(format: format)
}
```

Let's explain how this works in detail. The values used in the y-axis are of type `Double`. For the `values` parameter in the `AxisMarks` initializer, we used an `Array` of `Double` and initialized it using a `stride` function:

```
Array(stride(from: 0, through: y, by: step))
```

The `stride` function produced `[0, 1, 2, 3, 4, 5]` for the US units, and `[0, 20, 40, 60, 80, 100, 120]` for the international units. We used computed variables depending on the `@State` var `isUS`:

```
var y: Double { isUS ? 5 : 120 }
var step: Double { isUS ? 1 : 20 }
```

So, when `isUS` changes its value, the variables are computed again, the `stride` is computed again, and the chart changes the y-axis customization.

For the value label formatting, we used a `Decimal.FormatStyle` of one fraction digit:

```
let format = Decimal.FormatStyle.number.precision(.fractionLength(1))
```

For the second set of `AxisMarks`, we used a `stride`, with a step that was $1/5$ of the step of the `stride` used in the first `AxisMarks` instance:

```
Array(stride(from: 0, through: y, by: step/5))
```

This produced five times more values than the values produced in the first `stride`. For example, for the US unit selection, the array of values produced was $[0, 0.2, 0.4, 0.6, 0.8, 1, 1.2, \dots, 4.6, 4.8, 5]$. Since the values with no fraction digits overlap with the values used in the first `AxisMarks` instance, we used a conditional to only use the values with a non-zero fractional part:

```
if let rainfall = value.as(Double.self), fmod(rainfall, step) != 0 {...}
```

We used this second instance of `AxisMarks` only to plot gridlines, and we omitted ticks and labels:

```
AxisGridLine(  
    stroke: StrokeStyle(lineWidth: 0.5, dash: [5, 2.5], dashPhase: 0)  
)
```

Using the example of the US units, and to summarize our y-axis customization, the first `AxisMarks` instance was used to plot the labels with values $1.0, 2.0, 3.0, 4.0$, and 5.0 , with ticks and solid grid lines at these values. Also, the second `AxisMarks` instance was used to plot dashed grid lines every 0.2 , skipping the values without a fractional part.

At the end of the recipe, we used the `chartPlotStyle(content:)` modifier on the `Chart` view, changing the background of the plot area and drawing a border around it.

See also

Apple's document on encoding and decoding custom types, implementing the `Encodable` and `Decodable` protocols: https://developer.apple.com/documentation/foundation/archives_and_serialization/encoding_and_decoding_custom_types.

Different types of charts: marks and mark configuration

If we must choose a building block to showcase the power and simplicity of Swift Charts, we most probably will choose marks. In this recipe, we are going to learn how to use different types of marks to plot different charts. We will be using the same dataset for all the charts, so we can compare how the same data is presented in different ways.

In this recipe, we will be building upon the recipe *Customizing charts: axes, annotations, and rules*. The best learning experience is to work on the previous recipe first, and then come back to this recipe. But for those who choose to work on this recipe, we have included references to the previous recipe throughout the steps.

Getting ready

Create a new SwiftUI iOS app named `WorkingWithMarks`.

How to do it...

Our app will display climate data for three cities in the USA for each month of the year 2022. The main screen will have a segmented picker to choose from two different datasets, rainfall data and temperature data. We will also have two more controls, one picker to choose the mark used to plot the data and a toggle to choose between US units or international units. The dataset, the toggle to choose the units, the unit conversions, and how the UI changes when we choose different units were explained in detail in the recipe *Customizing charts: axes, annotations, and rules*. Please refer to that recipe to understand these topics in detail. Let's start working on the data first, and then switch to the views. These are the steps:

1. We will be using the same data as used in the previous recipe. Drag and drop the `climate_data.json` file from the book's Resources folder at GitHub (<https://github.com/PacktPublishing/SwiftUI-Cookbook-3rd-Edition/tree/main/Resources/Chapter15/Recipe02>) into the project's main folder. Make sure the file is added to the target.
2. From the recipe *Customizing charts: axes, annotations, and rules*, copy the file `CityClimate.swift` into the project's main folder. Make sure the file is added to the target. The file includes two custom data types with a few extensions. The first extension for the struct `CityClimate` includes two computed variables to calculate values that will be used in the charts. We will add five more computed variables to calculate different magnitudes. After adding the five computed variables, the code for the extension should look like the following:

```
extension CityClimate {  
    var year: Int {  
        Calendar.current.dateComponents([.year], from: climate[0].date).  
        year!  
    }  
    var averageRainfall: Measurement<UnitLength> {  
        let initial = Measurement(value: 0, unit: UnitLength.meters)  
        let average = climate.map {  
            $0.rainfall.converted(to: .meters)  
        }.reduce(initial, +) / Double(climate.count)  
        return average  
    }  
    var averageLow: Measurement<UnitTemperature> {  
        let initial = Measurement(value: 0, unit: UnitTemperature.  
        celsius)  
        let average = climate.map {  
            $0.minTemperature.converted(to: .celsius)  
        }.reduce(initial, +) / Double(climate.count)  
    }  
}
```

```
        return average
    }

    var averageHigh: Measurement<UnitTemperature> {
        let initial = Measurement(value: 0, unit: UnitTemperature.
celsius)
        let average = climate.map {
            $0.maxTemperature.converted(to: .celsius)
        }.reduce(initial, +) / Double(climate.count)
        return average
    }

    var highestRainfallMonth: ClimateData {
        let initial = climate[0]
        let month = climate.reduce(initial) { partial, result in
            partial.rainfall > result.rainfall ? partial : result
        }
        return month
    }

    var highestTemperatureMonth: ClimateData {
        let initial = climate[0]
        let month = climate.reduce(initial) { partial, result in
            partial.maxTemperature > result.maxTemperature ? partial :
result
        }
        return month
    }

    var lowestTemperatureMonth: ClimateData {
        let initial = climate[0]
        let month = climate.reduce(initial) { partial, result in
            partial.minTemperature <= result.minTemperature ? partial :
result
        }
        return month
    }
}
```

3. Once we have completed our data layer, it is time to move to the view layer. Let's go to `ContentView.swift` and declare a few variables at the top: one to store our sample data and then three more state variables to drive the views. The first one, which will be used in a `Toggle` control, indicates a selection between US measurement units and international units. The second one will be used to select if we want to display the rainfall or the temperature charts. And the third one will be used to disable the control to select measurement units. A couple of our charts will not have units; we will disable the unit selection control for these charts.

The code should look like this:

```
struct ContentView: View {
    private var climateData: [CityClimate] = CityClimate.sample
    @State private var isUS = true
    @State private var isRainfallChart = true
    @State private var isToggleDisabled = false

    var body: some View {
        // ...
    }
}
```

4. Now, let's work on the `ContentView` view. We will use a `VStack` and embed a `Picker` at the top, then a conditional view depending on the value of the picker in the middle, and a `Toggle` view to switch between US and international units at the bottom. The conditional view will display a chart for the rainfall and a different chart for the temperatures. We will use custom views for the charts, so for now, we will use some dummy views. Replace the content of the `body` property with the following:

```
var body: some View {
    VStack {
        Picker("Type of chart", selection: $isRainfallChart) {
            Text("Rainfall").tag(true)
            Text("Temperatures").tag(false)
        }
        .pickerStyle(.segmented)
        .padding(.bottom)
        if isRainfallChart {
            Color.cyan
        } else {
            Rectangle().foregroundStyle(Gradient(colors: [.orange,
                .blue]))
        }
        HStack {
            Text("US")
            Toggle("", isOn: $isUS)
                .labelsHidden()
                .disabled(isToggleDisabled)
            Text("US")
        }
        .font(.largeTitle)
    }
}
```

```
    .padding()  
}
```

5. Use the live preview on the canvas to interact with the app. The preview should look like this:



Figure 15.10: Preview with the two different chart choices with dummy views

6. We are going to work on the rainfall charts. To keep our code more organized, we will use a separate file. Go ahead and create a new SwiftUI file named `RainfallView.swift`. Switch to this newly created file, and at the top, import the Charts framework and create three variables that will be initialized outside the view:

```
import Charts  
struct RainfallView: View {  
    var isUS: Bool  
    var climateData: [CityClimate]  
    @Binding var isToggleDisabled: Bool  
  
    var body: some View {  
        // ...
```

```
    }
}
```

7. Adjust the preview to remove the errors in Xcode:

```
#Preview {
    RainfallView(isUS: true, climateData: CityClimate.sample,
    isToggleDisabled: .constant(true))
        .padding()
}
```

8. Our view will have a `VStack` with a `Text` view, a `Chart` view, and an `HStack` view. Replace the content of the `body` variable with the following:

```
var body: some View {
    VStack {
        let rainfallUnit: UnitLength = isUS ? .inches : .millimeters
        Text("Average Rainfall (\(rainfallUnit.symbol))")
            .font(.title)
        Chart {
            // more code will follow
        }
        HStack {
            Text("Type of chart:")
            // more code will follow
        }
    }
}
```

9. The `HStack` will include a `Picker` to choose among the different types of charts, and its selection will be based on an `enum` variable. Before we work on the `Picker`, let's define a new `enum`. At the bottom of the file, outside the `RainfallView` struct, include the following extension:

```
extension RainfallView {
    enum ChartType: String, CaseIterable {
        case bar
        case line
        case points
        case area
        case areaNormalized = "area normalized"
    }
}
```

10. Now, at the top of the `RainfallView` struct, include the following `@State` variable to store the selection about the type of chart:

```
@State private var typeOfChart: ChartType = .bar
```

11. We are ready to work on the `Picker`, which will be in the `HStack` right below the `Text` view. After adding the code, it should look like the following:

```
HStack {  
    Text("Type of chart:")  
    Picker("Type of chart", selection: $typeOfChart) {  
        ForEach(ChartType.allCases, id: \.self) { chartType in  
            Text(chartType.rawValue).tag(chartType)  
        }  
    }  
.onChange(of: typeOfChart) { _, newValue in  
    isToggleDisabled = newValue == .areaNormalized  
}  
}
```

12. For our charts, we want to plot the data for the three cities and change the mark used depending on the user's selection. To iterate through our dataset, we will have two loops. The outer loop will iterate through the cities, and the inner loop will iterate through the monthly data about the climate. To select which type of mark to use to plot the chart, we will have a `switch` statement. Replace the `Chart` code with the following:

```
Chart(climateData, id:\.city) { cityClimate in  
    let city = cityClimate.city  
    ForEach(cityClimate.climate, id:\.date) { monthlyData in  
        let xValue: PlottableValue = .value("Month", monthlyData.date,  
unit: .month)  
        let rainfall = monthlyData.rainfall.converted(to: rainfallUnit).  
value  
        let yValue: PlottableValue = .value("Rain", rainfall)  
        switch typeOfChart {  
            // more code will follow  
            @unknown default:  
                BarMark(x: xValue, y: .value("Rain", 0))  
        }  
    }  
}
```

13. Use the live preview on the canvas to see what we have built so far. The preview should look like this:



Figure 15.11: Preview of the RainfallView with the chart selection picker

14. Now, we will implement our first chart, which is going to be the bar chart. We will add an annotation to the bar of the highest rainfall to display its value. Inside the `switch`, add the following case before the `@unknown default` case:

```
case .bar:  
    BarMark(x: xValue, y: yValue)  
        .foregroundStyle(by: .value("City", city))  
        .position(by: .value("City", city))  
        .annotation {  
            if monthlyData.date == cityClimate.highestRainfallMonth.date  
            {  
                let format = isUS ? "%.1f" : "%.0f"  
                Text(String(format: format, rainfall))  
            }  
        }  
}
```

```
.font(.caption2)
.padding(4)
.overlay {
    RoundedRectangle(cornerRadius: 4)
        .stroke(.black, lineWidth: 1)
}
}
```

15. Use the live preview on the canvas to preview the bar chart. The preview should look like this:

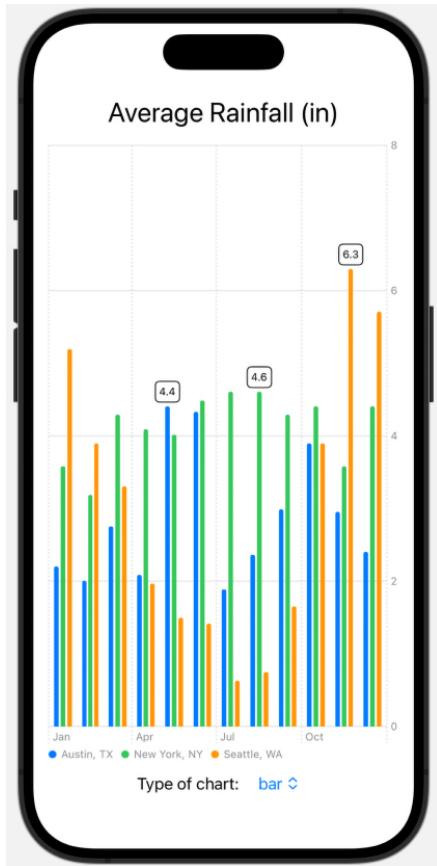


Figure 15.12: Preview of the bar chart with annotations

16. The second chart will be the line chart. We will use a `LineMark` to plot the values. Inside the `switch`, add the following `case` right before the `@unknown default` case:

```
case .line:
```

```
LineMark(x: xValue, y: yValue)
    .foregroundStyle(by: .value("City", city))
```

17. Use the live preview on the canvas to preview the line chart. The preview should look like this:

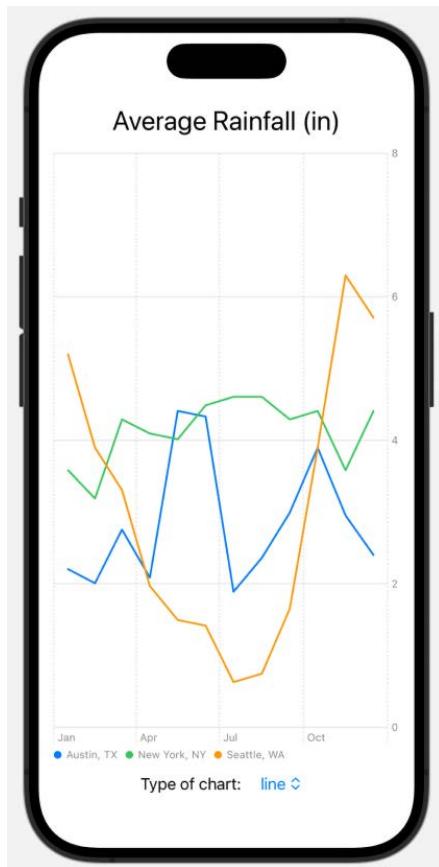


Figure 15.13: Preview of the bar chart

18. For our third chart, the point chart, we will use a `PointMark`. To make this chart more appealing, we will connect the points with a dotted line with a smooth interpolation, using a `LineMark`. Inside the `switch`, add the following `case` before the `@unknown default` case:

```
case .points:
    PointMark(x: xValue, y: yValue)
        .foregroundStyle(by: .value("City", city))
        .symbol(by: .value("City", city))
    LineMark(x: xValue, y: yValue)
        .foregroundStyle(by: .value("City", city))
```

```
.interpolationMethod(.catmullRom)  
.lineStyle(StrokeStyle(dash: [2]))
```

19. Use the live preview on the canvas to preview the points chart. The preview should look like this:

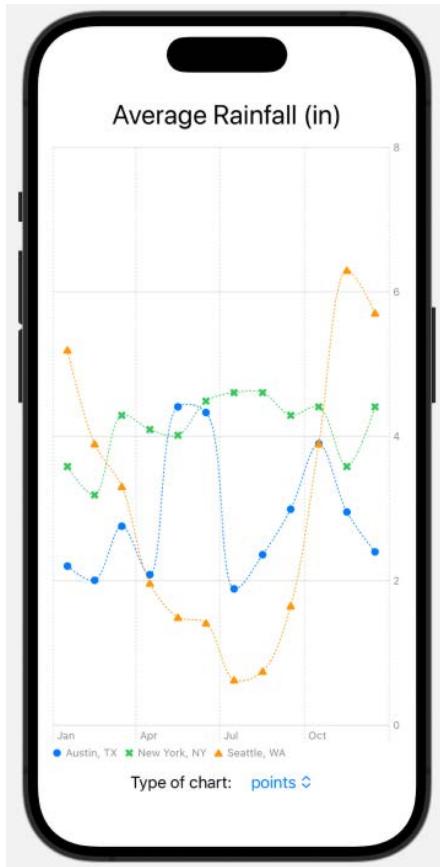


Figure 15.14: Preview of the points chart

20. The fourth chart will be an area chart, and we will use an `AreaMark`. We will add an annotation to each area with the name of the city. Inside the switch, add the following case before the `@unknown default` case:

```
case .area:  
    AreaMark(x: xValue, y: yValue)  
        .foregroundStyle(by: .value("City", city))  
        .annotation(position: .trailing) {  
            Text(city)  
                .font(.caption)
```

```
    .background(Color.white.opacity(0.5))  
}
```

21. Use the live preview on the canvas to preview the area chart. The preview should look like this:

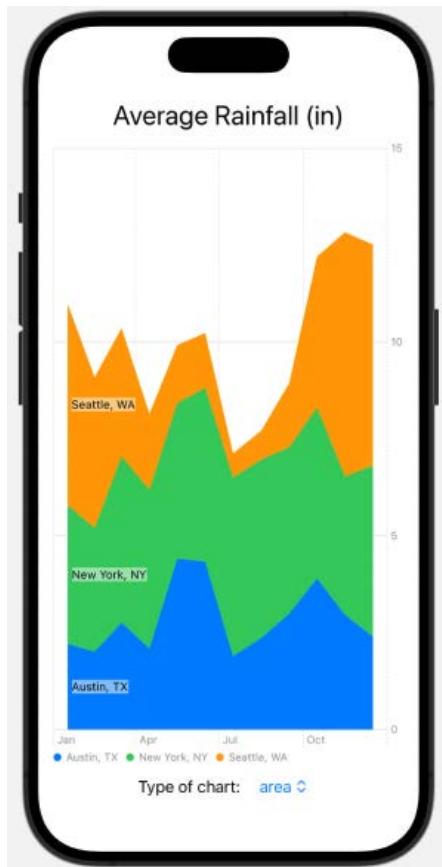


Figure 15.15: Preview of the area chart

22. Finally, our fifth chart will be a normalized area chart, where the stacked areas are normalized to 100% for each month, so we can compare our data percentage-wise. The `AreaMark` accepts an optional parameter, `stacking`, to account for this type of area chart. Inside the switch, add the following case before the `@unknown` default case:

```
case .areaNormalized:  
    AreaMark(x: xValue, y: yValue, stacking: .normalized)  
        .foregroundStyle(by: .value("City", city))  
        .annotation(position: .overlay) {  
            Text(city)
```

```
.font(.caption)
```

```
}
```

23. Use the live preview on the canvas to preview the normalized area chart. The preview should look like this:

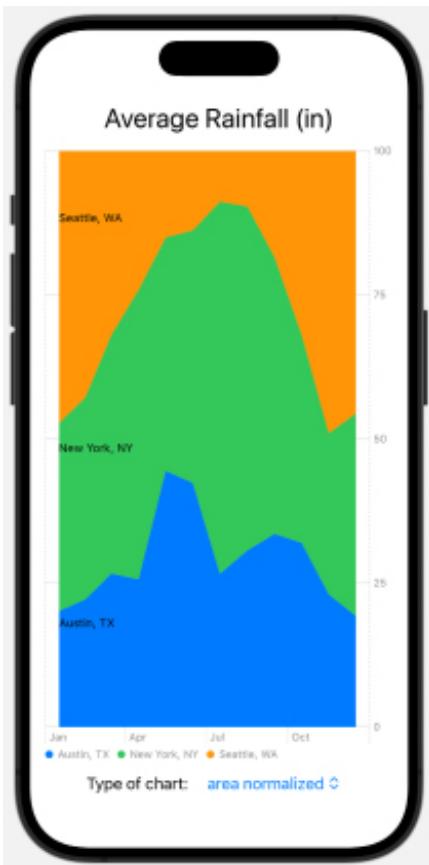


Figure 15.16: Preview of the normalized area chart

24. Now that we have finished our rainfall charts, if you wish, comment out the code for the case to remove the Xcode warning.
25. Switch to `ContentView.swift`, and modify the conditional view code to include our finished rainfall charts:

```
if isRainfallChart {  
    RainfallView(isUS: isUS, climateData: climateData, isToggleDisabled:  
    $isToggleDisabled)
```

```

    } else {
        Rectangle().foregroundStyle(Gradient(colors: [.orange, .blue]))
    }
}

```

26. Use the live preview on the canvas to preview the app so far for the rainfall charts. Interact with the unit selection, and notice how when we select the normalized area chart, the unit selection is disabled. The result should be similar to the following screenshots:

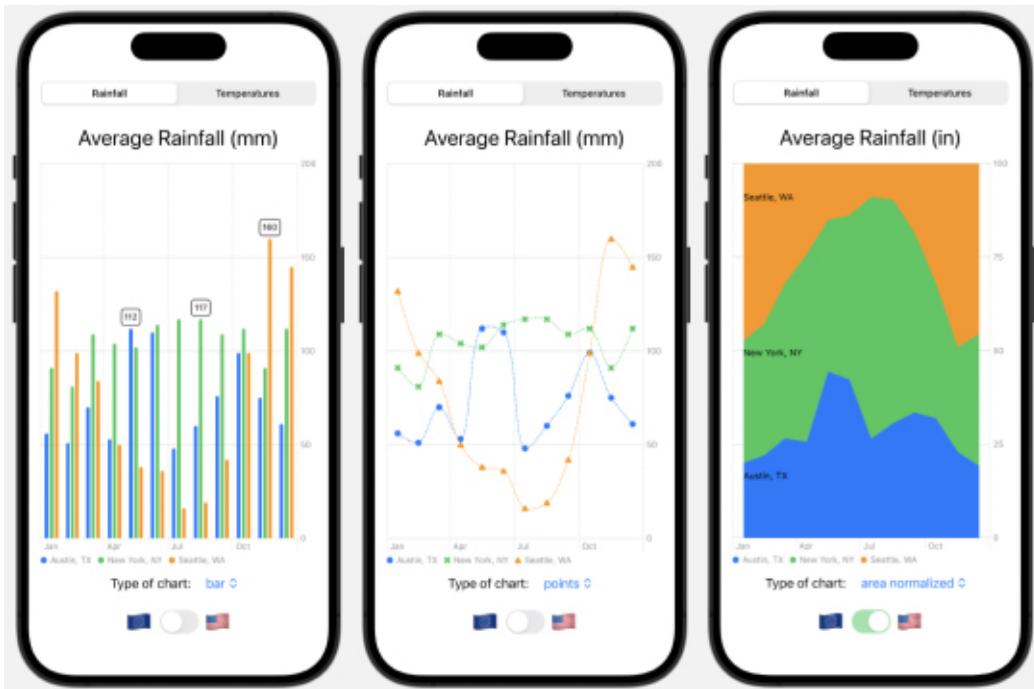


Figure 15.17: Preview of the finished rainfall charts

27. It is time to move to the temperature charts. Go ahead and create a new SwiftUI file named `TemperatureView.swift`. The code for this view is very similar to the one we created for the rainfall charts, so we won't break down the steps, except for the actual chart code. Replace the contents of the file with the following:

```

import SwiftUI
import Charts

struct TemperatureView: View {
    var isUS: Bool
    var climateData: [CityClimate]
    @Binding var isToggleDisabled: Bool
    @State private var typeOfChart: ChartType = .bar
    @State private var cityIndex: Int = 0

```

```
var body: some View {
    VStack {
        let temperatureUnit: UnitTemperature = isUS ? .fahrenheit :
            .celsius
        Text("Average Temperatures (\(temperatureUnit.symbol))")
            .font(.title)
        Chart {
            let selectedCity = climateData[cityIndex]
            ForEach(selectedCity.climate, id:\.date) { monthlyData in
                // chart code will go here
            }
        }
        .chartYScale(domain: isUS ? 25...100 : -5...45)
        Picker("City", selection: $cityIndex) {
            ForEach(0..<climateData.count, id: \.self) { index in
                Text(climateData[index].city).tag(index)
            }
        }
        .pickerStyle(.segmented)
        HStack {
            Text("Type of chart:")
            Picker("Type of chart", selection: $typeOfChart) {
                ForEach(ChartType.allCases, id: \.self) { chartType in
                    Text(chartType.rawValue).tag(chartType)
                }
            }
        }
        .onAppear {
            isToggleDisabled = false
        }
    }
}

extension TemperatureView {
    enum ChartType: String, CaseIterable {
        case bar
        case area
    }
}
```

```
#Preview {  
    TemperatureView(isUS: true, climateData: CityClimate.sample,  
    isToggleDisabled: .constant(true))  
    .padding()  
}
```

28. Use the live preview on the canvas to see what we have built so far. The preview should look like this:



Figure 15.18: Preview of the TemperatureView with the city and chart selection pickers

29. For our charts, we will use the dates for the x-axis and the temperatures for the y-axis. However, for each date, we have two temperature values, so instead of plotting the individual values, we will plot the range of values, from the low temperature to the high temperature. For this type of chart, called *range chart* or *interval chart*, Swift Charts provides three options, which are bars, areas, and rectangles. Furthermore, to declutter the chart, we will only plot one city at a time, and the user will have to switch cities with the picker. To iterate through our dataset, we will choose the data for the city chosen by the user, and then iterate through the monthly climate data. To select which type of mark to use to plot the chart, we will have a `switch` statement. Replace the `Chart` code with the following:

```
Chart {
    let selectedCity = climateData[cityIndex]
    ForEach(selectedCity.climate, id:\.date) { monthlyData in
        let date = monthlyData.date
        let xValue: PlottableValue = .value("Month", date, unit: .month)
        let low = monthlyData.minTemperature.converted(to:
            temperatureUnit).value
        let lowValue: PlottableValue = .value("Temperature", low)
        let high = monthlyData.maxTemperature.converted(to:
            temperatureUnit).value
        let highValue: PlottableValue = .value("Temperature", high)
        let gradient = Gradient(colors: [.orange, .cyan])
        switch typeOfChart {
            case .bar:
                BarMark(x: xValue, yStart: lowValue, yEnd: highValue)
                    .foregroundStyle(gradient)
                    .cornerRadius(5)
            case .area:
                AreaMark(x: xValue, yStart: lowValue, yEnd: highValue)
                    .interpolationMethod(.cardinal)
                    .foregroundStyle(gradient)
        }
    }
}.chartYScale(domain: isUS ? 25...100 : -5...45)
```

30. Use the live preview on the canvas to preview our work so far. Interact with the city selection and the chart selection. The result should be similar to the following screenshots:

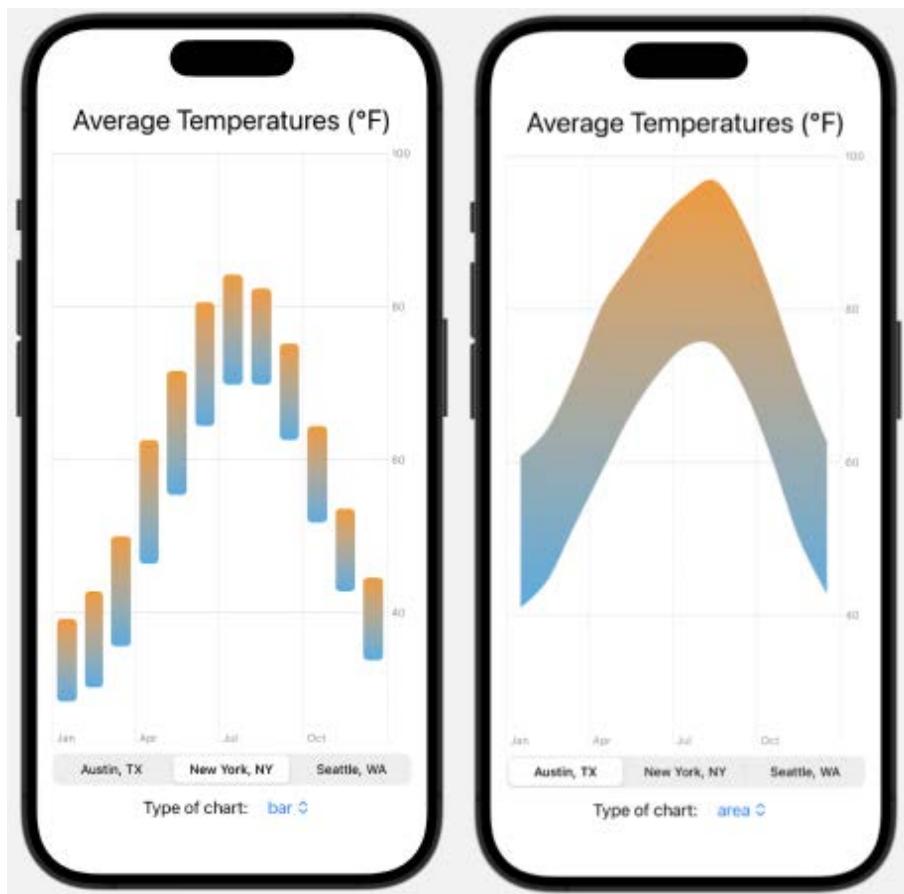


Figure 15.19: Preview of our initial temperature range charts

31. Let's customize our bar chart with some annotations. First, create a new `AnnotationText` custom view inside the extension where we created the enum `ChartType`. The code should look like this:

```
extension TemperatureView {  
    enum ChartType: String, CaseIterable {  
        case bar  
        case area  
    }  
  
    struct AnnotationText: View {  
        var text: String  
        var color: Color  
    }  
}
```

```
var body: some View {
    Text(text)
        .font(.caption2)
        .foregroundStyle(color)
        .padding(.horizontal, 4)
        .background(.white.opacity(0.7))
    }
init(_ text: String, color: Color = .primary) {
    self.text = text
    self.color = color
}
}
```

32. We would like to display the values of the average high and average low temperatures as annotations. Since these annotations will affect the whole chart, they will be placed inside the Chart body but outside the `ForEach`. Add the following code right after the `ForEach` closing curly bracket:

```
ForEach(selectedCity.climate, id:\.date) { monthlyData in
    // ...
}
let averageLow = selectedCity.averageLow.converted(to: temperatureUnit).value
let averageHigh = selectedCity.averageHigh.converted(to: temperatureUnit).value
switch typeOfChart {
    case .bar:
        // more code will follow
    case .area:
        // more code will follow
}
```

33. For the bar chart, we will plot a `RectangleMark` delimited by the average high and average low temperatures. Add this code for the `.bar` case:

```
case .bar:
    let firstDay = DateComponents(year: selectedCity.year, month: 1, day:1)
    let firstDate = Calendar.current.date(from: firstDay)!
    let lastDay = DateComponents(year: selectedCity.year, month: 12, day:31)
    let lastDate = Calendar.current.date(from: lastDay)!
    let color = Color.black.opacity(0.5)
```

```
RectangleMark(xStart: .value("First month", firstDate),
               xEnd: .value("Last month", lastDate),
               yStart: .value("Average low", averageLow),
               yEnd: .value("Average high", averageHigh))
.foregroundStyle(.gray.opacity(0.1))
.annotation {
    AnnotationText("Average High", color: color)
}
.annotation(position: .bottom) {
    AnnotationText("Average Low", color: color)
}
```

34. For the area chart, we will plot two RuleMark instances, one for the average high temperature and another one for the average low temperature. Add this code for the .area case:

```
case .area:
    RuleMark(y: .value("Average high", averageHigh))
        .foregroundStyle(.red)
        .lineStyle(.init(dash: [4]))
    .annotation {
        AnnotationText(
            "Average High = \(String(format: "%.1f", averageHigh))",
            color: .red
        )
    }
    RuleMark(y: .value("Average low", averageLow))
        .foregroundStyle(.blue)
        .lineStyle(.init(dash: [4]))
    .annotation(position: .bottom) {
        AnnotationText(
            "Average Low = \(String(format: "%.1f", averageLow))",
            color: .blue
        )
    }
}
```

35. Use the live preview on the canvas to preview our finished charts. The result should be similar to the following screenshots:

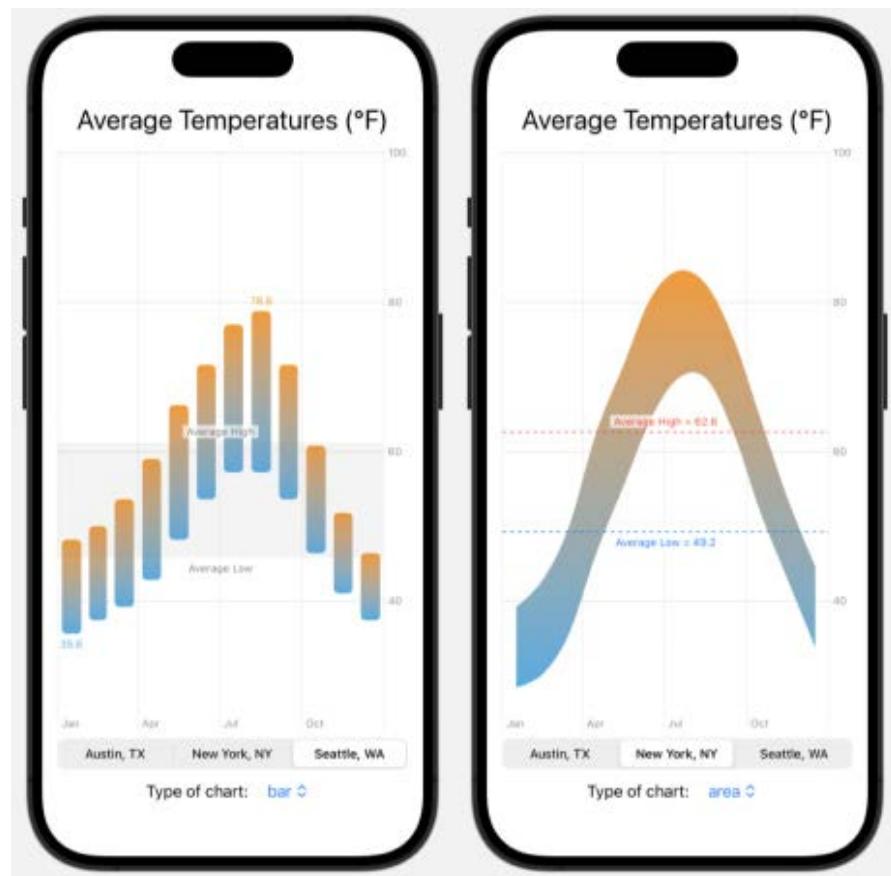


Figure 15.20: Preview of our final temperature range charts

36. Finally, include our finalized view in its parent view. Switch to the `ContentView` view and replace the code for the dummy view. The finished code should be this:

```
if isRainfallChart {  
    RainfallView(isUS: isUS, climateData: climateData, isToggleDisabled:  
    $isToggleDisabled)  
} else {  
    TemperatureView(isUS: isUS, climateData: climateData,  
    isToggleDisabled: $isToggleDisabled)  
}
```

37. Use the live preview on the canvas to interact with our finished app. The result should be similar to the following screenshots:

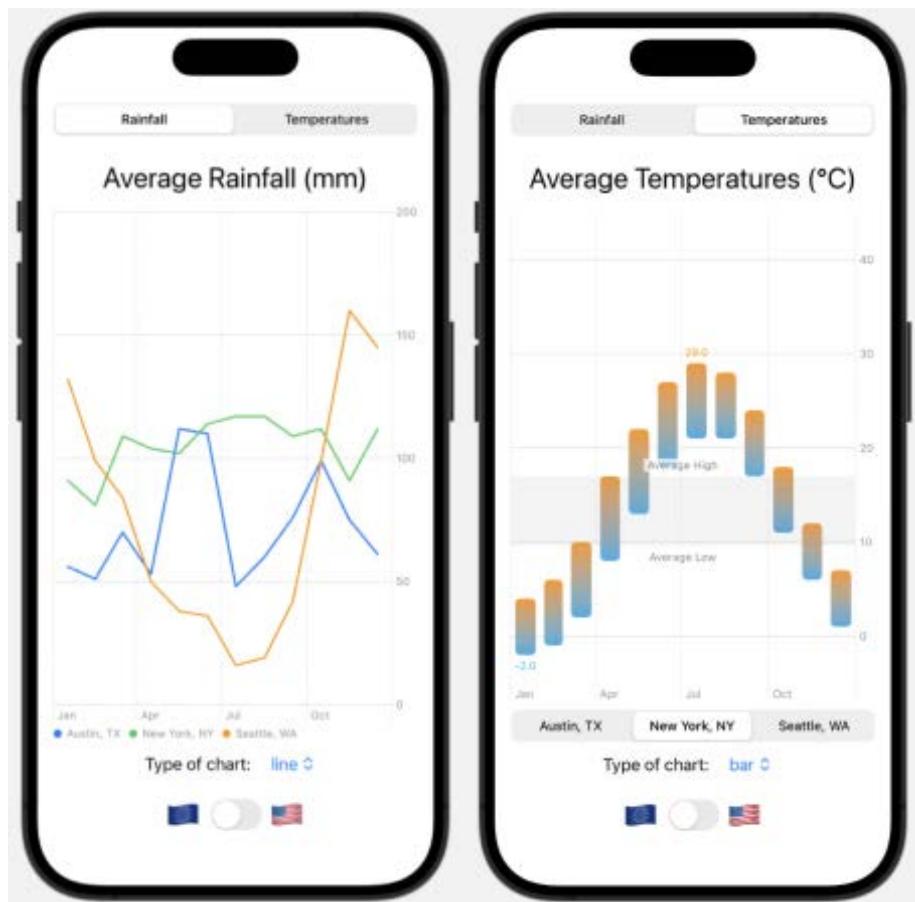


Figure 15.21: Preview of the final app

How it works...

Remember the two-step process when using charts for data visualization: prepare the data so that it can be easily used in a chart, and then use Swift Charts to accomplish the desired visualization.

At the start of the recipe, we worked on our data layer. Since we used the code from the recipe *Customizing charts: axes, annotations, and rules*, I suggest you check out that recipe in order to understand the data layer better. We just added a few computed variables to `CityClimate`, which were used in our views.

Our `ContentView` layout was achieved with a `VStack`, with a `Picker` at the top, a conditional view in the middle, and an `HStack` with two icons and a `Toggle` control at the bottom. This control was used in the previous recipe, so refer to that recipe for an explanation of how it works. The conditional view, an `if` statement with an `else` clause, included two custom views, `RainfallView` for the rainfall data and `TemperatureView` for temperature data.

For `RainfallView`, we chose to plot the rainfall data for the three cities side by side with different types of charts. We used the familiar layout of a `VStack` with a `Text` view at the top for the title, a `Chart` view in the middle, and a `Picker` at the bottom, choosing among the five different types of charts. For the chart, we used a nested loop, that is, one loop inside another loop, to iterate over the data. The outer loop iterated over the cities in our array, and we used the `Chart` initializer to do it:

```
Chart(climateData, id:\.city) { cityClimate in ... }
```

For the inner loop, we used a `ForEach` views to iterate over the set of climate data for the selected city:

```
ForEach(cityClimate.climate, id:\.date) { monthlyData in ... }
```

Inside the nested loop, we used a `switch` statement to choose which type of mark to use for the chart, depending on the user's selection. For the bar charts, we used `BarMark` to plot the data for the three cities side by side. This type of bar chart was explained in detail in the recipe *Swift Charts basics*; please refer to that recipe for more details. We used an annotation with a conditional to display a custom text, inside a rounded rectangle, at the top of the bar with the highest rainfall:

```
.annotation {
    if monthlyData.date == cityClimate.highestRainfallMonth.date { ... }
}
```

Since we iterated over 12 dates, one for each month of the year, when the above condition was true, it meant that the `BarMark` we were plotting was the one with the highest rainfall. Note how useful the computed variable `highestRainfallMonth` was, which we created in our data layer!

We continued working on different types of charts, and for the line chart, we used a plain `LineMark` with a `foregroundStyle(by:)` modifier.

For the point chart, we used a `PointMark` with two modifiers, `foregroundStyle(by:)` and `symbol(by:)`. To improve the presentation, we added a dotted line, with a `LineMark`, which smoothly connected all the points thanks to an interpolation function, made possible with the following modifier:

```
.interpolationMethod(.catmullRom)
```

We implemented two types of area charts using `AreaMark`, the stacked-standard chart and the stacked-normalized chart. This was very easy because the initializer of `AreaMark` provided a parameter:

```
AreaMark(x: xValue, y: yValue, stacking: .normalized)
```

For `TemperatureView`, because we had two values for each month, the low and high temperature, we decided to plot range charts, for one city at a time. We used the familiar layout of a `VStack` with a `Text` view at the top for the title, a `Chart` view in the middle, and two `Picker` views at the bottom, the first with a segmented layout, to choose the city, and the second to choose from the two different types of range charts.

Three of the marks of Swift Charts support range charts thanks to specific initializers: `BarMark`, `AreaMark`, and `RectangleMark`.

We used `BarMark` to plot the temperatures for each month, having a bar that spanned from the low temperature to the high temperature:

```
BarMark(x: xValue, yStart: lowValue, yEnd: highValue)
    .foregroundStyle(gradient)
```

We made the bar more appealing by using a gradient style for the foreground:

```
let gradient = Gradient(colors: [.orange, .cyan])
```

The orange color is for higher temperatures, transitioning to cyan for lower temperatures.

In a similar way, we used `AreaMark` as an alternative option for the charts, with an interpolation curve to smooth the edges at the intersections:

```
AreaMark(x: xValue, yStart: lowValue, yEnd: highValue)
    .interpolationMethod(.cardinal)
    .foregroundStyle(gradient)
```

We enhanced the temperature charts with some annotations. Because the annotations affected the whole plotting area, we placed these inside the `Chart` view but outside the `ForEach` view.

For the bar chart, we used a `RectangleMark` as an annotation:

```
RectangleMark(xStart: .value("First month", firstDate),
              xEnd: .value("Last month", lastDate),
              yStart: .value("Average low", averageLow),
              yEnd: .value("Average high", averageHigh))
```

With `RectangleMark`, we easily plotted a rectangle using the units of the chart, dates for the x-axis and degrees of temperature for the y-axis. With `BarMark`, we could have plotted the same rectangle, but we would have been forced to use point calculations for the width or height.

For the area chart, we used two `RuleMark` instances, one for the low temperature and one for the high temperature.

For the text annotations used in the rules and the rectangle, we created a custom view `AnnotationText` so that we could reuse the formatting code everywhere.

See also

- To annotate a rectangular area: <https://developer.apple.com/documentation/charts/rectanglemark>
- For range bar charts, see interval bar charts: <https://developer.apple.com/documentation/charts/barmark>
- For stacked area charts and range area charts: <https://developer.apple.com/documentation/charts/areamark>

Histograms with data bins

In the three other recipes in this chapter, we used Swift Charts to create beautiful charts. We used and learned the different building blocks and view modifiers available. Please refer to the previous recipes if you're not familiar with Swift Charts.

In this recipe, we will focus on one specific data visualization chart called histogram. Histograms are used in statistics, and it is the most used chart to plot the frequency distribution of an event. A histogram is a very useful chart to understand a large dataset. The histogram looks like a bar chart, but here, the main difference is that each bar represents the frequency of a *range* of values, instead of the frequency of a single value. In the y-axis, we have the frequency of the event, and in the x-axis, the range of the event. For example, imagine we want to understand the frequency distribution of the age of a population of 1,000,000 people, measured in years. We have an initial dataset of 1,000,000 values, with most of the ages being repeated. We count how many times each age appears in the dataset to obtain the frequency distribution dataset, which is going to have less than 1,000,000 values due to the repeated ages. To plot a distribution bar chart, we would plot the age in the x-axis, from the youngest to the oldest age recorded, and the number of times that the specific age appeared in the original dataset in the y-axis. Let's say we have 120 different ages; our bar chart would have 120 bars, and the height of the bar would be the number of times that specific age appeared. But what if we wanted to group the ages in buckets? This will help us reduce the number of bars in the chart and simplify it. That is precisely what a histogram does. Following our example, we could group the ages in groups of 10 years, having 12 bars instead of 120 bars. The first bar would be the count of the people between 0 to 9 years old, the second one people from 10 to 19 years old, and so forth.

Getting ready

Create a new SwiftUI iOS app named `Histograms`.

How to do it...

Our app will display the frequency distribution of the score obtained when we roll several dice. For each roll of the dice, we will have a result, and we will plot a histogram chart of the distribution of the results. We will use a random number generator to simulate the roll of the dice. This will allow us to create some code to generate the dataset, rather than importing the data from a JSON file. Let's start working on the data first, and then switch to the views. These are the steps:

1. Since this is going to be a simple app, we will only use the file `ContentView.swift` for our code. We will include our data layer and view layer in the same file. Go to the top of the file and create the following struct:

```
struct HistogramBin {  
    let index: Int  
    let range: ChartBinRange<Int>  
    let frequency: Int  
}
```

```
struct Histogram {  
    let bins: [HistogramBin]  
    private let xMin: Int  
    private let xMax: Int  
    var xDomain: [Int] { [xMin, xMax] }  
    private(set) var maxY: Int  
}
```

2. For the code to simulate the roll of several dice, we will use a random number generator. Since random number generation is commonly used in games, the `GameplayKit` framework provides this feature. At the top of the file, add another `import` statement:

```
import GameplayKit
```

3. Now, let's create an extension on `Histogram` with a `static` function to generate the data for the histogram. The code will be explained in detail in the section *How it works...*, so for now, just include this code after the two struct declarations:

```
extension Histogram {  
    static func generate(  
        lowestValue: Int,  
        highestValue: Int,  
        numberoftimes: Int = 10000,  
        numberofbins: Int = 25  
    ) -> Histogram {  
        let randomSource = GKRandomSource()  
        let generator = GKGaussianDistribution(  
            randomSource: randomSource,  
            lowestValue: lowestValue,  
            highestValue: highestValue  
        )  
        let dataSet: [Int] = (0..nextInt() }  
        let minimumStride = (highestValue - lowestValue) / numberofbins  
        let bins = NumberBins(  
            data: dataSet,  
            desiredCount: numberofbins,  
            minimumStride: minimumStride  
        )  
        let groupedData = Dictionary(grouping: dataSet) { bins.index(for:  
$0) }  
        let data: [HistogramBin] = groupedData.map { key, values in
```

```
        .init(index: key, range: bins[key], frequency: values.  
count)  
    }  
    let padding = (highestValue - lowestValue) / 10  
    let maxY = data.reduce(data[0].frequency) { maxFrequency, bin in  
        maxFrequency > bin.frequency ? maxFrequency : bin.frequency  
    }  
    return Histogram(  
        bins: data,  
        xMin: lowestValue - padding,  
        xMax: highestValue + padding,  
        maxY: maxY  
    )  
}  
}  
}
```

4. Once we have our data ready, it is time to switch to the `ContentView` view. To start, declare our only `@State` variable at the top, `histogram`, which will include our dataset. Add a `private` function at the bottom to generate the data and store it in the `histogram` variable. The code should be this:

```
struct ContentView: View {  
    @State private var histogram: Histogram?  
    var body: some View {  
        // ...  
    }  
  
    private func generateHistogram() -> Histogram {  
        let number_of_dice = 25  
        let lowestValue = number_of_dice  
        let highestValue = number_of_dice * 6  
        return .generate(lowestValue: lowestValue, highestValue:  
            highestValue)  
    }  
}
```

5. Once we have the data, replace the content of the `body` variable with the following code:

```
var body: some View {  
    if let histogram {  
        VStack {  
            // ...  
        }  
    }  
}
```

```

        .padding()
    } else {
        ProgressView()
            .onAppear {
                histogram = generateHistogram()
            }
    }
}

```

6. Time to work on the layout for our main view. We will have a `VStack` with the `Text` view at the top with the title, the `Chart` view in the middle, and a `Button` at the bottom. With a tap of the button, a new dataset will be created, and the view will be generated again. Replace the `VStack` with the following code:

```

VStack {
    Text("Histogram of our data")
        .font(.title)
    Chart { }
    Button("Get new data") {
        withAnimation {
            self.histogram = generateHistogram()
        }
    }
}
.padding()

```

7. It's time to finally work on our chart. A histogram is a bar chart that uses a special data structure for the x-axis, the `ChartBinRange` struct, used to represent data bins for a histogram. Replace the `Chart` view code with the following:

```

Chart(histogram.bins, id: \.index) { bin in
    BarMark(
        x: .value("Range", bin.range),
        y: .value("Frequency", bin.frequency)
    )
}
.chartYScale(domain: [0, max(1000, histogram.maxY)])
.chartXScale(domain: histogram.xDomain)
.chartXAxis {
    AxisMarks(values: .automatic(desiredCount: 5))
}
.aspectRatio(contentMode: .fit)

```

8. Use the live preview of the canvas to interact with the app. Use the button to generate the data again and again. If everything went well, you should see a preview like the following screenshot:

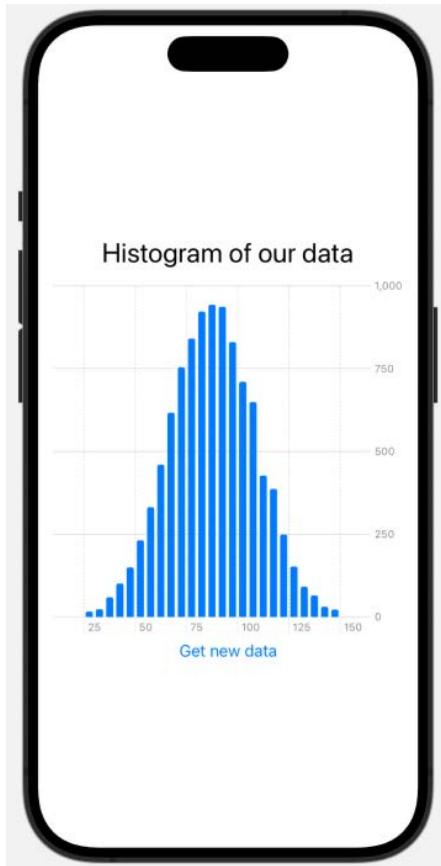


Figure 15.22: Preview of the final app with the histogram chart

How it works...

At the start of the recipe, we prepared our data so that it could be easily used in our histogram chart. Since the purpose of our recipe was to plot a histogram chart, we created our own code to generate a large dataset.

We started by modeling our data with two structs. For the data bins of our histogram, we declared the struct `HistogramBin`. We needed two properties for the data bin, a range of values and the frequency. We used a `ChartBinRange` for the range of values. This data type is provided by Swift Charts for histograms, and it uses an associated type conforming to the `Comparable` protocol, like `Int`, `Double`, or `String`.

Since our data represents the score of a roll of dice, we chose `Int` for the associated type. And for the frequency, we chose `Int` too, because we represented the number of times a specific score occurred. We added a third property, an `index` of type `Int`, which we used to iterate through the bins, in the `Chart` view initializer.

For our data container, we created the `struct Histogram`, which is basically an array of `HistogramBin` with a couple of computed variables, used to configure the scales of the two axes of the chart.

We created an extension of `Histogram` with a `static` function to generate an array of random numbers that follow a *Gaussian distribution*, also known as a *normal distribution*. The sum of the results of a roll of dice follows this type of distribution. The random numbers generated by our function were distributed in data bins.

Let's break down our function. Its parameters are self-explanatory. The function generates `numberOfTimes` integer random numbers (by default 10,000), following a normal distribution, from `lowestValue` to `highestValue`, placing the frequency distribution results in a `Histogram` instance with `numberOfBins` number of data bins (by default, 25):

```
static func generate(lowestValue: Int, highestValue: Int,  
    numberOfTimes: Int = 10000, numberOfBins: Int = 25) -> Histogram
```

The first three lines of code of the function stored the random numbers in the local `dataSet` variable:

```
let dataSet: [Int] = (0..
```

To distribute the numbers in different data bins, we used the `NumberBins` struct, which is a part of Swift Charts, as a helper type:

```
let minimumStride = (highestValue - lowestValue) / numberOfBins  
let bins = NumberBins(data: dataSet, desiredCount: numberOfBins,  
    minimumStride: minimumStride)
```

Then, we created a `Dictionary`, where the keys were numbers representing the `index` of one bin, and the values were an array of integer numbers that belong to the specific data bin:

```
let groupedData = Dictionary(grouping: dataSet) { bins.index(for: $0) }
```

Then, finally, we mapped our `Dictionary` to an array of `HistogramBin` instances:

```
let data: [HistogramBin] = groupedData.map { key, values in  
    .init(index: key, range: bins[key], frequency: values.count)  
}
```

Notice how `groupedData` contains the 10,000 random numbers distributed in several data bins, but `data` only contains the frequency for each bin, which makes the dataset more manageable and ready to be used in our histogram chart.

We finished the `generate` function by calculating the range for the x-axis, with a 10% padding on each side:

```
let padding = (highestValue - lowestValue) / 10
```

And the maximum value of all the frequencies:

```
let maxY = data.reduce(data[0].frequency) { maxFrequency, bin in
    maxFrequency > bin.frequency ? maxFrequency : bin.frequency
}
```

Finally, we returned an instance of our `Histogram` type:

```
return Histogram(bins: data, xMin: lowestValue - padding,
                 xMax: highestValue + padding, maxY: maxY)
```



If you need to work with histograms, it is very important that you understand how to prepare the data. We have prepared an Xcode playground file, `Histogram.playground`, which you can download from the book's Resources folder at GitHub (<https://github.com/PacktPublishing/SwiftUI-Cookbook-3rd-Edition/tree/main/Resources/Chapter15/Recipe04>). With this file, you will be able to see all the intermediate values used in the calculation and the generation of the histogram bins.

For our view, we used one `@State` `histogram` variable of type `Histogram` to store the histogram data. When the view appeared for the first time, the `histogram` variable was nil, and we used a conditional view to show a `ProgressView` instead of a chart. We added a `onAppear` modifier to call the private function `generateHistogram()` to generate the data for the roll of 25 dice. Once we generated the histogram data, we showed a `VStack` with a `Text` view for the title, our `Chart` view in the middle, and a `Button` at the bottom. Each tap of the button generated new histogram data, and the chart was redrawn with new values.

For the `Chart` view, we used `BarMark`, and we iterated through all the bins using the initializer:

```
Chart(histogram.bins, id: \.index) { bin in
```

We used the `range` property of our `bin` for the x-axis and the `frequency` property for the y-axis.

Since we knew our dataset very well, we customized the scale of both axes to enhance the presentation of the histogram:

```
.chartYScale(domain: [0, max(1000, histogram.maxY)])
.chartXScale(domain: histogram.xDomain)
```

We customized the labels for the x axes using an automatic setting, with a desired count of 5 axis labels:

```
AxisMarks(values: .automatic(desiredCount: 5))
```

There's more

Swift Charts also provides the DateBins structure to plot histograms, where we have ranges of dates instead of ranges of numbers.

See also

Random number generator: <https://developer.apple.com/documentation/gameplaykit/gkgaussiandistribution>

Pie charts and donut charts

When Apple released the Swift Charts framework in 2022, it didn't include any marks to plot pie charts. The following year, with the updates announced at WWDC 2023, a new mark was added, SectorMark, which allows us to plot pie charts and donut charts.

A pie chart is a normalized statistical graphic in the shape of a circle, which is used to show numerical proportions among several quantities. The circle is divided into different slices, where each slice shows the relative size of the quantity with respect to the total. The pie represents the total, and the slices represent each individual quantity. A donut chart is a pie chart with the area at the center of the circle removed.

In this recipe, we will create a simple app that will display a dataset using a pie chart and a donut chart.

Getting ready

Create a new SwiftUI iOS app named `PieAndDonutCharts`.

How to do it...

Our app will display a typical business chart of the sales of our product in four different cities. We will plot the data in a pie chart and a donut chart, and the user will have a picker to choose between the two charts. These are the steps:

1. Let's start working on the data first. Create a new Swift file, `CitySales.swift`. We will define an enumeration and two structures. Replace the content of the file with the following:

```
import Foundation

enum City: String {
    case austin = "Austin"
    case dallas = "Dallas"
    case houston = "Houston"
    case sanAntonio = "San Antonio"
}

struct CitySales {
```

```
    let city: City
    let sales: Decimal
}

struct TotalSales {
    let total: [CitySales]

    init(total: [CitySales]) {
        self.total = total.sorted { $0.sales > $1.sales }
    }
}
```

2. At the bottom of the file, add an extension to `TotalSales`, with a `static` variable to include some sample data for our chart. Since our data is of type `Decimal`, we use the `init(integerLiteral:)` initializer to create a `Decimal` instance from the provided `Int` value:

```
extension TotalSales {
    static var texas: TotalSales = .init(total: [
        .init(city: .austin, sales: .init(integerLiteral: 5321878)),
        .init(city: .dallas, sales: .init(integerLiteral: 18399650)),
        .init(city: .houston, sales: .init(integerLiteral: 35562804)),
        .init(city: .sanAntonio, sales: .init(integerLiteral: 10391429))
    ])
}
```

3. Let's move to `ContentView.swift` and work on our views. Declare a variable to store the sample data, and a `@State` variable to represent the type of chart selected by the user. Additionally, for our `City` enumeration, add conformance to the `Plottable` protocol:

```
struct ContentView: View {
    private var data: TotalSales = .texas
    @State private var isDonutChart = false
    var body: some View {
        // ...
    }
}

extension City: Plottable {}
```

4. For the layout of our view, we will use a `VStack`. Replace the contents of the `body` variable with this:

```
var body: some View {
    VStack {
```

```
Picker("", selection: $isDonutChart) {
    Text("Pie Chart").tag(false)
    Text("Donut Chart").tag(true)
}
.pickerStyle(.segmented)
.padding(.bottom)
Text("Sales by City")
    .font(.title)
Chart {
}
.aspectRatio(contentMode: .fit)
}
.padding()
}
```

5. It's time to work on the chart. We will use the `Chart` initializer to iterate over our dataset, and `SectorMark` to plot the sales quantities. For the donut chart, we will indicate that the inner radius is 45% of the total radius. Replace the code for the `Chart` view with the following:

```
Chart(data.total, id:\.city) { citySales in
    SectorMark(angle: .value("Sales", citySales.sales),
                innerRadius: isDonutChart ? .ratio(0.45) : 0,
                angularInset: 1)
    .foregroundStyle(by: .value("City", citySales.city))
}
.aspectRatio(contentMode: .fit)
```

6. With just a few lines of code, and the power of Swift Charts, we have functional circular charts. Use the live preview on the canvas, interact with the picker, and observe what we have achieved so far. It should look like the following screenshots:



Figure 15.23: Preview of the basic charts

7. Let's customize our charts. We will use an annotation on each slice. Add the following view modifier to the SectorMark:

```
.annotation(position: .overlay) {  
    VStack {  
        if isDonutChart {  
            Text(citySales.city.rawValue)  
        }  
    }  
}
```

```
        Text(  
            citySales.sales,  
            format: Decimal.FormatStyle.Currency(code: "USD").scale(1e-6)  
        ) + Text("M")  
    }  
.font(.caption2)  
.foregroundStyle(.black)  
.padding(4)  
.background(.white.opacity(0.7))  
.cornerRadius(4)  
}
```

8. To finish our customizations, we will change the colors of the slices and hide the legend for the donut chart. Add these two view modifiers to the chart. The code should be this:

```
Chart(data.total, id:\.city) { citySales in  
    // ...  
}  
.aspectRatio(contentMode: .fit)  
.chartForegroundStyleScale(  
[  
    City.houston: .red,  
    .dallas: .yellow,  
    .sanAntonio: .brown,  
    .austin: .cyan  
]  
)  
.chartLegend(isDonutChart ? .hidden : .visible)
```

9. Use the live preview to test our app. The result should be like the following screenshots:



Figure 15.24: Preview of the finalized charts

How it works...

At the start of the recipe, we prepared our data so that it could be easily used in our circular charts. We used a dataset representing hypothetical sales figures by city.

For a circular chart, we need our data to have two variables, one categorical variable and one numerical variable. In our app, the categorical variable was the name of the city, and the numerical variable was the sales figure. The circular charts plotted the data graphically so that each slice represented the sales figure associated with a city.

We started by modeling our data with one enumeration and two structures. We declared the enum `City`, which stores `String` raw values to model our cities. We modeled our sales with the struct `CitySales` with two properties, `city` of type `City` and `sales` of type `Decimal`. We used the `Decimal` type because it guarantees accurate precision for financial quantities. Finally, we declared the struct `TotalSales` as our type to hold the data with only one property, named `total`, of the array type `CitySales`. We included an initializer to sort the data by sales quantity:

```
init(total: [CitySales]) {
    self.total = total.sorted { $0.sales > $1.sales }
}
```

Thanks to having the data ordered by sales, our circular charts plotted the data following that order, starting with the biggest slice and ending with the smallest slice.

For our root view, we used the traditional vertical layout with a `VStack`, with a `Picker` at the top, to choose between a pie chart or a donut chart, a `Text` view for a title, and the `Chart` view.

For the charts, we iterated over our dataset using the chart initializer:

```
Chart(data.total, id:\.city) { citySales in
```

We used the `SectorMark` to plot our data, using the sales quantity for the angle of the sectors. We set an angular inset of 1 degree to have some separation between sectors, and we used the city name to drive the foreground style of the sectors. Depending on the user selection, we also set the inner radius to a 45% ratio for donut charts or 0 for pie charts:

```
SectorMark(angle: .value("Sales", citySales.sales),
           innerRadius: isDonutChart ? .ratio(0.45) : 0,
           angularInset: 1)
.foregroundStyle(by: .value("City", citySales.city))
```

We customized the pie chart with overlay annotations for the sales figure, and for the donut chart, we also included the name of the city. We used the `format` argument of the `Text` view to format our sales figure to show millions of dollars:

```
Text(
    citySales.sales,
    format: Decimal.FormatStyle.Currency(code: "USD").scale(1e-6)
) + Text("M")
```

Thanks to the `Decimal.FormatStyle.Currency` struct, we could format the sales to the currency used, US dollars, and we used a scale of $1/10^6$ to set the unit to a million. For example, with this formatting, we show the sales quantity of 5,321,878 dollars as \$5.32M.

For the colors of the slices, we used the `chartForegroundColorScale` modifier on the chart:

```
.chartForegroundColorScale(
[
```

```
        City.houston: .red,  
        .dallas: .yellow,  
        .sanAntonio: .brown,  
        .austin: .cyan  
    ]  
)
```

See also

- Decimal type: <https://developer.apple.com/documentation/foundation/decimal>
- SectorMark documentation: <https://developer.apple.com/documentation/charts/sectormark>

Interactive charts: selection

In this recipe, we are going to learn how to add interactivity to our charts and, specifically, how to select data on a chart.

We will build a financial chart to understand the price of Apple stock for the month of August 2023. We will use a specialized financial chart called a `candlestick` chart. A candlestick chart is a type of range chart that displays four stock prices, using two superimposed bars. A thicker bar, the candle, shows the range of prices, from the price at the opening of the trading day to the price at the closing. A thinner bar, the stick, shows the price range of the stock, from the lowest price for the trading day to the highest price for the day. If you're not familiar with range charts, please refer to the recipe *Different types of charts: marks and mark configuration* in this chapter. For a detailed explanation of a candlestick chart, please refer to the link provided at the end of the recipe.

Below the price chart, we will display a second chart with the trading volume of the day, which is the total number of shares that were traded on the day.

We will add interactivity by enabling users to tap on a price or volume for a day, which will then display the price details for the trading day in a numerical way, using annotations.

Getting ready

Create a new SwiftUI iOS app named `ChartSelection`.

How to do it...

Our app will have a main view, which will include the interactive charts, and a couple of views for the detailed price annotations. We will also have a data layer in a separate file. Let's start working on the data first and then switch to the views. These are the steps:

1. Drag and drop the `AAPL_Aug_2023.json` file from the book's Resources folder at GitHub (<https://github.com/PacktPublishing/SwiftUI-Cookbook-3rd-Edition/tree/main/Resources/Chapter15/Recipe06>) into the project's main folder. Make sure the file is added to the target.

2. Inspect the JSON file to understand how our data is organized. We can see that the top-level object is an array of objects, each representing the trading details for the stock. The object has several properties, one date, five prices, and the trading volume. The names of the properties are self-explanatory:

```
[  
 {  
   "date": "2023-08-01",  
   "open": 196.240005,  
   "high": 196.729996,  
   "low": 195.279999,  
   "close": 195.610001,  
   "prev_close": 196.449997,  
   "volume": 35175100  
 },  
 ...  
 ]
```

3. We will model our data for the stock prices with two structures. Create a new Swift file named `Stock.swift`. Start by creating a struct `StockPrice` to model the trading prices for a specific day:

```
struct StockPrice: Decodable {  
   let date: Date  
   let open: Double  
   let high: Double  
   let low: Double  
   let close: Double  
   let prevClose: Double  
   let volume: Double  
}
```

4. Create a struct `Stock` to model our stock. This new type will have some attributes for the stock, such as the company name, the stock exchange name, the ticker symbol, the currency, and, of course, an array with the stock prices. We will modify the initializer to sort the stock data by date, so our charts will get the prices in chronological order:

```
struct Stock {  
   let ticker: String  
   let company: String  
   let exchange: String  
   let currency: String  
   let prices: [StockPrice]
```

```
        init(ticker: String, company: String, exchange: String, currency:  
String, prices: [StockPrice]) {  
    self.ticker = ticker  
    self.company = company  
    self.exchange = exchange  
    self.currency = currency  
    self.prices = prices.sorted { $0.date < $1.date }  
}  
}
```

5. We are ready now to read the data from the JSON file and decode it into our own struct instances. A few things to mention are that we are getting the file from the main bundle, we need a custom `DateFormatter` to decode the dates, and we need to convert the names of the JSON properties in `snake_case` to `camelCase`, which is the standard in Swift. Note the use of the `assert` functions for internal sanity checks before force-unwrapping optional values. The code should be this:

```
extension Stock {  
    static var AAPL: Stock = {  
        let url = Bundle.main.url(  
            forResource: "AAPL_Aug_2023",  
            withExtension: "json"  
        )  
        assert(url != nil)  
        let data = try? Data(contentsOf: url!)  
        assert(data != nil)  
        let dateFormatter = DateFormatter()  
        dateFormatter.dateFormat = "yyyy-MM-dd"  
        let decoder = JSONDecoder()  
        decoder.keyDecodingStrategy = .convertFromSnakeCase  
        decoder.dateDecodingStrategy = .formatted(dateFormatter)  
        let array = try? decoder.decode([StockPrice].self, from: data!)  
        assert(array != nil)  
        return Stock(  
            ticker: "AAPL",  
            company: "Apple Inc.",  
            exchange: "NASDAQ",  
            currency: "USD",  
            prices:array!  
        )  
    }()  
}
```

6. To finish with our model layer, create a new Swift file named `Stock+Charts.swift`, where we will place some type extensions needed for our charts. It is a practice from the Objective-C days to name a file with type extensions using the type's name, a plus sign, and another name indicating what the extensions are for. Add this code, which will be used in the candlestick charts:

```
extension StockPrice {  
    var closedHigher: Bool { close > open }  
    var closedHigherThanPrevious: Bool { close > prevClose }  
}
```

7. Now, let's add two computed properties that will be used to calculate the price range for the price chart. The properties will return the highest and the lowest price of the series. Implement the following code:

```
extension Stock {  
    var highestPrice: StockPrice {  
        prices.max { $0.high < $1.high }!  
    }  
    var lowestPrice: StockPrice {  
        prices.min { $0.low < $1.low }!  
    }  
}
```

8. After finishing the model layer, we will start working on the views. First, create a new SwiftUI view named `TickerView.swift`. This view will be used in our root view. Replace the contents of the file with the following code:

```
import SwiftUI  
  
struct TickerView: View {  
    var stock: Stock  
    var body: some View {  
        VStack(alignment: .leading) {  
            HStack(alignment: .lastTextBaseline) {  
                Text(stock.ticker)  
                    .font(.largeTitle)  
                    .fontWeight(.bold)  
                Text(stock.company)  
                    .font(.caption)  
            }  
            Text("\(stock.exchange) - \(stock.currency)")  
                .font(.footnote)  
        }  
    }  
}
```

```
}
```

```
#Preview {
    TickerView(stock: .AAPL)
}
```

9. Open `ContentView.swift`, and build the basic layout of the root view. We will use a `VStack` layout, with the recently created `TickerView` view at the top, followed by a `Chart` view for the candlestick price chart, and a dummy `Color` view at the bottom, which will be replaced by the volume chart later in the recipe. Replace the content of the file with the following:

```
import SwiftUI
import Charts

struct ContentView: View {
    private let stock: Stock = .AAPL

    var body: some View {
        VStack(alignment: .leading) {
            TickerView(stock: stock)
            Chart {
                ForEach(stock.prices, id: \.date) { price in
                    let color: Color = price.closedHigher ? .green : .red
                    BarMark(x: .value("Date", price.date),
                            yStart: .value("Price", price.low),
                            yEnd: .value("Price", price.high),
                            width: 1)
                        .foregroundStyle(color)
                    BarMark(x: .value("Date", price.date),
                            yStart: .value("Price", price.open),
                            yEnd: .value("Price", price.close),
                            width: 8)
                        .foregroundStyle(color)
                }
            }
            Color.gray.frame(height: 100)
        }
        .padding()
    }
}

#Preview {
```

```
    ContentView()  
}
```

10. Use the live preview of the canvas to see what we have accomplished so far. Looking at the result we see we have a candlestick chart, but much of the chart plot area is blank, and we should remove the wasted space in the chart. We have highlighted this in the following screenshot:

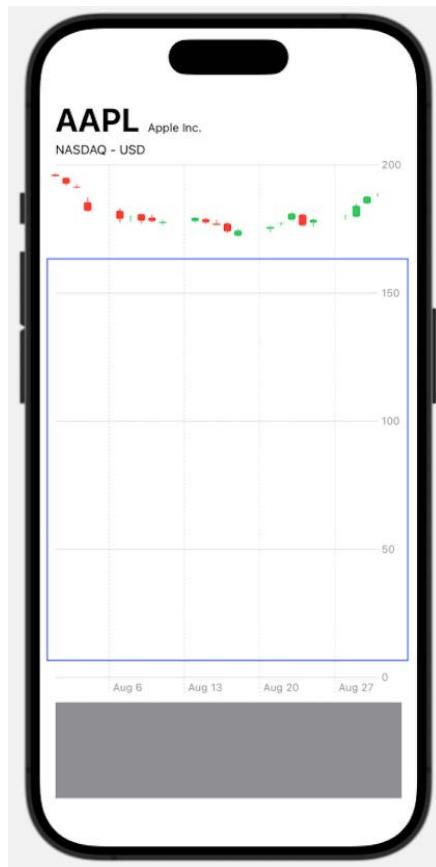


Figure 15.25: Preview of the first pass of the candlestick price chart

11. We will change the scale of the y-axis to be the range of prices. To accomplish this, we will use some computed variables. Switch to `Stock+Charts.swift` and create a new extension:

```
extension Stock {  
    var yMax: Double { highestPrice.high }  
    var yMin: Double { lowestPrice.low }  
    var yDomain: [Double] { [yMin, yMax] }  
}
```

12. Go back to the ContentView view, and add the following modifier to the Chart views to adjust the y-axis scale:

```
.chartYScale(domain: stock.yDomain)
```

13. Use the live preview of the canvas to preview our changes. Looking at the result, we can see we have a better presentation, but we have some discontinuities in the x-axis, which we have highlighted in the following screenshot:

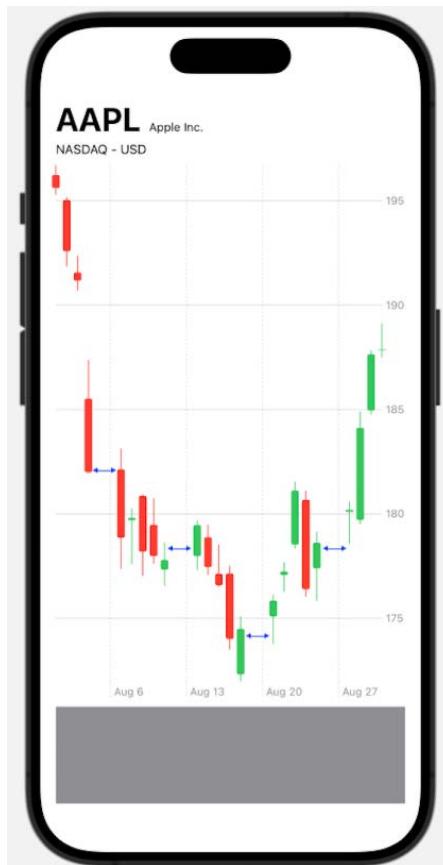


Figure 15.26: Preview of the candlestick price chart after the y-axis scale adjustment

14. We used dates for the x-axis. The month of August has 31 days, but the stock only traded during weekdays and not weekends, so we only have 22 trading days. To eliminate these discontinuities, we will have to change our x-axis to use a different variable. The best choice is to use the index of the price in the prices array. Modify the ForEach to use the `indices` property of the `prices` array. The final code should be this:

```
Chart {  
    ForEach(stock.prices.indices, id: \.self) { index in
```

```
let price = stock.prices[index]
let color: Color = price.closedHigher ? .green : .red
BarMark(x: .value("Index", index),
        yStart: .value("Price", price.low),
        yEnd: .value("Price", price.high),
        width: 1)
.foregroundColor(color)
BarMark(x: .value("Index", index),
        yStart: .value("Price", price.open),
        yEnd: .value("Price", price.close),
        width: 8)
.foregroundColor(color)
}
}
.chartYScale(domain: stock.yDomain)
```

15. Use the live preview of the canvas to preview our changes. Looking at the result, we can see that we no longer have discontinuities in the x-axis. However, more changes need to be made to it. We need to change the scale because of the blank space to the right, and we need to change the labels, which display the array indices instead of dates. We have highlighted this in the following screenshot:



Figure 15.27: Preview of the candlestick price chart after the x-axis variable change

16. First, switch to `Stock+Charts.swift`, and add one computed variable to fix the domain for the x-axis. Add the variable to the extension we created in step 10. The final code should be this:

```
extension Stock {  
    var xDomain: [Int] { [0, prices.count - 1] }  
    var yMax: Double { highestPrice.high }  
    var yMin: Double { lowestPrice.low }  
    var yDomain: [Double] { [yMin, yMax] }  
}
```

17. Now, before we continue working on the candlestick price chart, let's extract the code to a separate file. Create a new SwiftUI file named `PriceView.swift`. Replace the content of the file with the following code:

```
import SwiftUI
import Charts

struct PriceView: View {
    var stock: Stock
    var body: some View {
        Chart {
            ForEach(stock.prices.indices, id: \.self) { index in
                let price = stock.prices[index]
                let color: Color = price.closedHigher ? .green : .red
                BarMark(x: .value("Index", index),
                        yStart: .value("Price", price.low),
                        yEnd: .value("Price", price.high),
                        width: 1)
                    .foregroundStyle(color)
                BarMark(x: .value("Index", index),
                        yStart: .value("Price", price.open),
                        yEnd: .value("Price", price.close),
                        width: 8)
                    .foregroundStyle(color)
            }
        }
        .chartXScale(domain: stock.xDomain)
        .chartYScale(domain: stock.yDomain)
    }
}

#Preview {
    PriceView(stock: .AAPL)
    .frame(height: 500)
    .padding()
}
```

18. Use the live preview of the canvas to preview how our `PriceView` looks so far. After adjusting the scales, the chart looks as we expected, except for the labels. The preview should look like the following screenshot:

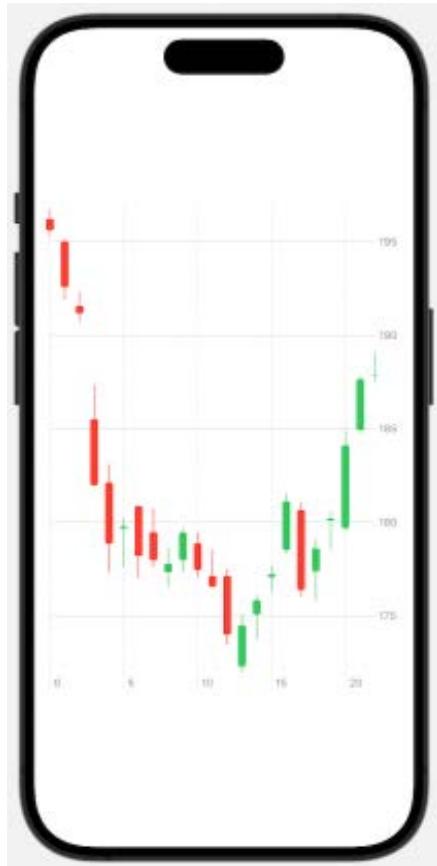


Figure 15.28: Preview of the PriceView view

19. Now, before we continue working on the candlestick price chart, let's remove the chart code from the `ContentView` view and replace it with a `Color.orange` view. We know that when we finish with our code, we will have to replace the orange view with the candlestick price chart and the gray view with the volume chart. After the changes, the content of the `body` variable should be this:

```
var body: some View {
    VStack(alignment: .leading) {
        TickerView(stock: stock)
        Color.orange
        Color.gray.frame(height: 100)
    }
    .padding()
}
```

20. Also, if you use the live preview of the canvas to preview the `ContentView` view with the two dummy views, it should look like the following screenshot:

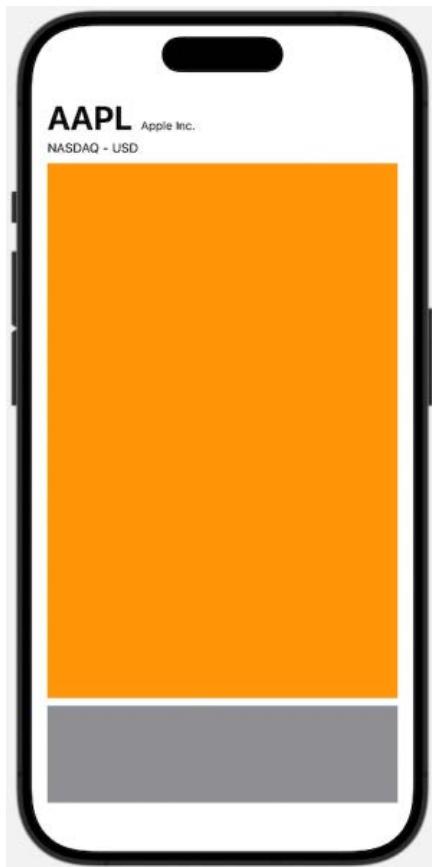


Figure 15.29: Preview of the `ContentView` view with dummy views

21. Switch to the `PriceView` and change the x-axis labels to display the days of the month, instead of the indices of the array. For the first day of the month, we will use an extra label to display the month and the year. We will finish the configuration of the x-axis by displaying a gridline for each day. Add the following modifier at the end of the `Chart` view:

```
.chartXAxis {  
    let minX = stock.xDomain[0]  
    let maxX = stock.xDomain[1]  
    let values = Array(stride(from: minX, through: maxX, by: 1))  
    AxisMarks(values: values) { value in  
        AxisGridLine()  
    }  
}
```

```
let index = value.as(Int.self)!  
let day = stock.prices[index].date.formatted(.dateTime.day())  
AxisValueLabel(day, horizontalSpacing: 0)  
if index == 0 {  
    let monthYear = stock.prices[index].date.formatted(.dateTime.  
month().year())  
    AxisValueLabel(monthYear, horizontalSpacing: 0,  
verticalSpacing: 20)  
}  
}  
}
```

22. Use the live preview of the canvas to preview our progress. The preview should look like this:



Figure 15.30: Preview of the PriceView view with the final x-axis labels

23. It's time to customize the y-axis. We would like to add gridlines for each dollar and keep the default labels and gridlines when the price is a multiple of five. Add the following modifier at the end of the Chart view:

```
.chartYAxis {  
    let minY = stock.yDomain[0].rounded(.up)  
    let maxY = stock.yDomain[1].rounded(.down)  
    let values = Array(stride(from: minY, through: maxY, by: 1))  
    AxisMarks(values: values) { value in  
        let price = value.as(Int.self)!  
        if price % 5 != 0 {  
            AxisGridLine(stroke: .init(lineWidth: 0.25, dash: [3, 2]))  
        } else {  
            AxisGridLine()  
            AxisTick()  
            AxisValueLabel()  
        }  
    }  
}
```

24. Use the live preview of the canvas to preview our finalized candlestick price chart. The preview should look like this:



Figure 15.31: Preview of the finalized PriceView view

25. Now that we have our price view finalized, let's work on the selection. Selection is very simple to implement, with just a view modifier and a `@State` property. We will implement the selection on the x-axis. Add the following property to the `PriceView` struct:

```
    @State var selectedIndex: Int?
```

26. Then, add the following modifier at the end of the `Chart` view:

```
    .chartXSelection(value: $selectedIndex)
```

27. In our chart, to know if the specific candlestick bar has been selected, add the following code at the top of the `ForEach` view:

```
let isSelected = selectedIndex != nil && index == selectedIndex!
```

28. Now, we will change the color of the candlestick selected by the user to `yellow`. Modify the `foregroundStyle` modifier for both `BarMark` instances with the following code:

```
.foregroundStyle(isSelected ? .yellow : color)
```

29. Use the live preview of the canvas to preview our selection code so far. You'll notice that when you select a candlestick, its color changes. To select a candlestick, click and hold on it, using your trackpad or mouse. The preview should look like this:



Figure 15.32: Preview of the `PriceView` view with a candlestick selected

30. Changing the color on the selected bar is cool but not very useful to the user. To make it more appealing, we will add an overlay with some information regarding the prices for the selected candlestick. Let's create a custom view to show information about the prices. Create a new SwiftUI file named `PriceDetailView.swift`. Replace its contents with the following code:

```
struct PriceDetailView: View {
    var ticker: String
    var price: StockPrice
    private let priceFormat: FloatingPointFormatStyle<Double> = .number.
precision(.fractionLength(2...2))

    var body: some View {
        VStack {
            Text("\(ticker) - \(price.date, format: .dateTime.month(.abbreviated).day())")
                .font(.callout)
            Group {
                Text("O: \(price.open, format: priceFormat) - C: \(Text(price.close, format: priceFormat).foregroundStyle(price.closedHigher ? .green : .red))")
                    Text("H: \(price.high, format: priceFormat) - L: \(price.low, format: priceFormat)")
                }
                .font(.caption2)
                .monospaced()
            }
            .padding(8)
            .background {
                RoundedRectangle(cornerRadius: 4)
                    .foregroundStyle(.gray.opacity(0.2))
            }
        }
    }

    #Preview {
        VStack {
            PriceDetailView(ticker: StockAAPL.ticker, price: StockAAPL.prices[0])
            PriceDetailView(ticker: StockAAPL.ticker, price: StockAAPL.prices[5])
        }
    }
}
```

31. With the live preview of the canvas, we can see the preview for a couple of instances of this newly created view. The preview should look like this:



Figure 15.33: Preview of two PriceDetailView instances

32. Now, let's add this new view to our price chart so that when the user selects a candlestick, we display the additional information. We will use a RuleMark with an annotation, which will include our PriceDetailView view. Switch back to PriceView, and add the following code inside the Chart view but immediately after the ForEach view:

```
Chart {  
    ForEach(stock.prices.indices, id: \.self) { index in  
        // ...  
    }  
    if let index = selectedIndex {  
        let price = stock.prices[index]  
        RuleMark(x: .value("Selected", index))  
            .foregroundStyle(.gray.opacity(0.3))
```

```
        .offset(yStart: 0)
        .zIndex(-1)
        .annotation(
            position: .top,
            overflowResolution: .init(x: .fit(to: .chart), y: .disabled)
        ) {
            PriceDetailView(ticker: stock.ticker, price: price)
        }
    }
}
```

33. Use the live preview of the canvas to interact with the view and see how, when a candlestick is selected, its color changes, and the rule with the custom annotation shows at the top. The preview should look like this:



Figure 15.34: Preview of finalized PriceView with selection

34. To make the preview macro more useful, you could pass an initial value to the `selectedIndex` property. Replace the `#Preview` macro with the following:

```
#Preview {
    PriceView(stock: .AAPL, selectedIndex: .constant(6))
        .frame(height:500)
        .padding()
}
```

35. Now, we will work on our volume view. It is simpler than our price view and uses a similar approach, iterating through the indices of the array to remove the discontinuities in the x-axis. Create a new SwiftUI file named `VolumeView.swift`. Replace its contents with the following code:

```
import SwiftUI
import Charts

struct VolumeView: View {
    var stock: Stock

    var body: some View {
        Chart {
            ForEach(stock.prices.indices, id: \.self) { index in
                let price = stock.prices[index]
                let color: Color = price.closedHigherThanPrevious ?
                    .green : .red
                BarMark(x: .value("Index", index),
                        y: .value("Volume", price.volume),
                        width: 14)
                    .foregroundStyle(color)
            }
        }
        .frame(height: 100)
        .chartXScale(domain: stock.xDomain)
        .chartXAxis(.hidden)
        .chartYAxis {
            let format: Decimal.FormatStyle =
                .number.scale(1e-6) .precision(.fractionLength(0))
            AxisMarks { value in
                AxisGridLine()
                AxisValueLabel(format: format, horizontalSpacing: 8)
            }
        }
    }
}
```

```
        }
    }

#Preview {
    VolumeView(stock: .AAPL)
        .padding()
}
```

36. Use the live preview of the canvas to preview our volume view. The preview should look like this:



Figure 15.35: Preview of the VolumeView

37. Let's implement selection for the VolumeView too. When the user selects a volume bar, we will change the color of the bar and add an overlay that will display the volume information. We will create a custom view to show as an overlay. Create a new SwiftUI file named `VolumeDetailView.swift`.

Replace its contents with the following code:

```
import SwiftUI

struct VolumeDetailView: View {
    var price: StockPrice
    private let volumeFormat: FloatingPointFormatStyle<Double> =
        .number.scale(1e-6).precision(.fractionLength(1...1))

    var body: some View {
        HStack(spacing: 0) {
            Text("VOL: ")
            Text("\(price.volume, format: volumeFormat)M")
                .foregroundStyle(price.closedHigherThanPrevious ? .green : .red)
                .bold()
        }
        .font(.caption2)
        .monospaced()
        .padding(8)
        .background {
            RoundedRectangle(cornerRadius: 4)
                .foregroundStyle(.gray.opacity(0.2))
        }
    }
}

#Preview {
    VStack {
        VolumeDetailView(price: StockAAPL.prices[5])
        VolumeDetailView(price: StockAAPL.prices[8])
    }
}
```

38. With the live preview of the canvas, we can see the preview for a couple of instances of this newly created view. The preview should look like this:

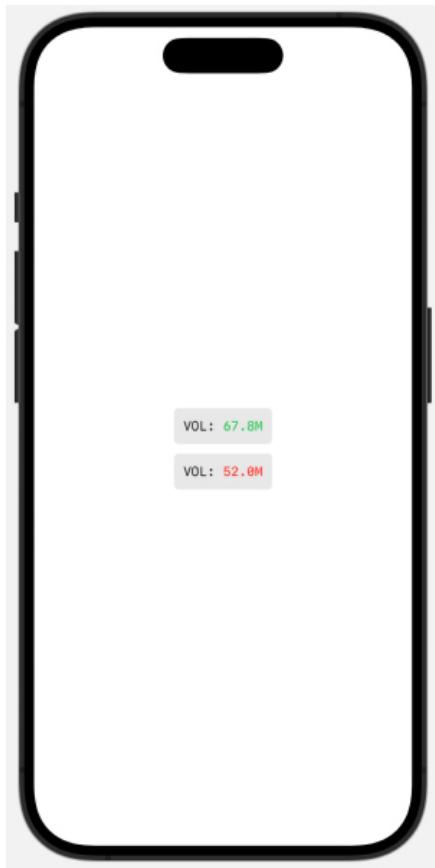


Figure 15.36: Preview of two VolumeDetailView instances

39. Switch back to `VolumeView` and work on the selection on the x-axis. We will implement the selection on the x-axis. Add the following `@State` property to the `VolumeView` struct:

```
    @State var selectedIndex: Int?
```

40. Then, add the following modifier at the end of the `Chart` view:

```
    .chartXSelection(value: $selectedIndex)
```

41. In our chart, to know if the specific volume bar has been selected, add the following code at the top of the `ForEach` view:

```
    let isSelected = selectedIndex != nil && index == selectedIndex!
```

42. Change the color of the volume bar selected by the user to *yellow*. Modify the `foregroundStyle` modifier to the `BarMark` with the following code:

```
    .foregroundStyle(isSelected ? .yellow : color)
```

43. To display the additional information about the volume, we will use a `RuleMark` with an annotation, which will include our `VolumeDetailView` view. Add the following code inside the `Chart` view but immediately after the `ForEach` view:

```
if let index = selectedIndex {  
    let price = stock.prices[index]  
    RuleMark(x: .value("Selected", index))  
        .foregroundStyle(.gray.opacity(0.3))  
        .offset(yStart: 25)  
        .zIndex(-1)  
        .annotation(  
            position: .top,  
            overflowResolution: .init(x: .fit(to: .chart), y: .disabled)  
        ) {  
            VolumeDetailView(price: price)  
        }  
}
```

44. To make the preview macro more useful, you could pass an initial value to the `selectedIndex` property. Replace the `#Preview` macro with the following:

```
#Preview {  
    VolumeView(stock: .AAPL, selectedIndex: 8)  
        .padding()  
}
```

45. Use the live preview of the canvas to interact with the view and see how, when a volume bar is selected, its color changes, and the rule with the custom annotation shows at the top. The preview should look like this:



Figure 15.37: Preview of finalized VolumeView with selection

46. Finally, replace the dummy views in our root view with the custom views we built. Go to `ContentView`, and replace the contents of the `body` variable with the following:

```
var body: some View {
    VStack(alignment: .leading) {
        TickerView(stock: stock)
        PriceView(stock: stock)
        VolumeView(stock: stock)
    }
    .padding()
}
```

47. Use the live preview of the canvas to interact with the app. The functionality we implemented works as expected; however, we detected one issue. The custom annotation with the prices at the top of the screen overlaps the header view. We can also see that we can select a volume bar and a candlestick bar on different days. We have highlighted these in the following screenshot:



Figure 15.38: Preview of the app with one issue

48. To fix the presentation issue, the `ContentView` view needs to know when the annotation is selected. The best way to know about this is to share the state of the selection with the `PriceView`. To accomplish this, we will use the `@Binding` property wrapper. If you are not familiar with this, we recommend you review the recipe of *Chapter 10, Using @Binding to Pass a State Variable to Child Views*. At the top of the struct, create a `@State` variable:

```
    @State private var selectedIndex: Int?
```

Go to `PriceView`, and replace the `@State` variable with a `@Binding`. Also, modify the preview macro to pass a binding instead of a value:

```
@Binding var selectedIndex: Int?  
...  
#Preview {  
    PriceView(stock: .AAPL, selectedIndex: .constant(6))  
        .frame(height:500)  
        .padding()  
}
```

49. Go back to `ContentView`, and make two modifications. Pass a binding to our `@State` variable as a parameter to `PriceView`, and hide `TickerView` when the user selects a candlestick bar. After these changes, the first two views in the `VStack` inside the `body` variable should be this:

```
TickerView(stock: stock)  
    .opacity(selectedIndex == nil ? 1: 0)  
PriceView(stock: stock, selectedIndex: $selectedIndex)
```

50. To enhance our user experience, let's synchronize the selection of the candlestick price bar with the selection of the volume var. Go to `VolumeView`, and replace the `@State` variable with a `@Binding`. Also, modify the preview macro to pass a binding instead of a value:

```
@Binding var selectedIndex: Int?  
...  
#Preview {  
    VolumeView(stock: .AAPL, selectedIndex: .constant(8))  
        .padding()  
}
```

51. Go back to `ContentView`, and make the last modification. Pass a binding to our `@State` variable as a parameter to `VolumeView`. After these changes, the contents of the `body` variable should be this:

```
var body: some View {  
    VStack(alignment: .leading) {  
        TickerView(stock: stock)  
            .opacity(selectedIndex == nil ? 1: 0)  
        PriceView(stock: stock, selectedIndex: $selectedIndex)  
        VolumeView(stock: stock, selectedIndex: $selectedIndex)  
    }  
    .padding()  
}
```

52. Use the live preview of the canvas to interact with the final version of the app. With the latest changes, the header view is hidden when the user selects a bar, and the selected price bar and volume vars are synchronized. The result should be like the following screenshot:



Figure 15.39: Preview of the final version of the app

How it works...

At the start of the recipe, we prepared our data so that it could be easily used in our price charts. We encountered a discontinuity issue in the x-axis, and to solve it, instead of using dates, we decided to use the index of the price in the array of prices.

We started by modeling our data with two structures. We declared the `struct StockPrice`, which includes the different prices and volumes for the stock on a specific date. We created the `struct Stock` with four `String` properties, for the name, ticker symbol, exchange and currency of the stock, and an array of type `StockPrice` to hold the prices. We included an initializer to sort the prices by date in ascending order.

This guaranteed that our stock charts would be plotted correctly:

```
init(ticker: String, company: String, exchange: String, currency: String,  
prices: [StockPrice]) {  
    ...  
    self.prices = prices.sorted { $0.date < $1.date }  
}
```

We created a couple of extensions to our structures with computed variables, which we used in our charts. Of special interest are the variables to drive the scale of the axes of the charts:

```
extension Stock {  
    var xDomain: [Int] { [0, prices.count - 1] }  
    var yMax: Double { highestPrice.high }  
    var yMin: Double { lowestPrice.low }  
    var yDomain: [Double] { [yMin, yMax] }  
}
```

For our root view, we used the traditional vertical layout with a `VStack`, with three custom views. At the top, `TickerView` displayed the details of a stock, `PriceView` in the middle displayed the candlestick price chart, and `VolumeView` at the bottom displayed the volume chart.

For the charts, we iterated over our dataset with a `ForEach` view, and we decided to use the indices of the prices array:

```
ForEach(stock.prices.indices, id: \.self) { index in ... }
```

We used two `BarMark` instances to plot each candlestick bar: one wider mark for the real body or candle, which plotted the price range between the opening and closing prices, and another mark, just one point wide, for the shadow or stick, which plotted the price range between the highest and lowest prices. We also changed the color of the marks, depending on whether the stock closed higher than its opening price (green) or lower (red).

We customized both axes with labels and gridlines. If you are not familiar with this type of customization, please refer to the recipe *Customizing charts: axes, annotations, and rules* in this chapter.

We implemented the selection of the prices using the following view modifier:

```
.chartXSelection(value: $selectedIndex)
```

This modifier was introduced in iOS 17, and it takes care of converting the coordinates of the touch point to chart x-axis units. It is simple and powerful, and it avoids the nightmare of coordinate calculations. You may encounter some tutorials online that still use gestures and a `ChartProxy` object to perform selection, but this is no longer necessary.

The modifier used a `@State` property to store the selected index:

```
@State private var selectedIndex: Int?
```

When a user taps on a price in the chart, Swift Charts automatically sets the value of the property to an `Int`, corresponding to the index of the selected price in the `prices` array. This greatly simplified our code to detect bar selection:

```
let isSelected = selectedIndex != nil && index == selectedIndex!
```

On selection, we changed the color of the bars to yellow, giving the user appropriate feedback. We also displayed an annotation with the rule at the mark's x-axis position. Since the annotation was centered above the rule, without special configuration, it ended up being partially out of the screen. We used a modifier on the annotation to handle the overflow to avoid this:

```
.annotation(position: .top, overflowResolution: .init(x: .fit(to: .chart), y: .disabled))
```

Toward the end of the recipe, we synchronized the selection of the price chart with the volume chart and the `ContentView`. We used a `@State` variable in the parent view and implemented `@Binding` variables in the child views.

There's more

There is a variant of the chart selection modifier, `.chartXSelection(range:)`, which allows us to select a range of values with a two-finger tap in iOS. If we wanted to provide our own custom gesture, we could use a `.chartGesture` modifier with a `ChartProxy` to convert the gesture coordinates to x-axis values:

```
.chartGesture { proxy in
    DragGesture(minimumDistance: 0)
        .onChanged { value in
            proxy.selectXValue(at: value.location.x)
        }
        .onEnded { _ in
            selectedIndex = nil
        }
}
```

`selectXValue(at:)` is another function added in iOS 17, which greatly simplifies coordinate conversions between screen points and chart units.

See also

- About candlestick charts: <https://www.investopedia.com/trading/candlestick-charting-what-is-it/>
- `ChartProxy` documentation: <https://developer.apple.com/documentation/charts/chartproxy>
- `DragGesture` documentation: <https://developer.apple.com/documentation/swiftui/draggesture>

Interactive charts: scrollable content

In this recipe, we are going to learn how to manage scrollable content in our charts. When we have a large dataset, it would be impractical to display it all at once on the screen, so the content must be scrollable. In this recipe, we will learn how to add scrolling to our charts.

Like what we did in the recipe *Interactive charts: selection*, we will build a financial chart to understand the price of Apple stock through its trading history. We will plot a line chart with the prices, using the y-axis to show the price and the x-axis to show the date.

For the y-axis, we will adjust the scale dynamically, depending on the range of prices shown on the screen. We will provide a picker above the chart to choose the scale of the x-axis. Every time the user picks a different setting from the picker, we will reset the chart to show the last price of the series to the right.

Getting ready

Create a new SwiftUI iOS app named `ScrollableCharts`.

How to do it...

The root view of our app will have the traditional vertical layout provided by a `VStack`, which will include a custom view at the top, followed by a `Picker` view and then the `Chart` view, which will occupy most of the screen. We will also have a data layer in a separate file. Let's start working on the data first, and then switch to the views. These are the steps:

1. We will reuse two files from the recipe *Interactive charts: selection* in this chapter. Copy the files `Stock.swift` and `TickerView.swift` to the project's folder. Make sure both files are added to the target.
2. Drag and drop the `AAPL_weekly_max.json` file from the book's Resources folder at GitHub (<https://github.com/PacktPublishing/SwiftUI-Cookbook-3rd-Edition/tree/main/Resources/Chapter15/Recipe07>) into the project's `main` folder. Make sure the file is added to the target.
3. The format of the JSON file is the same as the format of the JSON file used in the previous recipe, mentioned in step 1. This allows us to reuse the code to model our data. The only change we must make is to read the new JSON file instead of the old. Go to `Stock.swift`, and in the extension to `Stock`, change the name of the file. After the change, the code should look like this:

```
extension Stock {  
    static var AAPL: Stock = {  
        let url = Bundle.main.url(  
            forResource: "AAPL_weekly_max",  
            withExtension: "json"  
        )  
        ...  
    }()  
}
```

- As we did in the previous recipe, create a new Swift file named `Stock+Charts.swift`, where we will place some type extensions needed for our charts. In the newly created file, we will define an enum `DateRange`, which we will use to store the range of dates to display on the x-axis. Add this code to the file:

```
enum DateRange: String, CaseIterable {
    case oneYear = "1Y"
    case twoYears = "2Y"
    case fiveYears = "5Y"
    case tenYears = "10Y"

    var isMonthly: Bool {
        self == .oneYear || self == .twoYears
    }
    var timeInterval: TimeInterval {
        return switch self {
            case .oneYear: 31536000
            case .twoYears: 63072000
            case .fiveYears: 157766400
            case .tenYears: 315532800
        }
    }
    func startDate(from endDate: Date) -> Date {
        endDate - timeInterval
    }
}
```

- Now, let's extend `Stock` with a computed property to obtain the date of the last sample in our dataset, and a function, which will be used to calculate the price range for the chart, given a range of dates. Implement the following code, right after the code of the previous step:

```
extension Stock {
    var endDate: Date { prices.last!.date }
    func yDomain(from date1: Date, for range: DateRange) ->
        ClosedRange<Double> {
        let date2 = date1 + range.timeInterval
        let subset = prices.filter { (date1 <= $0.date) && ( $0.date <=
date2) }
        let high = subset.max { $0.close < $1.close }!
        let low = subset.min { $0.close < $1.close }!
        return low...high
    }
}
```

- After finishing the model layer, we will start working on the chart. Open `ContentView.swift` and build the basic layout of the root view. We will use a `VStack` layout, with a `TickerView` view at the top, followed by a `Picker` view and a `Chart` view for the price chart. Replace the `ContentView` struct of the file with the following:

```
struct ContentView: View {  
    private let stock: Stock = .AAPL  
    @State private var range: DateRange = .oneYear  
    @State private var start: Date = Stock.AAPL.endDate  
    var body: some View {  
        VStack(alignment: .leading) {  
            TickerView(stock: stock)  
            Picker("Date range", selection: $range) {  
                ForEach(DateRange.allCases, id: \.self) { range in  
                    Text(range.rawValue).tag(range)  
                }  
            }  
            .pickerStyle(.segmented)  
            Chart {  
                ForEach(stock.prices, id: \.date) { price in  
                    LineMark(  
                        x: .value("Date", price.date),  
                        y: .value("Price", price.close)  
                    )  
                    AreaMark(  
                        x: .value("Date", price.date),  
                        y: .value("Price", price.close)  
                    )  
                    .foregroundStyle(.blue.opacity(0.4))  
                }  
            }  
        }  
        .padding()  
    }  
}
```

- Use the live preview of the canvas to see what we have accomplished so far. Looking at the result we can see we have all the prices for the Apple stock since the 1980s. However, the data for the last century is not distinguishable because the prices were in the pennies compared to today's prices in the hundreds of dollars.

We have highlighted this in the following screenshot:

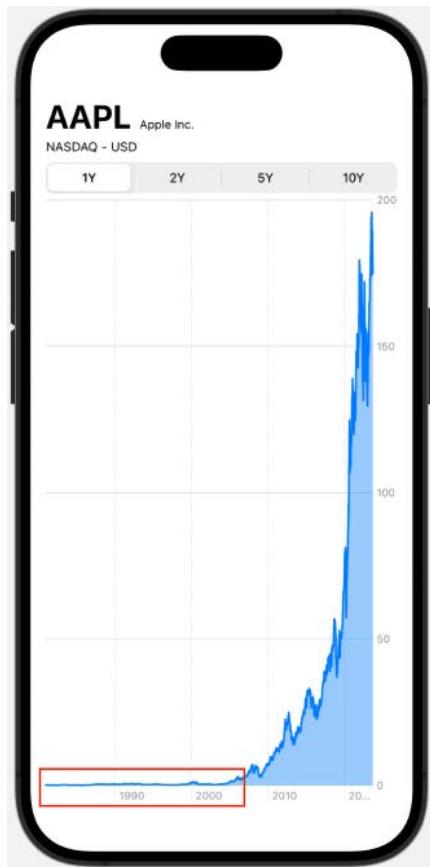


Figure 15.40: Preview of the first pass of the price chart

8. We will change the scale of the x-axis to be the range of dates reflected in the picker: one, two, five, or ten years; and we will make the chart scrollable. Add three view modifiers to the `Chart` view to accomplish this:

```
Chart { ... }
    .chartXVisibleDomain(length: range.timeInterval)
    .chartScrollableAxes(.horizontal)
    .chartScrollPosition(x: $start)
```

9. Use the live preview of the canvas to preview our changes. Interact with the scrolling and date picker. After scrolling for a while or using the date picker, the chart does not look right. Select 5Y and select 1Y; you'll see that the dates are from 1988. The prices displayed differ from the initial prices because the date range is different. We have highlighted this in the following screenshot:

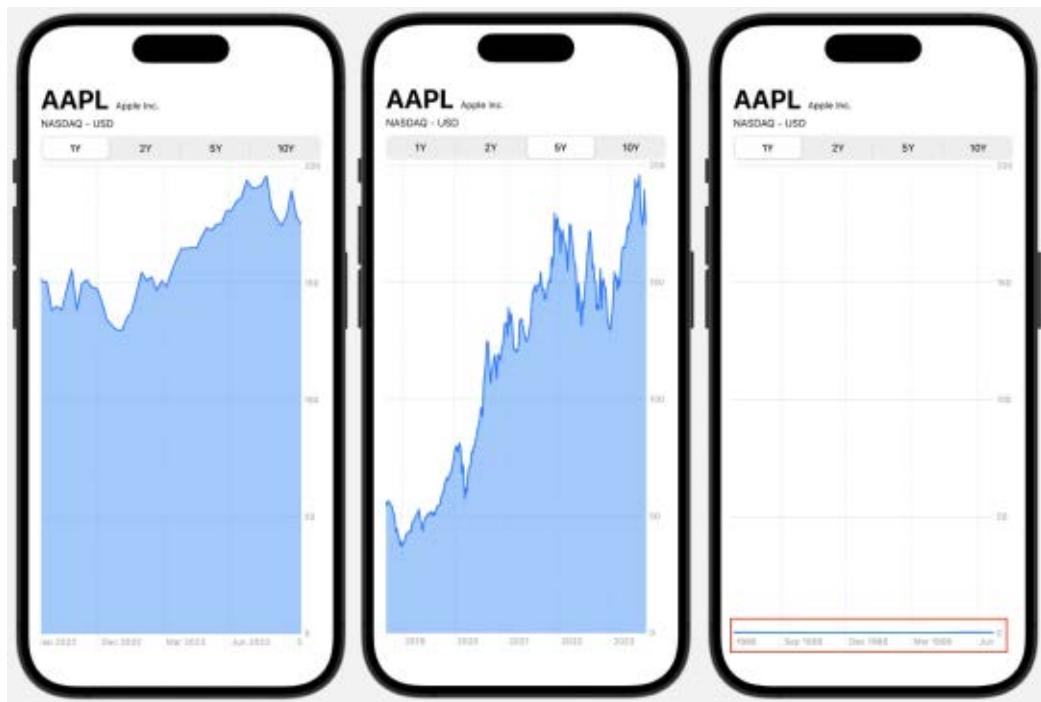


Figure 15.41: Screenshots of the chart with the date range picker implemented

10. To fix this behavior, each time we choose a different option on the picker, we will reset the date range so that it ends in the last price of our series. Add this additional modifier to the chart, right after the modifiers added in step 8:

```
.onChange(of: range, initial: true) {  
    start = range.startDate(from: stock.endDate)  
}
```

11. Let's customize our x-axis. For the one-year and two-year ranges, we want to display the month and the year in separate lines. For the five-year and ten-year ranges, we would only display the year. Add this modifier to the chart:

```
.chartXAxis {  
    AxisMarks(values: .automatic(desiredCount: 5)) {  
        if range.isMonthly {  
            AxisValueLabel(format: .dateTime.month(.abbreviated))  
        }  
        AxisValueLabel(format: .dateTime.year(), verticalSpacing: 16)  
        AxisTick()  
        AxisGridLine()  
    }  
}
```

12. Use the live preview of the canvas to preview our changes. Try the one-year and five-year scales. The results should be like the following screenshots:



Figure 15.42: Screenshots of the chart with the x-axis custom labels

13. To customize our y-axis, we would have to adjust the scale dynamically by calculating the highest and the lowest prices for a given date range. To this effect, we already implemented a function on our Stock struct in step 5. Additionally, we will customize the format of the labels. Add these two modifiers to the Chart view:

```
.chartYScale(  
    domain: stock.yDomain(from: start, for: range)  
)  
.chartYAxis {  
    AxisMarks { value in  
        AxisGridLine()  
        let y = value.as(Double.self)!
```

```
        AxisValueLabel(String(format: "%5.1f", y))  
    }  
}
```

14. Use the live preview of the canvas to preview our changes. Try to scroll to dates from 20 years ago, and observe how the y-scale adjusts dynamically, depending on the price range for the selected date range. The results should be like the following screenshots:



Figure 15.43: Screenshots of the finalized app

How it works...

At the start of the recipe, we prepared our data so that it could be easily used in our price charts. We started by modeling our data with two structures, `StockPrice` and `Stock`. Refer to the recipe *Interactive charts: selection* in this chapter to get an explanation of the JSON dataset and the two structures.

We created an extension to our `Stock` struct with a computed variable and a function. The variable provided the date of the most recent stock price, and we used this value every time the user changed the segmented picker value, which resulted in a change in the date range used in the x-axis.

The function calculated the range of prices for a given date range, and we used it to dynamically adjust the y-axis scale.

We created the enum `DateRange` type and used it in our `Picker` view. We added two computed variables and a function. For date range calculations, we used the computed variable `timeInterval` and the function `startDate` to calculate a start date, given the time interval for the selected date range.

The other computed variable, `isMonthly`, was used to drive the format of the labels for the x-axis.

For our root view, we used the traditional vertical layout with a `VStack`, with a custom view, a `Picker` view, and a `Chart` view. At the top, `TickerView` displayed the details of a stock, followed by a picker used to change the date range for the x-axis of the chart, and the historical prices of the Apple stock at the bottom of our chart.

For the chart, we iterated over our dataset with a `ForEach` view, and we used two marks, a `LineMark` and a `AreaMark`. The line mark plotted the price of the stock on the y-axis and the dates on the x-axis. The area mark was used for decoration to fill the area below the line.

We customized the labels for both axes. If you are not familiar with this type of customization, please refer to the recipe *Customizing charts: axes, annotations, and rules* in this chapter.

For the scrollable content, we used three modifiers. We implemented the selection of the prices using the following view modifiers:

```
.chartXVisibleDomain(length: range.timeInterval)
.chartScrollableAxes(.horizontal)
.chartScrollPosition(x: $start)
```

The three modifiers were introduced in iOS 17, and they are part of the new interactive functions added to Swift Charts in 2023. With `chartScrollableAxes(.horizontal)`, we configured the chart to scroll on the horizontal axis. This modifier took care of converting the drag gestures to horizontal scroll movements. It is simple and powerful, and it avoids the nightmare of coordinate calculations. You may encounter some tutorials online that still use gestures and a `ChartProxy` object to perform scrolling, but this is no longer necessary.

The `.chartScrollPosition(x: $start)` modifier used a `@State` property to store the current scroll position in x-axis units, which in our case are of type `Date`. And finally, the `.chartXVisibleDomain(length: range.timeInterval)` modifier sets the number of units shown in the x-axes. Since we used dates, `TimeInterval` is the unit used to calculate the difference between two dates.

To finalize the app, when the user changed the scale of the x-axis with the picker, we corrected the start date of the x-axis with the following modifier:

```
.onChange(of: range, initial: true) {
    start = range.startDate(from: stock.endDate)
}
```

We used our custom function `startDate(from:)` of `DateRange` to perform date arithmetic. We calculated our start date by subtracting the corresponding `TimeInterval` from the date of the last sample of our price series:

```
func startDate(from endDate: Date) -> Date {  
    endDate - timeInterval  
}
```

Thanks to these date calculations, we guaranteed that our chart showed the most current price series, ending in September 2023. We set the initial prices for our scrolling with two variables: `start` for the start date and `range.timeInterval` for the number of units shown in the x-axis, which is measured in `Timeinterval` units.

There's more

We can also select the behavior of the scrolling in the x-axis using the `.chartScrollTargetBehavior(_:)` view modifier. In our case, we could use the following code to make the scroll snap to the beginning of a month:

```
.chartScrollTargetBehavior(  
    .valueAligned(matching: DateComponents(day: 1))  
)
```

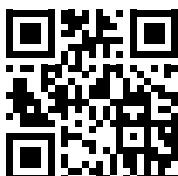
See also

Apple's documentation on `chartscrolltargetbehavior(_:)`: [https://developer.apple.com/documentation/swiftui/view/chartscrolltargetbehavior\(_:\)](https://developer.apple.com/documentation/swiftui/view/chartscrolltargetbehavior(_:))

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

<https://packt.link/swiftUI>



16

Creating Multiplatform Apps with SwiftUI

SwiftUI makes it easy to take some or all of 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 will create an iOS app and then reuse some components to create a macOS and a watchOS app.

When using multiplatform development in SwiftUI, we share common resources among platforms while creating other resources that are platform-specific. For example, models may be shared across platforms, but certain images, views, and controls are platform-specific. Creating platform-specific views allows us to follow best practices and design guidelines, which reach a broader audience and provide a user experience tailored to the platform.

Since the introduction of Xcode 14 in June 2022, Apple has provided a multiplatform target in Xcode. When you create a new iOS app using Xcode 14 or Xcode 15, the multiplatform target is used by default. The multiplatform target allows us to share a single target for iOS, iPadOS, macOS, visionOS, and tvOS, while the watchOS app remains in a separate target.

SwiftUI is in its fifth iteration and Apple has made big efforts to modify the SwiftUI views to be multiplatform, without the need for additional code. For example, `NavigationView` behaves differently depending on the platform used to run the app. SwiftUI takes care of the configuration automatically, while also providing configuration options to further refine the platform-specific behavior.

This chapter shows how to create an app for multiple platforms and best practices when targeting multiplatform apps in SwiftUI. We will start with an iOS app written in SwiftUI and evolve our code to target macOS and watchOS with the following recipes:

- Creating an iOS app in SwiftUI
- Creating the macOS version of the app with a new target
- Creating a multiplatform version of the app sharing the same target
- Creating the watchOS version of the iOS app



Note: The recipes in this chapter are based on the same Xcode project. You can skip to the recipe you're interested in but we recommend working on the recipes following the proposed order. If you decide to skip the order, we provide a starter project for each recipe in addition to the completed project.

Technical requirements

The code in this chapter is based on Xcode 15.0 and iOS 17.0. You can download and install the latest version of Xcode from the App Store. You'll also need to be running macOS Sonoma (14.0) or newer since we will be using the latest SwiftUI APIs on the Mac.

Simply search for Xcode in the App Store and select and download the latest version. Launch Xcode and follow any additional installation instructions that your system may prompt you with. Once Xcode has fully launched, you're ready to go.

All the code examples for this chapter can be found on GitHub at <https://github.com/PacktPublishing/SwiftUI-Cookbook-3rd-Edition/tree/main/Chapter16-Multiplatform-SwiftUI>.

Creating an iOS app in SwiftUI

Before going for multiplatform app development, we will 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 in Xcode 15*. The version created for this recipe will be 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, you can tap 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 iOS app in SwiftUI named `Multiplatform`. Make sure to select the iOS App template and not the Multiplatform template for this recipe.



Important Note:

The multiplatform template provided with Xcode creates an iOS app with a multiplatform target that already supports a native macOS app. This recipe creates an iOS app and it is the starting point for the following two recipes in the chapter. Using the multiplatform template instead of the iOS template will cause issues with the other recipes.

How to do it...

We will set up the model and data source used in this recipe, and then proceed to create the views needed to display the list of insects and the details of each one. The steps are as follows:

1. Since we are building a multiplatform app, and to keep our code more organized, let's create a group for our models:
 - a. Right-click on the `Multiplatform` folder in the navigation pane.
 - b. Select **New Group**.
 - c. Name the new folder `Models`.
2. Repeat the above steps and create two more groups, one named `Resources` and another one named `Views`.
3. Drag and drop the `Assets` catalog file into the `Resources` folder.
4. Add images for our insects to the project. Drag the images from the book's `Resources` folder from GitHub (<https://github.com/PacktPublishing/SwiftUI-Cookbook-3rd-Edition/tree/main/Resources/Chapter16/Recipe01>) into the `Assets` catalog.
5. Drag and drop the `insectData.json` file from the book's `Resources` folder from GitHub (<https://github.com/PacktPublishing/SwiftUI-Cookbook-3rd-Edition/tree/main/Resources/Chapter16/Recipe01>) into this project's `Resources` folder.
6. Click on `insectData.json` to view its content. From the data, we notice every insect has an ID, a name, an image name, a habitat, and a description. So, let's create a model type to match the JSON content.
7. 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 in the form and enter the text `Insect`.
 - e. Click the **Finish** button.
8. Create the `Insect` struct:

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

9. Below the struct declaration, create an extension with two `static` variables. The first variable, named `oneInsect`, will be used for previews. The second variable, named `manyInsects`, will be used in our app. For this variable, we will read the data from the `insectData.json` file and decode it into an array of `Insect` instances:

```
extension Insect {  
    static var oneInsect = Insect(  
        id: 1,  
        imageName: "grasshopper",  
        name: "grass",  
        habitat: "pond",  
        description: "long description here"  
    )  
    static var manyInsects: [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 ?? [oneInsect]  
    }()  
}
```

10. With the `Models` folder still selected, create the `InsectData` type. We'll use this type to store an array of `Insect` instances 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**.

11. Create an `Observable` object with one variable to store an array of insects, initialized with the static property `manyInsects`, which contains the data decoded from the JSON file:

```
@Observable class InsectData {  
    var insects = Insect.manyInsects  
}
```

12. Now we will work on the views. First, let's drag and drop `ContentView` to the `Views` folder.
13. The rest of the views we are going to create should be placed within the `Views` folder. Let's create a new 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**.
14. 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)  
        }  
    }  
}
```

15. To preview the design, pass the `oneInsect` static variable to the `InsectCellView` initializer inside the `#Preview` macro:

```
#Preview {  
    InsectCellView(insect: .oneInsect)  
}
```

The result should be similar to the following figure:



Figure 16.1: InsectCellView preview

16. 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` view should display the attributes of the insect:

```
struct InsectDetailView: View {
    var insect: Insect
    var body: some View {
        VStack(alignment: .leading) {
            Text(insect.name)
                .font(.largeTitle)
            Image(insect.imageName)
                .resizable()
                .aspectRatio(contentMode: .fit)
            Text("Habitat")
                .font(.title)
```

```
        Text(insect.habitat)
        Text("Description")
            .font(.title)
        Text(insect.description)
    }
    .padding(.horizontal)
}
}
```

17. Preview the design by passing the `oneInsect` static variable to the `InsectDetailView` initializer in the `#Preview` macro:

```
#Preview {
    InsectDetailView(insect: .oneInsect)
}
```

The `InsectDetailView` preview should look as follows:

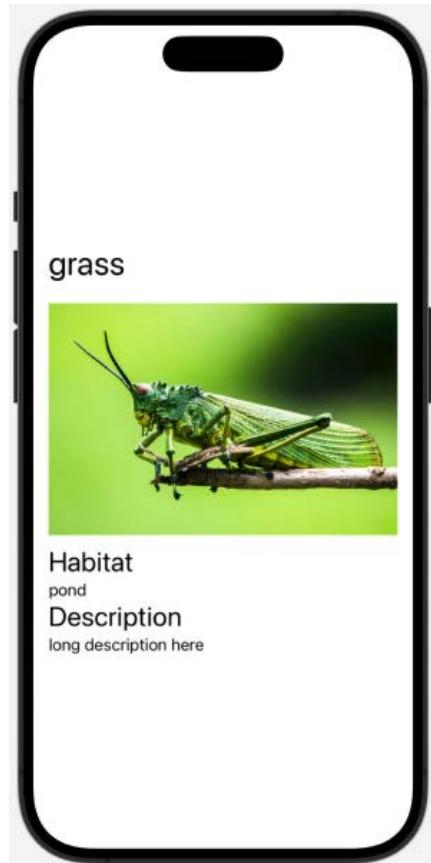


Figure 16.2: `InsectDetailView` preview

18. Now create a SwiftUI view called `InsectListView`.
19. Above the `body` variable of our `InsectListView` view, declare and initialize our `@Environment` variable that contains the insect data as an instance of the `InsectData` class:

```
    @Environment(InsectData.self) private var insectData: InsectData
```

20. Within the `InsectListView` `body`, implement a `List` view that iterates over the data and displays it using instances of the `InsectCellView` view. Each cell will be in a `NavigationLink` and its navigation destination will be an instance of the `InsectDetailView` view:

```
var body: some View {
    List{
        ForEach(insectData.insects){insect in
            NavigationLink(value: insect) {
                InsectCellView(insect: insect)
            }
        }
    }
    .navigationDestination(for: Insect.self) { insect in
        InsectDetailView(insect: insect)
    }
    .navigationTitle("Insects")
}
```

21. To preview the design, add the `.environment()` modifier to the `InsectListView` initializer call in the `#Preview` macro, and embed the `InsectListView` in a `NavigationStack`:

```
#Preview {
    NavigationStack {
        InsectListView()
            .environment(InsectData())
    }
}
```

The InsectListView preview should look as follows:

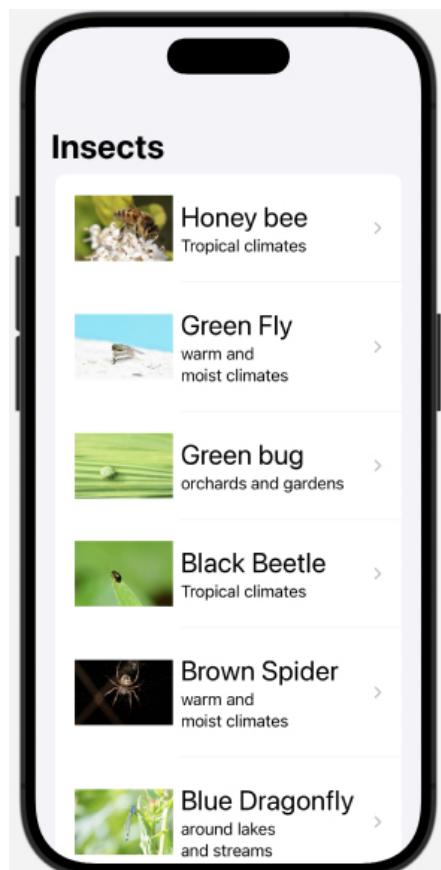


Figure 16.3: InsectListView preview

22. Using the live preview of the Xcode canvas, tap on one insect cell to reveal the details on the screen, as shown in the following figure:



Figure 16.4: InsectDetailView preview for the first insect cell

23. Finally, let's work on the `ContentView` struct, which is the first view displayed by the app. Add the `@Environment` variable above the `body` variable:

```
    @Environment(InsectData.self) private var insectData: InsectData
```

24. Replace the contents of the `body` variable with a `NavigationStack` and the `InsectListView`:

```
var body: some View {
    NavigationStack {
        InsectListView()
    }
}
```

25. Add the `.environment()` modifier to the `#Preview` macro and pass an instance of `InsectData`:

```
#Preview {
    ContentView()
        .environment(InsectData())
}
```

26. We have used the live preview in the canvas to visualize our views. To finalize the app and make it work on the iOS simulator or a real iOS device, we need to initialize the environment from the `App` struct. Let's move to `MultiplatformApp.swift` file and use the `environment` modifier to pass the data to the `ContentView` view:

```
struct MultiplatformApp: App {
    @State private var insectData = InsectData()
    var body: some Scene {
        WindowGroup {
            ContentView()
                .environment(insectData)
        }
    }
}
```

Congrats! You have finished the app, and now you can run it in the iOS simulator of your choice.

How it works...

Our goal in this recipe was to read insect data from a JSON file and display the content in our app. To get started, we inspected the raw JSON data and created a struct to hold the data: that is, the `Insect` struct.

The `Insect` struct implements the `Decodable`, `Identifiable`, and `Hashable` 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 `Hashable` protocol conformance allows us to implement the value-based navigation on `Insect` structs.

The `InsectData.swift` file is the most important component of this recipe. It is used to define an `Observable` type called `InsectData`, which will store our model data in a variable named `insects`. The `@Observable` macro provides automatic conformance to our class to the `Observable` protocol. Thanks to this protocol conformance, SwiftUI will trigger a refresh of any view reading the `insect` variable when its value changes. To share the instance of `InsectData` with the view hierarchy, we used the SwiftUI environment.

Since we are building a multiplatform application, we have organized the code in different groups so it will be easier to share files across different platforms later on. Our view hierarchy is composed of smaller views, which handle different pieces of the UI.

ContentView provides a NavigationStack that embeds InsectListView. InsectListView iterates through our insect data and displays the content in a list where each row is an InsectCellView view. InsectCellView takes an Insect instance and displays its data in a summarized view. Finally, when a cell in the list of insects is tapped, the NavigationLink provides the Insect data and navigates to the destination of an InsectDetailView that contains the full information of the selected insect.

Creating the macOS version of the app with a new target

Our iOS app 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 the iPhone screen. However, a Mac 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.

In this recipe, we will create a new target for our macOS app and share code and resources between the two targets, iOS and macOS. This approach is best suited if one of the following applies:

- Your app has a considerably different user interface between the Mac and the iPhone.
- Your app has a large legacy code base in UIKit and AppKit.
- You have two separate apps and want to share code and resources between them.
- You need to support versions of iOS older than iOS 16 and you can't use the new multiplatform views included in the latest version of SwiftUI.



The purpose of this chapter is to use the legacy approach to multiplatform apps, which should only be followed in the cases explained above. If you start a new project from scratch and plan to target multiple platforms, the best approach is to use SwiftUI and a multiplatform target. Feel free to skip this recipe, as it is only included for legacy reasons, and go straight to the *Creating a multiplatform version of the app sharing the same target* recipe, where the modern approach to multiplatform apps is explained in detail. This recipe shows the legacy approach to adding a separate target for the macOS app. The iOS app uses the Observation framework, which is only available for iOS 17 and macOS Sonoma. The purpose of the recipe is to show how to work with two different targets, but if we need to support legacy code, we should use Combine publishers instead of the Observation framework. If you need a refresher on these topics, check *Chapter 10, Driving SwiftUI with Data*.

Getting ready

Download the chapter materials from GitHub:

[Open the Starter folder and double-click on Multiplatform.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.](https://github.com/PacktPublishing/SwiftUI-Cookbook-3rd-Edition/tree/main/Chapter16-Multiplatform-SwiftUI/02>Create-the-MacOS-legacy-version</p></div><div data-bbox=)

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, as shown in the following figure:

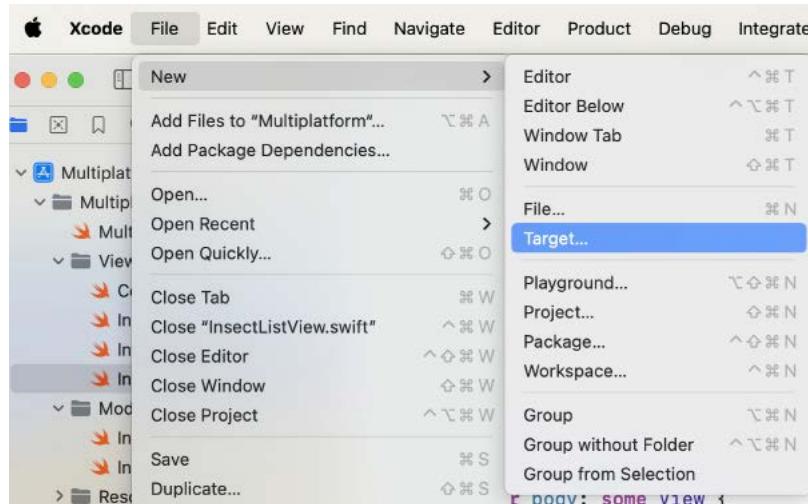


Figure 16.5: Creating a new target

2. Choose the macOS template, scroll down, select App, then click Next, as shown in the figure below:

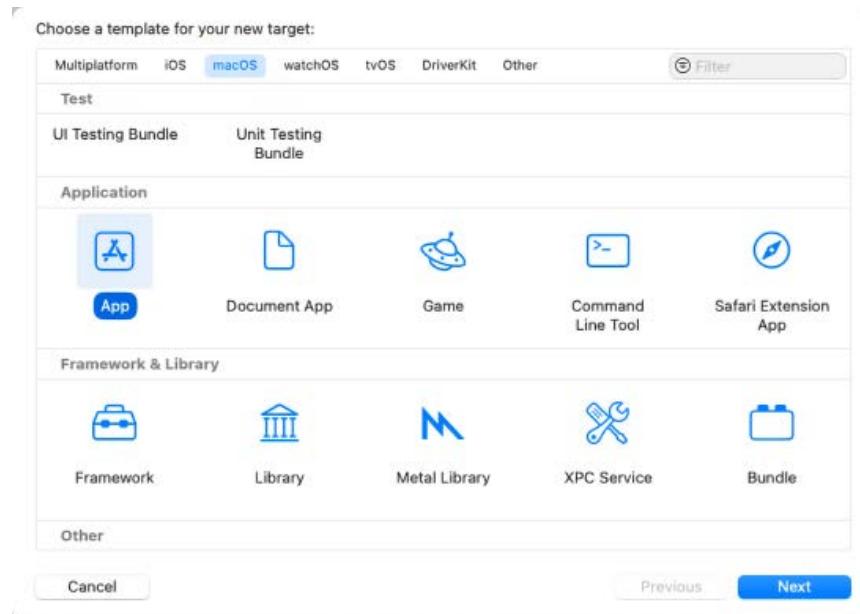


Figure 16.6: Selecting macOS App

3. In the next screen, enter the product name, `macOSMultiplatform`.
4. To be able to preview and run the macOS applications, let's set the Xcode active scheme to `macOSMultiplatform`:

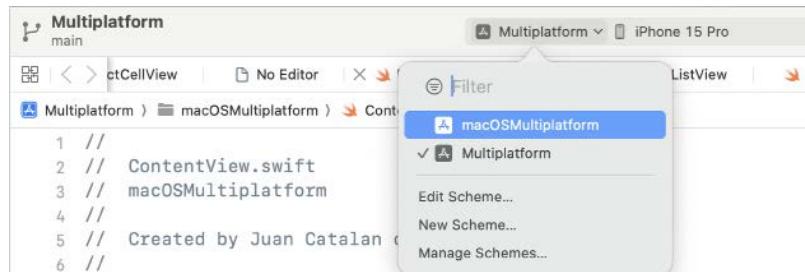


Figure 16.7: Changing the Xcode scheme to `macOSMultiplatform`

5. Select the `ContentView.swift` file in the `macOSMultiplatform` folder. Use the live preview of the Xcode canvas to visualize the view. It should look as follows:

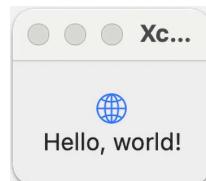


Figure 16.8: macOS default `ContentView` preview

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. It should look as follows:



Figure 16.9: Navigator and Inspector panes in Xcode

7. Let's share files across multiple targets. Select the `Insect.swift`, `InsectData.swift`, `insectData.json`, `InsectDetailView.swift`, and `Assets.xcassets` files. To select multiple files, hold the *Command* key and click on the files listed.

8. In the Inspector pane, check the **macOSMultiplatform** checkbox:

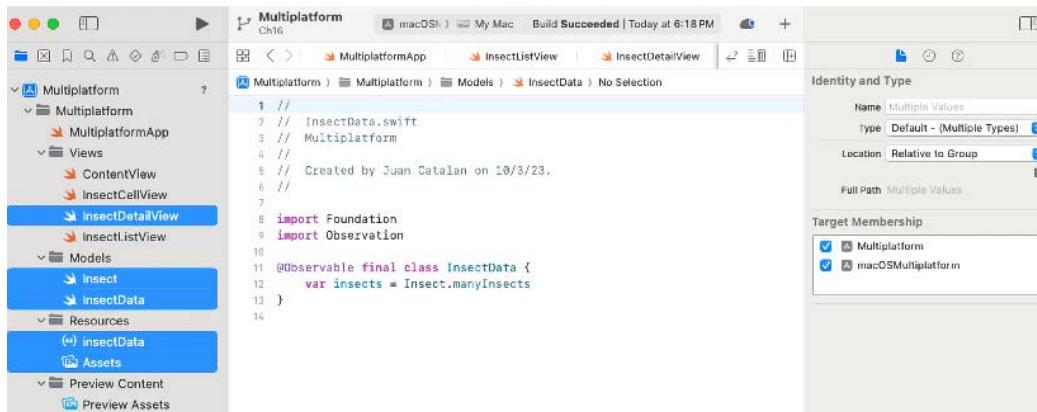


Figure 16.10: Making iOS files accessible in the macOS target

9. Create a new **MacInsectCellView** SwiftUI view in the **macOSMultiplatform** folder. Make sure the right platform target is selected (**macOSMultiplatform**), as shown in the following figure:

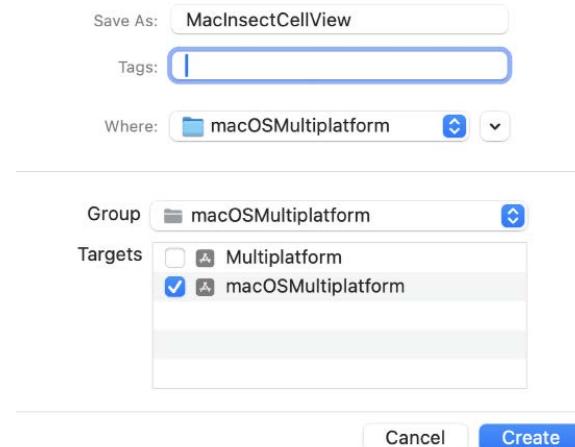


Figure 16.11: Creating a new macOS SwiftUI view and selecting the right target

10. The content of **MacInsectCellView** is like its counterpart **InsectCellView** in the iOS app. The only difference is the size of the frame. Replace the file with the following content:

```

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(.title)
        Text(insect.habitat)
    }
    .padding(.vertical)
}
}
}

```

- Fix the Xcode error by passing the `oneInsect` static variable to the view initializer in the pre-view macro:

```

#Preview {
    MacInsectCellView(insect: .oneInsect)
}

```

The preview should look as follows:



Figure 16.12: `MacInsectCellView` preview

- Create `MacInsectListView` to show a list of items. Make sure the `macOSMultiplatform` target is selected.
- Add the `@Environment` variable above the `body` variable of the `MacInsectListView` struct:

```

@Environment(InsectData.self) private var insectData: InsectData

```

- Replace the content of the `body` variable with a `List` view. The `List` view should display the contents of our `insectData` array obtained from our environment variable. Add the `.listStyle()` modifier that presents the list using a `.sidebar` style:

```

var body: some View {
    List(insectData.insects) { insect in
        NavigationLink(value: insect) {

```

```
        MacInsectCellView(insect: insect)
    }
}
.navigationDestination(for: Insect.self) { insect in
    ScrollView {
        InsectDetailView(insect: insect)
    }
}
.navigationBarTitle("Insects")
.listStyle(.sidebar)
}
```

15. Let's preview the design by passing in a constant binding and adding the `.environment()` modifier to our `MacInsectListView()` instance. In the preview macro, we will also enclose the view in a `NavigationStack`:

```
#Preview {
    NavigationStack {
        MacInsectListView()
            .environment(InsectData())
    }
}
```

The preview should look as follows:

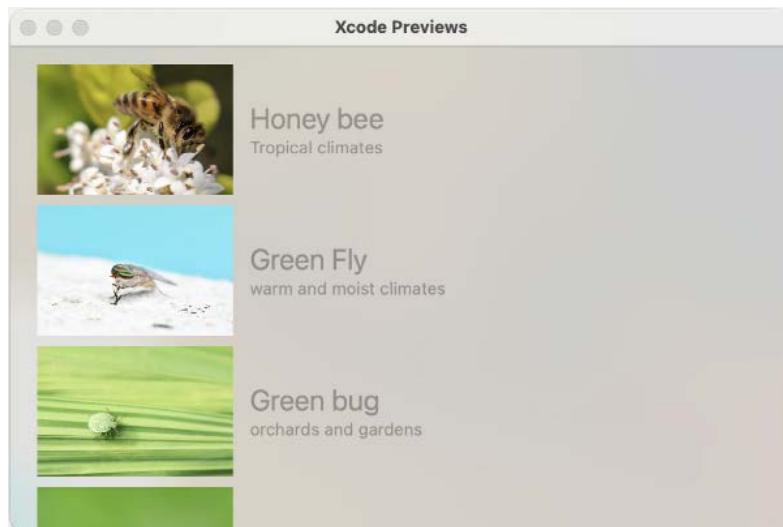


Figure 16.13: `MacInsectListView` preview

16. At this point, the `List` view is already interactive. Use the interactive preview to click on a cell and see the details:

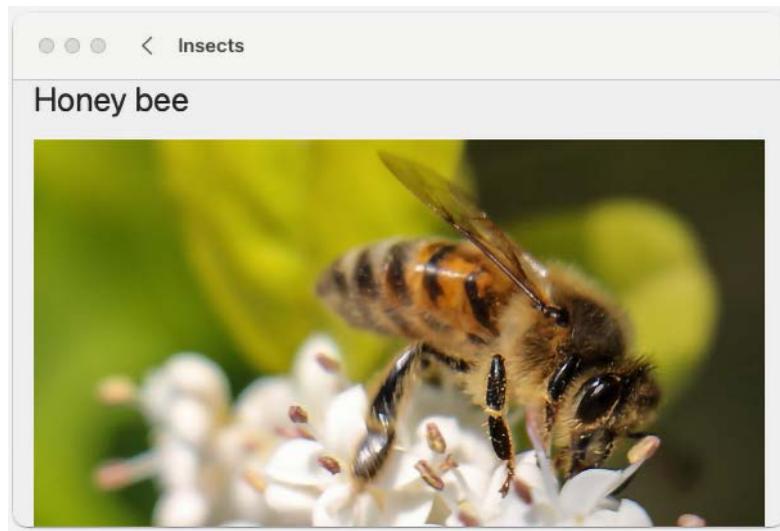


Figure 16.14: `InsectDetailView` preview for the first cell

17. If we try to run the Mac app, it will show the default content view provided by the Xcode template. We need to work on the `ContentView` to call our `MacInsectListView` and pass the data from the SwiftUI environment. Open the `ContentView.swift` file located in the `macOSMultiplatform` folder. Add an `@Environment` variable to hold the insect data we'll be displaying:

```
@Environment(InsectData.self) private var insectData: InsectData
```

18. Replace the body with a `NavigationSplitView` having a two-column layout, with a sidebar view and a detail view side by side. The sidebar view will be `MacInsectListView` and the detail view will be `InsectDetailView`. Also, set the size of the sidebar by adding a `.navigationSplitViewColumnWidth(min:ideal:max:)` modifier. For the detail view, we will display a `ContentUnavailableView` view. Once the user selects an insect, `NavigationSplitView` will take care of replacing the detail view with an instance of `InsectDetailView`:

```
var body: some View {
    NavigationSplitView {
        MacInsectListView()
            .navigationSplitViewColumnWidth(min: 360, ideal: 400, max:
500)
        } detail: {
        ContentUnavailableView(
            "Choose an insect from the list",
            systemImage: "hand.point.up.left"
        )
    }
}
```

```
        )  
    }  
}
```

19. To view the design in the canvas live preview, in the preview macro, add an `.environment` modifier and pass the insect data:

```
#Preview {  
    ContentView()  
        .environment(InsectData())  
}
```

The resulting `ContentView` preview should look as follows:

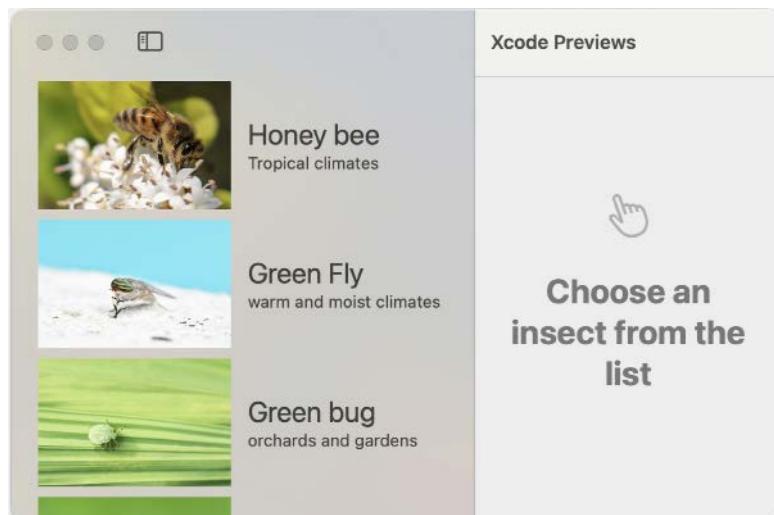


Figure 16.15: `ContentView` preview

20. Finally, let's modify the `App` struct to finish our Mac app. Switch to the `macOSMultiPlatformApp.swift` file in our `macOSMultiplatform` folder and modify the struct, which should be like the following code:

```
struct macOSMultiplatformApp: App {  
    @State private var insectData = InsectData()  
    var body: some Scene {  
        WindowGroup {  
            ContentView()  
                .environment(insectData)  
        }  
    }  
}
```

21. Run the app on your Mac and play with it. You have a macOS app built entirely with SwiftUI! The result should be like *Figure 16.16*:

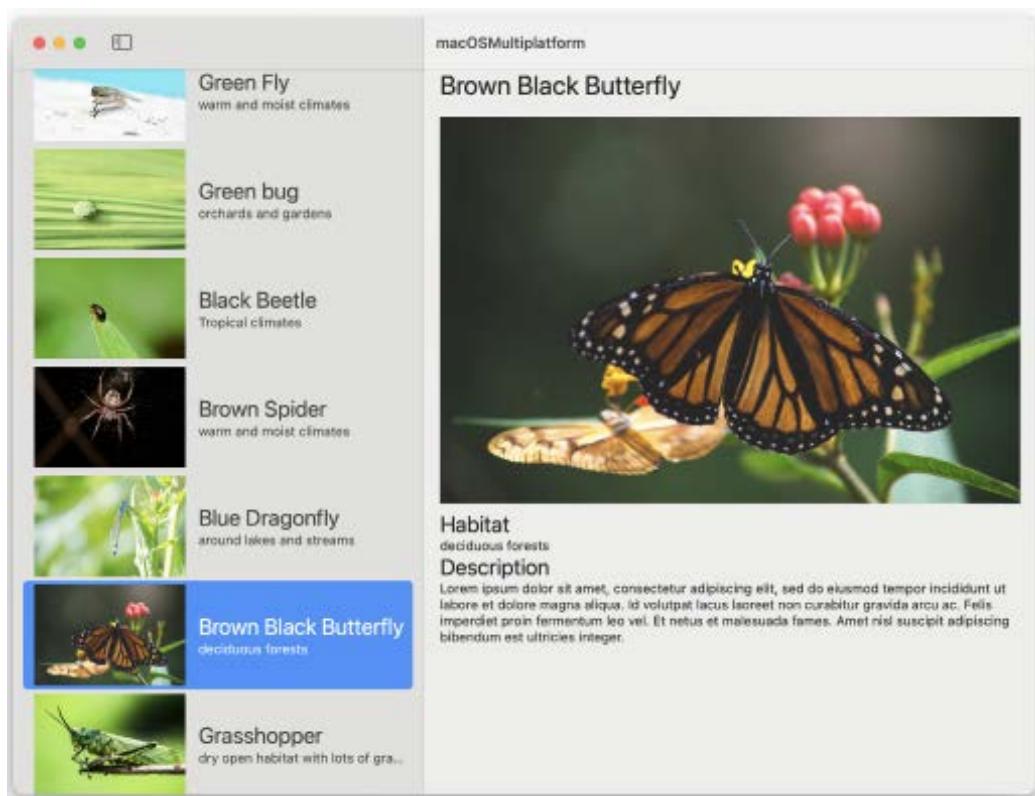


Figure 16.16: Fully functional macOS app

How it works...

When using SwiftUI code for more than one platform, 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. Therefore, we 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 `macOSMultiplatform` section without changing the scheme first, we'll get an error.

To use certain iOS files in macOS in Xcode, we select the corresponding files from the Navigator pane and check 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 improve the UI for the macOS app, we used a `NavigationView` with a two-column layout. For the sidebar, we used our `List` view with a `.sidebar` style, and for the detail view, we used `InsectDetailView`, which is the same view used in the iOS app. When the app runs for the first time, the user has not selected an insect yet. For this specific case, we used a `ContentUnavailableView` to display a message prompting the user to select an insect.

Creating a multiplatform version of the app sharing the same target

When you create a new app using Xcode 15, a multiplatform target is used by default. The multiplatform target allows us to share a single target for iOS, iPadOS, macOS, visionOS, and tvOS, while the watchOS app remains in a separate target.

SwiftUI is in its fifth iteration and Apple is making a big push to use SwiftUI as the preferred framework for multiplatform apps. Many SwiftUI views behave differently depending on the platform used to run the app. SwiftUI takes care of the configuration automatically, while also providing configuration options to further refine the platform-specific behavior.

This recipe shows how to create an app for iOS, iPadOS, and macOS, and best practices when targeting multiplatform apps in SwiftUI.

Getting ready

Download the chapter materials from GitHub:

<https://github.com/PacktPublishing/SwiftUI-Cookbook-3rd-Edition/tree/main/Chapter16-Multiplatform-SwiftUI/03-Create-a-multiplatform-app>

Open the `Starter` folder and double-click on `Multiplatform.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 that recipe.

How to do it...

If we want to target macOS with SwiftUI, it is better to target iPadOS first and make the UI look good on the iPad. Once the app looks native to the iPad, use the iPad version of the app as a starting point for the macOS app. The good news is that since iOS and iPadOS share many APIs, the SwiftUI views for the iPad already work for the iPhone too.

By default, the multiplatform target in the iOS project already supports the iPad and the Mac, for Mac computers with Apple Silicon. The support for an iOS/iPadOS app in a Mac with Apple Silicon is known as *Designed for iPhone* or *Designed for iPad*.

To see how our single target supports multiple platforms at the same time, select the target in Xcode, and in its **General** tab, you'll see the three platforms supported under **Supported Destinations**, as shown in the following screenshot:

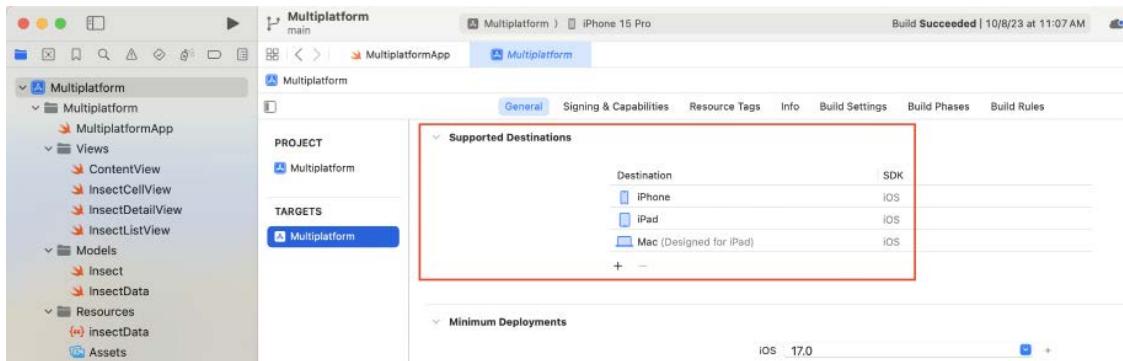


Figure 16.17: Supported destinations

If you choose an iPad as the preview device on the canvas, you'll see that the UI looks like the iPhone UI but much bigger. You'll notice a lot of wasted space, and this is not the best user experience for iPad users. We want a native look and feel for the iPad, so let's work on changing our code to make the UI look better on the iPad first. In our case, in `ContentView`, replacing `NavigationStack` with `NavigationSplitView` gives us a different UI for the iPad and the iPhone using a single code base, and this is exactly what we want. The steps are as follows:

1. Since the iPad is going to have a split user interface, with a list of insects and the details of one insect, we need to account for the initial case when no insect has been selected. In this case, the detail view will be empty, and we will use a `ContentUnavailableView` to prompt the user to select an insect. Go to `ContentView.swift` and replace the content of the `body` variable with a `NavigationSplitView` with a two-column layout:

```
var body: some View {
    NavigationSplitView {
        InsectListView()
    } detail: {
        ContentUnavailableView(
            "Choose an insect from the list",
            systemImage: "hand.point.up.left"
        )
    }
}
```

2. Let's make another modification to the app. Since we need to adapt to different screen sizes, we will use a scroll view in our detail view. Switch to the `InsectDetailView` and, in the `body` variable, enclose the `VStack` in a `ScrollView`. At the same time, remove the first `Text` view and its modifiers, used to display the name of the insect, and use a navigation title for this purpose. This change is possible because we know that the detail view is going to be embedded in a navigation split view. The code for the `body` variable should look as follows:

```
var body: some View {
    ScrollView {
        VStack(alignment: .leading) {
            Image(insect.imageName)
                .resizable()
                .aspectRatio(contentMode: .fit)
            Text("Habitat")
                .font(.title)
            Text(insect.habitat)
            Text("Description")
                .font(.title)
                .padding()
            Text(insect.description)
        }
        .padding(.horizontal)
    }
    .navigationTitle(insect.name)
}
```

3. Modify the preview macro to embed the view in a navigation stack so that we can see the navigation title in the preview section of the canvas:

```
#Preview {
    NavigationStack {
        InsectDetailView(insect: .oneInsect)
    }
}
```

4. Now go back to `ContentView`, choose an iPad in the preview canvas, and switch to a landscape orientation with the `device settings` button. Make sure the live preview is selected and once the preview appears, choose the first insect in the sidebar.

If everything goes well, the preview should look as follows:

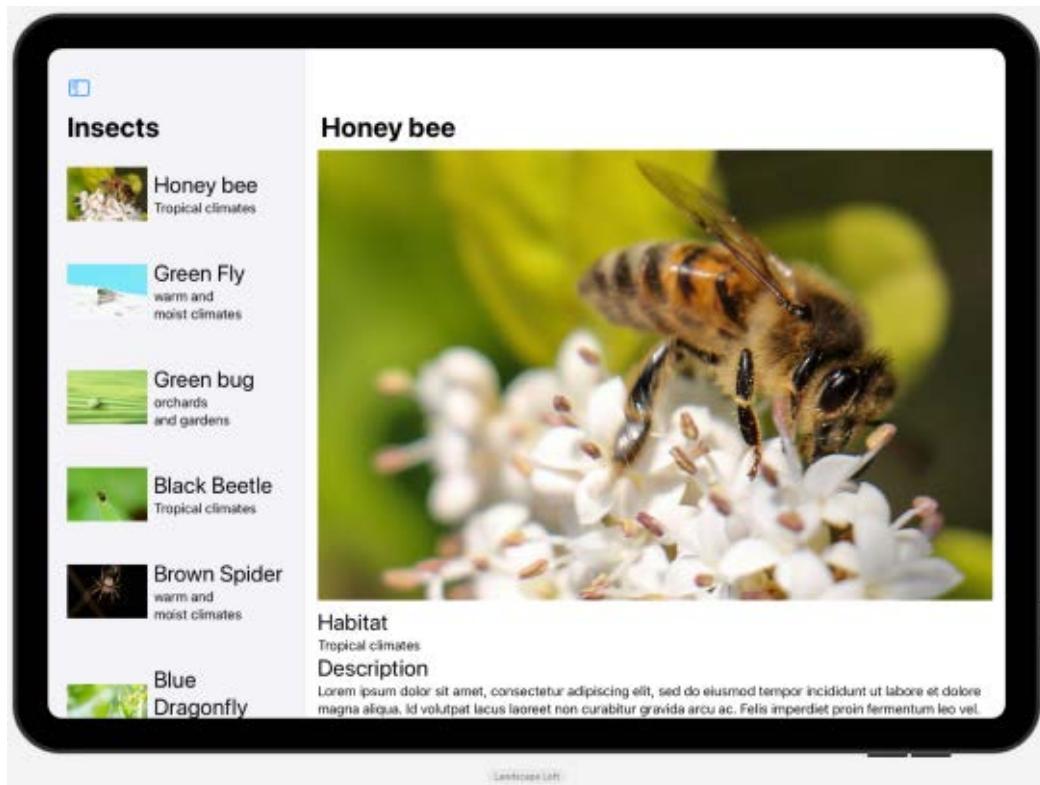


Figure 16.18: ContentView preview on the iPad

- At this point, we have an app that works on two different devices. We just saw the iPad preview and we already know that the iPhone works too. Without any additional modifications to our code, we can run the app on a Mac with Apple Silicon. If this is your case, you can give it a try. In Xcode, choose the **Mac (Designed for iPad)** destination, and click the play button to start the app from Xcode, as shown in the following screenshot:



Figure 16.19: Selecting the destination for our target

6. Once the Mac app starts, you'll notice that it behaves in a similar way to the app on the iPad. If you select the first cell on the list, the user interface should look similar to the following screenshot:

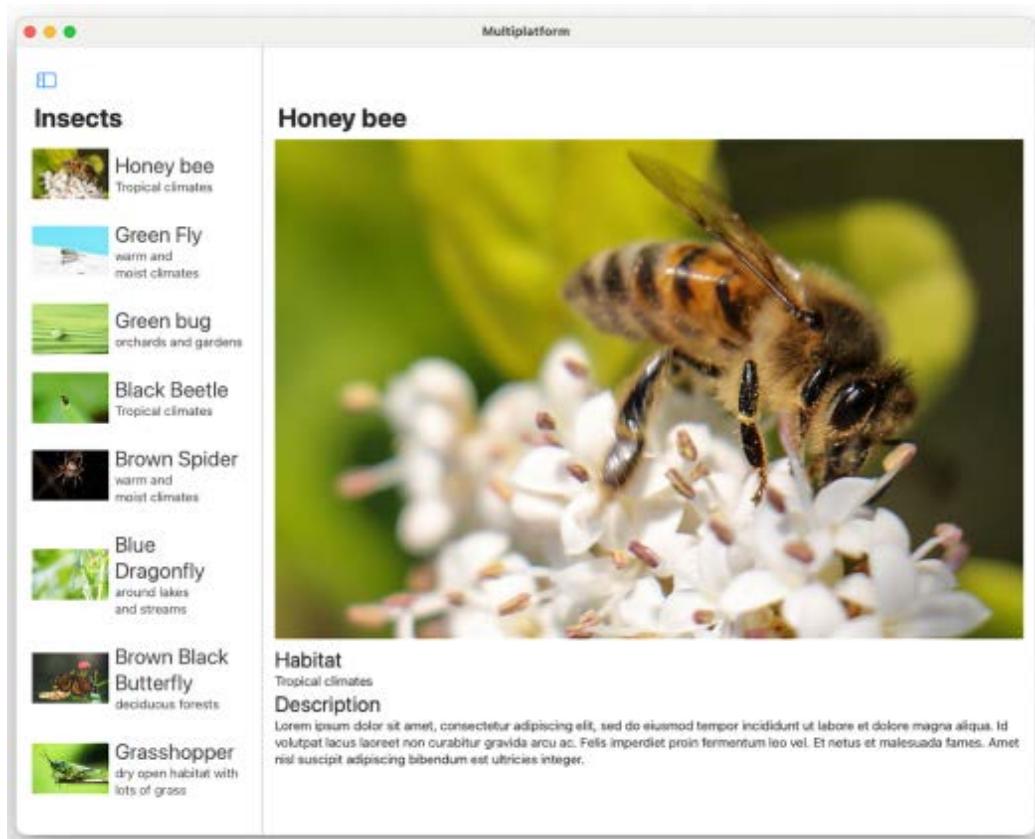


Figure 16.20: Mac (Designed for iPad) app

7. If you don't have a Mac with Apple Silicon, you can't run the Mac (Designed for iPad) app. So, let's work now on creating a destination native to macOS, to reach all Mac users and not only the ones with newer Macs. Select the target in Xcode, and in its General tab, you'll see **Supported Destinations**.

Click on the plus button under the three destinations, and since we are using SwiftUI, choose the **Mac** option instead of Mac Catalyst, as seen in the following screenshot:

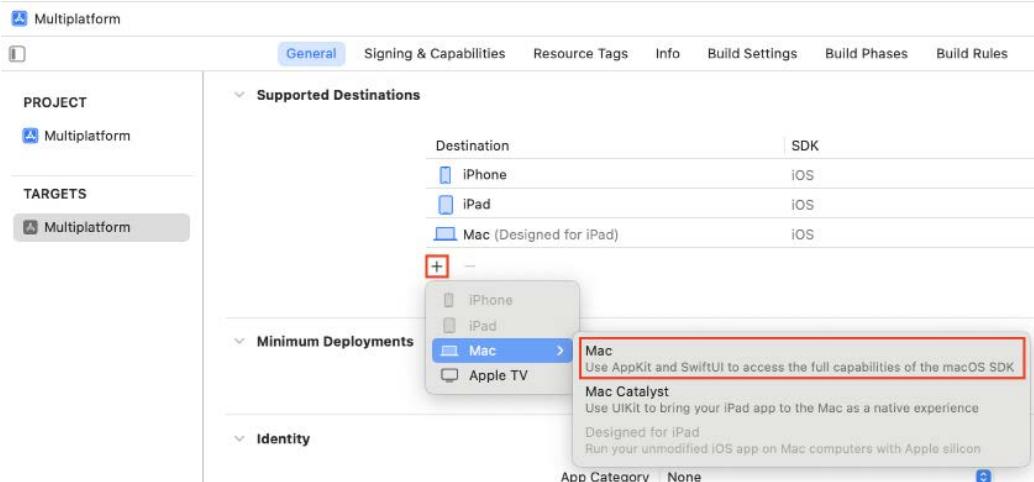


Figure 16.21: Adding a macOS-native destination

- After clicking on the **Mac** option, Xcode will show a modal window asking to confirm our action. Click on the **Enable** button, as in the following screenshot:

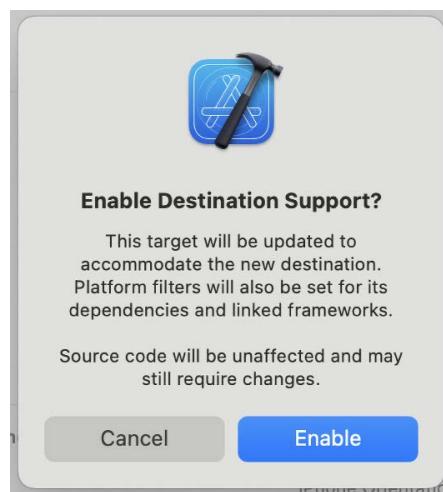


Figure 16.22: Enabling a macOS-native destination

9. Now, if we look at the destinations, we will notice we have four destinations and not three. Interestingly, the Mac has two destinations. The first one, which is the one we recently added, uses the macOS SDK and runs on all Macs natively. The other one is **Mac (Designed for iPad)**, which uses the iOS SDK and only runs on Macs with Apple Silicon, as we've seen before:

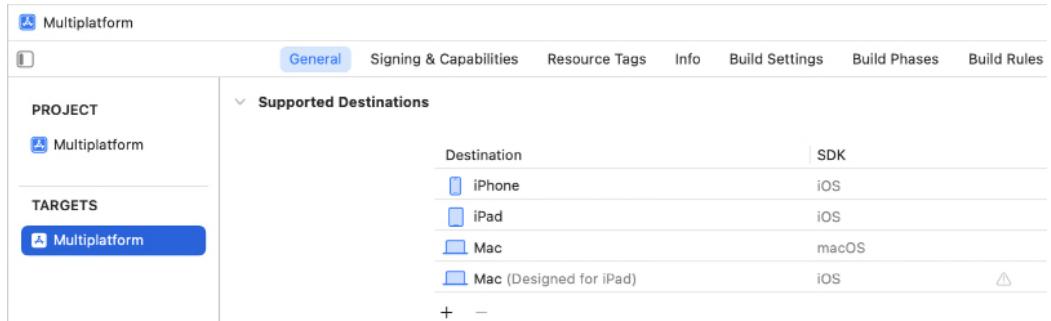


Figure 16.23: Available destinations showing the native macOS

10. If we click on the warning icon in the **Mac (Designed for iPad)** destination, we will have the option of removing the target since now we have a native macOS destination. Apple recommends testing the native macOS app and removing the other Mac destination once we are happy with the results. The following figure illustrates how to delete the target:

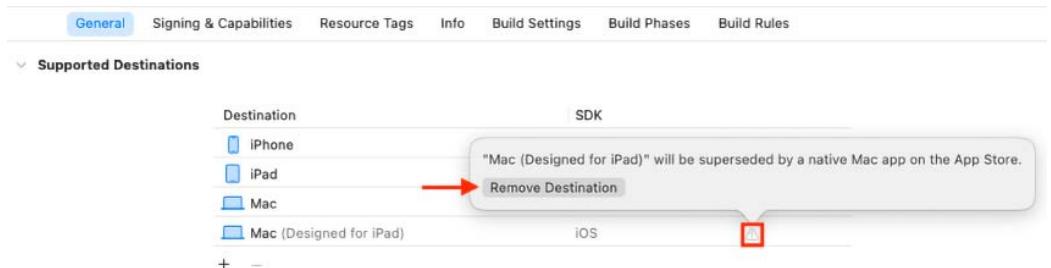


Figure 16.24: Mac (Designed for iPad) warning message

11. It is time to finally work on the native macOS. Go ahead and choose your Mac as the destination to run the current scheme. Make sure to choose the native Mac destination and not the one marked as **Designed for iPad**.

Click on the drop-down selector in the top bar of Xcode, as in the following screenshot:

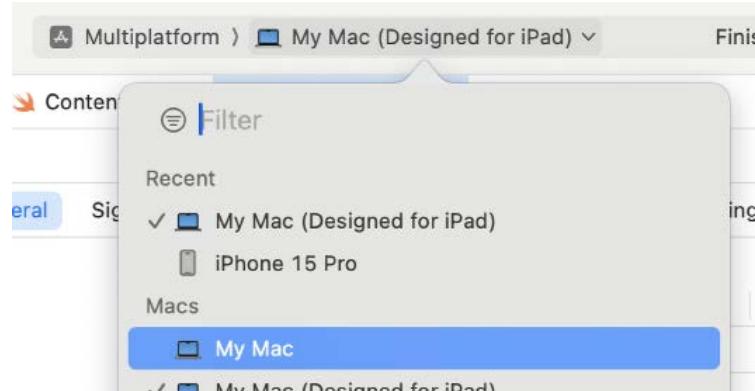


Figure 16.25: Choosing the native Mac destination

12. Now, with the native Mac destination, build and run the app with Xcode. You have a macOS app built with SwiftUI and sharing code with the iOS and iPad apps! Notice how the app looks more macOS-native than the **Designed for iPad** macOS version:

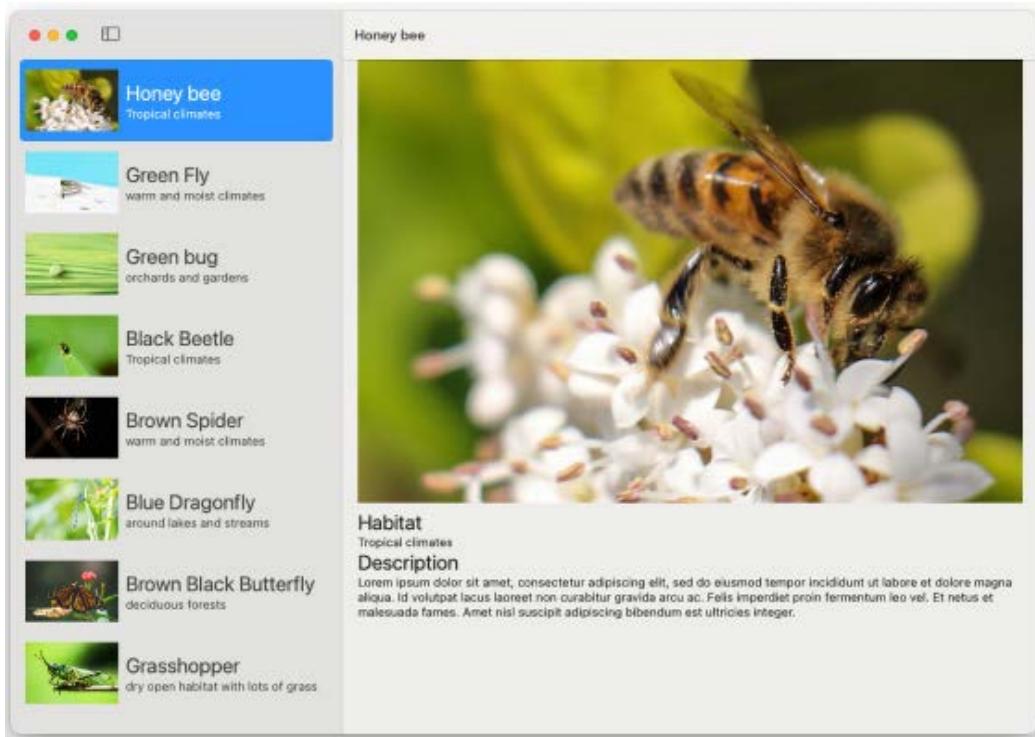


Figure 16.26: Fully functional native macOS app

13. Now let's see how we can improve the macOS app by having a slightly different UI than the iPad. First let's modify the sidebar to hide the toolbar, display a text view before the list of insects, and fix the column width. Since we only want our code for macOS, we will use conditional compiling. Switch to `ContentView.swift` and add some code to the sidebar closure to accomplish what we want.

```
NavigationSplitView {  
    #if os(macOS)  
    Text("Insects")  
        .font(.title2)  
        .toolbar(.hidden, for: .windowToolbar)  
        .navigationSplitViewColumnWidth(min: 240, ideal: 240, max: 480)  
    #endif  
    InsectListView()  
} detail: {  
    ContentUnavailableView(  
        "Choose an insect from the list",  
        systemImage: "hand.point.up.left"  
    )  
}
```

14. Let's modify the insect image in `InsectCellView` so that the macOS app has a different size than the iOS app. Switch to `InsectCellView.swift` and add a view modifier specific to macOS to the image view. The code should be as follows:

```
Image(insect.imageName)  
    .resizable()  
    .aspectRatio(contentMode: .fit)  
    .clipShape(Rectangle())  
    #if os(macOS)  
    .frame(width: 200, height: 160)  
    #else  
    .frame(width: 100, height: 80)  
    #endif
```

15. And finally, let's customize the detailed view for the macOS native app. Go to `InsectDetailView.swift` and modify the `VStack` to have a `Text` view at the top with the insect's name instead of adding the name to the navigation bar. We will use conditional compiling to accomplish this. After the modifications, the code should look like the following:

```
var body: some View {  
    VStack(alignment: .leading) {  
        #if os(macOS)  
        Text(insect.name)
```

```
        .font(.largeTitle)
        .padding(.horizontal)
#endif
ScrollView {
    VStack(alignment: .leading) {
        Image(insect.imageName)
            .resizable()
            .aspectRatio(contentMode: .fit)
        Text("Habitat")
            .font(.title)
        Text(insect.habitat)
        Text("Description")
            .font(.title)
        Text(insect.description)
    }
    .padding(.horizontal)
}
#endif
.navigationTitle(insect.name)
#endif
}
```

16. Run the app on your Mac and you'll see our changes come to life. If you choose the first cell, your app should look similar to the following screenshot:

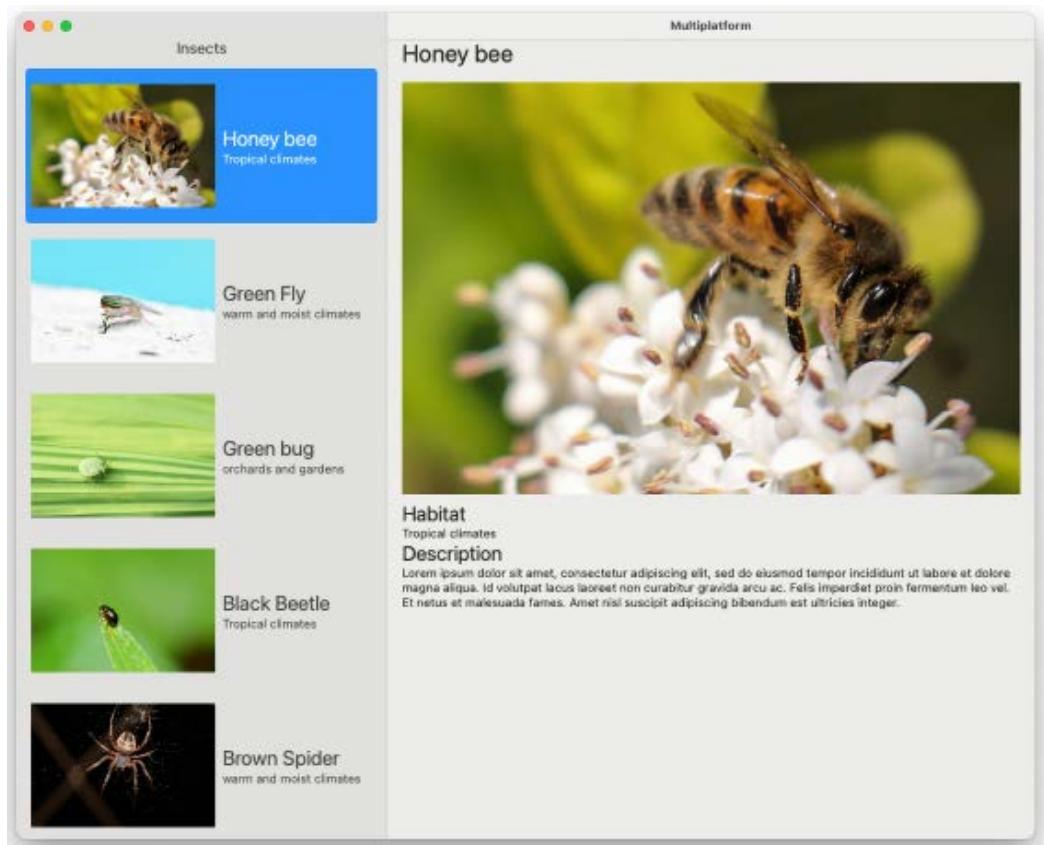


Figure 16.27: macOS-native app with a different UI

How it works...

Apple has made it very easy for iOS developers to build apps for the Mac. First, with the introduction of Catalyst, UIKit iPad apps could be converted to macOS apps. Then, with the introduction of the Mac with Apple Silicon, iOS apps were made available to macOS users. And now, with SwiftUI and Xcode's multiplatform target, we can develop native macOS apps with a little effort.

The code in this recipe is self-explanatory. We showed how to change an iOS app so that we can create a native macOS app without any effort. Our first step was to create a custom UI for the iPad and work on the iPad version of the app. For our purpose, we replaced our `NavigationStack` with a `NavigationView` with a two-column layout. Due to the new list-detail layout, we had to handle the initial case where no insect was selected, by using a `ContentUnavailableView` to display a label with a message, instead of an empty view. We also tried the iPad app in the live preview and ran the macOS (Designed for iPad) version of the app. We added a new destination to our multiplatform target to implement a native macOS app. Once we had the native macOS app, we used conditional compiling to add specific code for macOS and iOS.

There's more...

But now that we have on target, how can we have a different file for macOS or iOS? Xcode 15 provides a way to achieve this. Select the target and switch to the **Build Phases** tab. In the **Compile Sources** section, we have a list of all the source files included in the compilation. Next to the **Name** column, there is a **Filters** column. For each source file, you can click on the filter icon and reveal a popup, where you can select which platforms use the source file. The following figure illustrates how to use it:

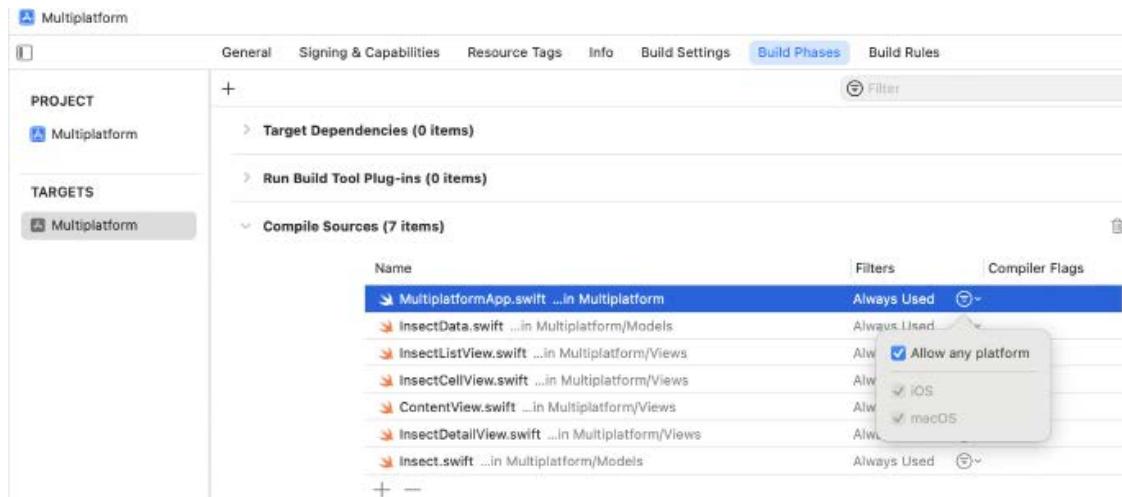


Figure 16.28: Filtering a source file per platform

Creating the watchOS version of the iOS app

In the previous recipes, we created an iOS app that works on the iPhone, the iPad, and Macs with Apple Silicon, and a native macOS version of that app. Now, finally, we'll create the watchOS version of the app. Even with the multiplatform target we used in the previous recipe, the watchOS app needs a separate target.

When creating apps for watchOS, we need to keep in mind that the screen is small, we have limited gestures to interact with the UI, and watchOS provides specific UI widgets available to us. We need to make appropriate design choices based on these constraints. The good news is that with WatchOS 10, SwiftUI takes care of the heavy lifting for us.

In this recipe, we will create the watchOS version of our insect app.

Getting ready

The watchOS simulator is not installed with Xcode by default. To install the watchOS 10 simulator, launch Xcode and go to the **Settings** menu. In the modal window, choose the **Platforms** tab. If the watchOS 10.0 simulator icon is blue, it means you already installed the simulator before and you can skip to the next paragraph. Otherwise, the icon will be gray, and a **Get** button will appear on the right side. Click on the **Get** button and wait for the download, verification, and installation.

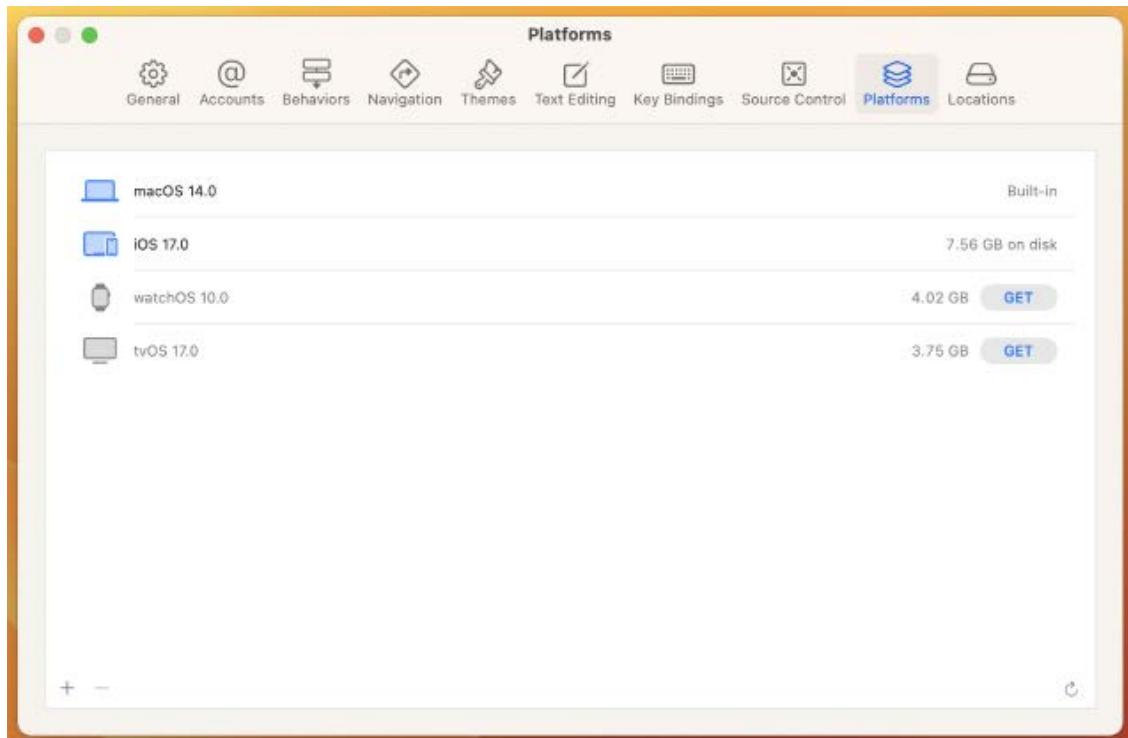


Figure 16.29: Getting the watchOS 10.0 simulator

After making sure the watchOS simulator is installed, download the chapter materials from GitHub:

<https://github.com/PacktPublishing/SwiftUI-Cookbook-3rd-Edition/tree/main/Chapter16-Multiplatform-SwiftUI/04>Create-the-watchOS-version>

Open the **Starter** folder and double-click on **Multiplatform.xcodeproj** to open the app that we built in this chapter's recipe, *Creating a multiplatform version of the app sharing the same target*. 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, as shown in the following screenshot:

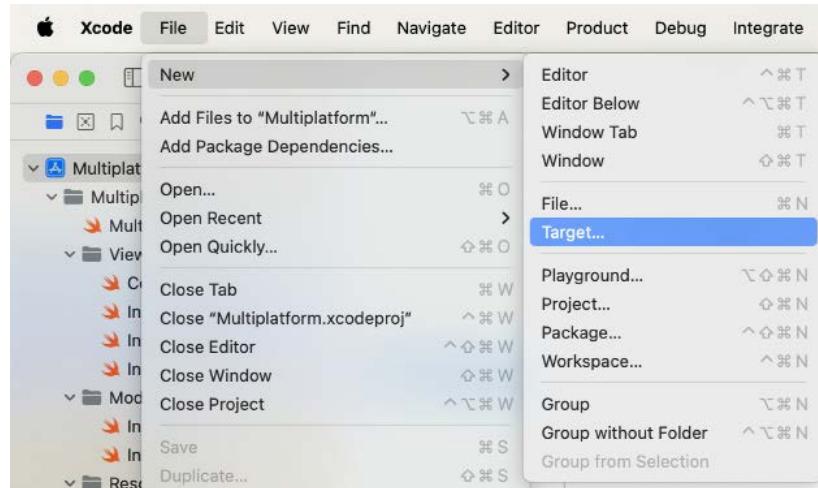


Figure 16.30: Creating a new target

2. Choose the watchOS tab, scroll down, and in the Application section, select App, then click Next, as in the figure below:

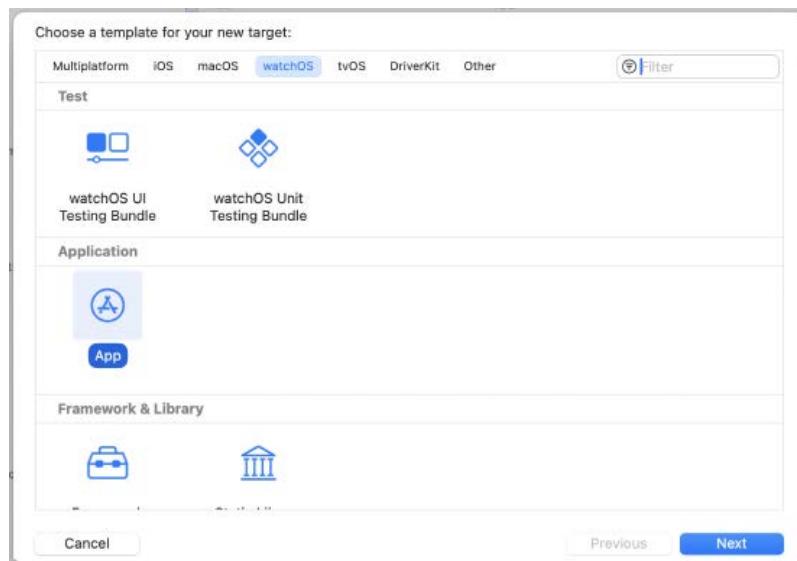


Figure 16.31: Selecting the watchOS app target

3. In the next screen, enter the product name, `watchOSMultiplatform`, and choose **Watch-only App**, as shown in *Figure 16.32*:

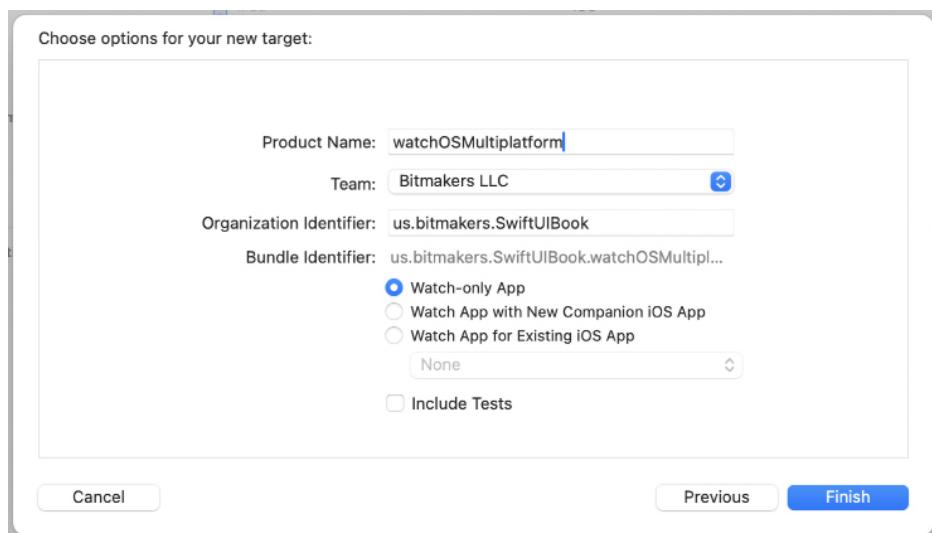


Figure 16.32: Selecting options for the watchOS app target

4. Click **Activate** in the pop-up window that appears:

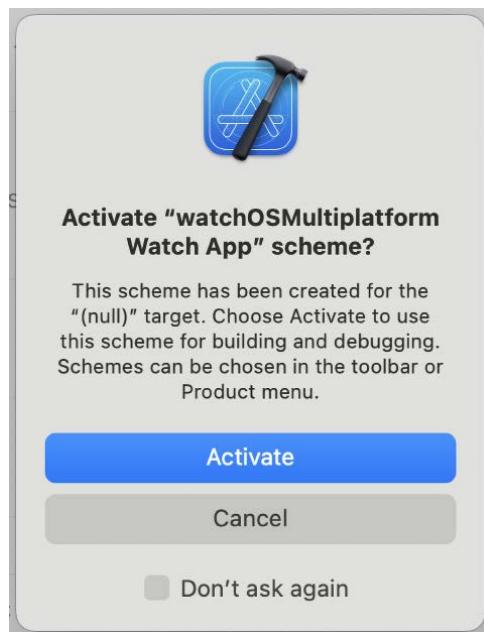


Figure 16.33: watchOS Activate popup

- The watchOS folders should appear in the navigation pane. Select the `ContentView.swift` file in the `watchOSMultiplatform` Extension folder. Preview it in the Xcode canvas. The preview should look as follows:



Figure 16.34: Initial watchOS preview

- Since we will be sharing the asset catalog with the other target, delete the `Assets.xcassets` folder from the `watchOSMultiplatform` Watch App group in the Navigator pane of Xcode.
- 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 16.35: Navigator and Inspector panes in Xcode

- Let's share some files between the multiplatform and watchOS targets. In the navigation pane, select the following files from our `Multiplatform` group: `InsectListView.swift`, `Insect.swift`, `InsectData.swift`, `insectData.json`, and `Assets.xcassets`.
- In the Inspector pane, check the `watchOSMultiPlatform Watch App` checkbox to share the files between the multiplatform and watchOS targets:

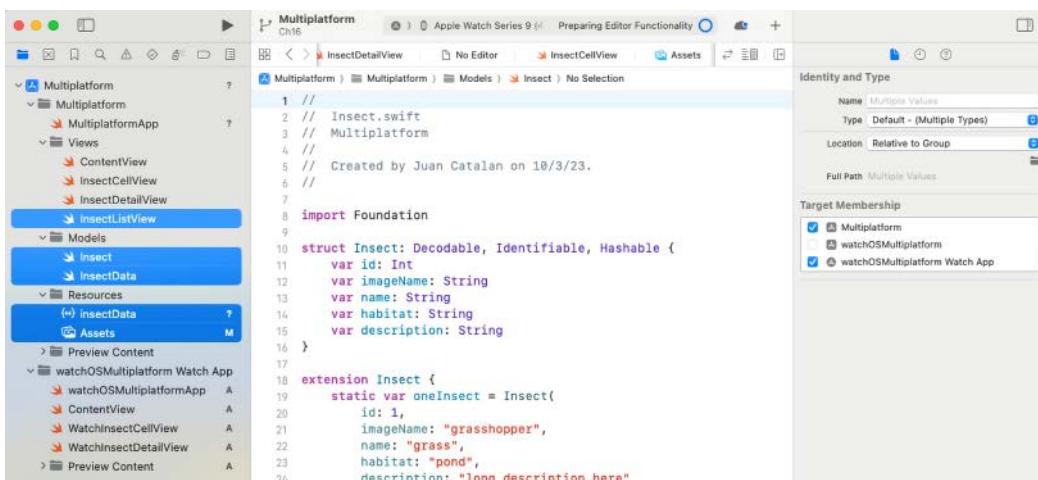


Figure 16.36: Selecting multiple files in Xcode Navigator

10. Create a new `WatchInsectCellView` SwiftUI view in the `watchOSMultiPlatform Watch App` folder. Make sure the right platform target is selected (`watchOSMultiplatform Watch App`):

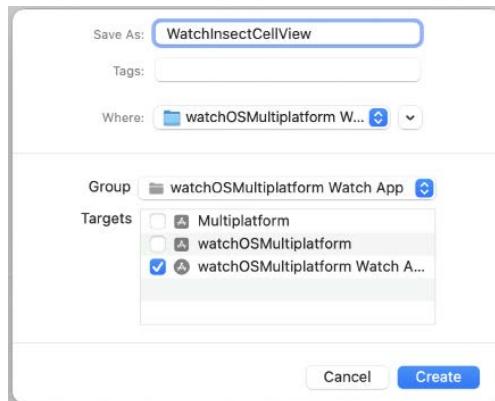


Figure 16.37: Creating a new view and selecting the watchOS target

11. Following the same instructions, create a new `WatchInsectDetailView` SwiftUI view in the `watchOSMultiPlatform Watch App` folder, and assign the `watchOSMultiplatform Watch App` target.
12. We are going to base the code for `WatchInsectCellView` on `InsectCellView` but, due to the smaller size of the screen of the watch, we will modify the layout. Go to `WatchInsectCellView.swift` and modify the content with the following code:

```
struct WatchInsectCellView: View {
    var insect: Insect
    var body: some View {
        VStack(alignment: .leading) {
            Image(insect.imageName)
                .resizable()
                .aspectRatio(contentMode: .fit)
            Text(insect.name)
                .font(.title2)
            Text(insect.habitat)
        }
    }
}
```

13. Modify the preview macro to embed the view in a `List` view and to use the static variable `oneInsect`:

```
#Preview {
    List {
        WatchInsectCellView(insect: .oneInsect)
```

```
    }
}
```

14. Now, `WatchInsectDetailView` will be based on `InsectDetailView` with changes to make it look better on a watch screen. Go to `WatchInsectDetailView.swift` and replace its content with the following code:

```
struct WatchInsectDetailView: View {
    var insect: Insect
    var body: some View {
        VStack(alignment: .leading) {
            ScrollView {
                VStack(alignment: .leading) {
                    Image(insect.imageName)
                        .resizable()
                        .aspectRatio(contentMode: .fit)
                    Text("Habitat")
                        .foregroundStyle(Color.accentColor)
                    Text(insect.habitat)
                    Text("Description")
                        .foregroundStyle(Color.accentColor)
                    Text(insect.description)
                }
                .padding(.horizontal)
            }
            .navigationTitle(insect.name)
        }
    }
}
```

15. Modify the preview macro to embed the view in a `NavigationStack` view and to use the static variable `oneInsect`:

```
#Preview {
    NavigationStack {
        WatchInsectDetailView(insect: .oneInsect)
    }
}
```

16. Now let's finish the `List` view for the watchOS target. Go to `InsectListView` and add code for the watchOS version of the app. We will use compiler directives to provide different cell and detail views for the watch app. The modified content should be like the following code:

```
struct InsectListView: View {
```

```
@Environment(InsectData.self) private var insectData: InsectData
var body: some View {
    List {
        ForEach(insectData.insects) {insect in
            NavigationLink(value: insect) {
                #if os(watchOS)
                WatchInsectCellView(insect: insect)
                #else
                InsectCellView(insect: insect)
                #endif
            }
        }
    }
    .navigationDestination(for: Insect.self) { insect in
        #if os(watchOS)
        WatchInsectDetailView(insect: insect)
        #else
        InsectDetailView(insect: insect)
        #endif
    }
    .navigationTitle("Insects")
}
}
```

17. With all these customizations, the `WatchInsectDetailView` we created earlier can now be previewed. Select `WatchInsectDetailView.swift` and open the live canvas preview. The preview should look as follows:



Figure 16.38: `WatchInsectDetailView` preview

18. Let's preview `WatchInsectCellView` too. Switch to `WatchInsectCellView.swift` and open the live canvas preview. The preview should look as follows:



Figure 16.39: `WatchInsectCellView` preview

19. Now, let's open the `ContentView.swift` file located in `watchOSMultiPlatform` Watch App and replace the `body` variable with our `InsectListView` embedded in a `NavigationStack`:

```
struct ContentView: View {
    var body: some View {
        NavigationStack {
            InsectListView()
                .environment(InsectData())
        }
    }
}
```

20. Modify the preview macro to get the `InsectData` from the environment:

```
#Preview {
    ContentView()
        .environment(InsectData())
}
```

21. If everything went well, the app should build and run on a watchOS simulator now. Let's use the live preview on the canvas with `ContentView`. Try scrolling through the list of insects, selecting one insect and going back to the list view. The following figure shows three screenshots of the app, from left to right, the List view, the List view with scrolling, and the detail view for the second insect:



Figure 16.40: watchOS app previews

How it works...

When building multiplatform applications, it is important to figure out how to organize the code in a way that can be shared among the different platforms. Structuring your code in groups and different files with smaller components increases readability and code reusability. In this recipe, we shared the files in the `Resources` and `Models` groups among all targets.

When using a multiplatform target, we can share the target among all destinations except for watchOS. For this reason, we created a new target for the watchOS app, and we chose a standalone watch app. This means that the watch app doesn't need a companion iOS app to work, and it can be published to the watchOS App Store by itself.

Selecting the files and checking the `watchOSMultiplatform Watch app` target from the Xcode file inspector pane made those source files available to the watchOS app.

We opted to share `InsectListView.swift` among all the targets, and we used conditional compilation to have different view code for our different platforms. However, for the root view `ContentView`, the cell view `InsectCellView`, and the detail view `InsectDetailView`, we chose different source files for the watchOS app, rather than reusing the existing files.

In this last recipe of the chapter, we have applied all the techniques that have been learned in the previous recipes. Now you're equipped with the complete knowledge needed to build multiplatform apps in SwiftUI.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

<https://packt.link/swiftUI>



17

SwiftUI Tips and Tricks

In the previous chapters, we tried to solve different problems in 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 by exploring how we can test SwiftUI views using XCTest, which is Apple's official framework for implementing automated tests.

SwiftUI provides a variety of built-in fonts that we can use in most of our apps but, sometimes, we may want more customization. So, we'll see how to use custom fonts in SwiftUI.

Sometimes we must show some kind of documentation in the app, so we'll see how to present **Portable Document Format (PDF)** documents.

Finally, we'll implement a Markdown editor that shows an attributed preview while adding 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:

- Using XCTest to test SwiftUI apps
- Using custom fonts in SwiftUI
- Showing a PDF in SwiftUI
- Implementing a Markdown editor with preview functionality

Technical requirements

The code in this chapter is based on Xcode 15.0 and iOS 17.0. You can download and install the latest version of Xcode from the App Store. You'll also need to be running macOS Ventura (13.5) or newer.

Simply search for Xcode in the App Store and select and download the latest version. Launch Xcode and follow any additional installation instructions that your system may prompt you with. Once Xcode has fully launched, you're ready to go.

All the code examples for this chapter can be found on GitHub at <https://github.com/PacktPublishing/SwiftUI-Cookbook-3rd-Edition/tree/main/Chapter17-SwiftUI-Tips-and-Tricks>.

Using XCTest to test SwiftUI apps

When we build an app and implement the different features, we need to test that the app works correctly. Testing an app manually is a repetitive and time-consuming process, which becomes very boring. It is very common to make mistakes when performing boring and repetitive tasks. These mistakes could lead to the release of an app with defects to the App Store, affecting our users. Luckily, with the help of test automation, we can make Xcode test the app for us. Test automation is a very important discipline in software engineering and is based on the use of automated tests, with different levels of abstraction. The automated tests verify that the app under test works according to a well-known specification. Automated tests are a part of the software development process, and they verify that the code changes did not inadvertently introduce defects. With adequate automated test coverage, we have a higher confidence level, and we can catch the code defects before the app gets published.

In this recipe, we will use Apple's XCTest framework, included with Xcode, to implement UI automated tests in a simple SwiftUI app with one screen. We will test that the different views, such as text views or buttons, are present and have the right configuration. We will also test that the actions of the buttons behave as expected.

Getting ready

Let's start by creating an Xcode SwiftUI project called `MyCountry`, remembering to enable **Include Tests**, as shown in the following screenshot:

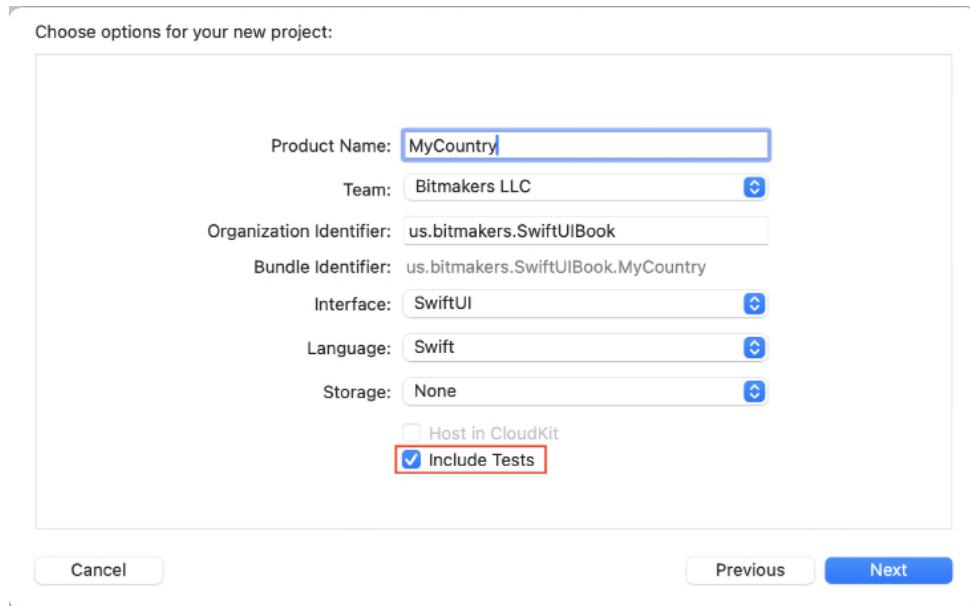


Figure 17.1: MyCountry app with automated tests included

How to do it...

We are going to implement an app that asks you to choose your country of origin from a few choices. The main screen presents a question at the top, and a list of buttons right below. Each button represents a country. After tapping on the button representing your country, the name of the country is displayed prominently on the screen. Additionally, the app keeps count of the number of taps on the buttons and displays a counter with the number of taps. Finally, we have a reset button, which sets the counter of taps to zero and selects the first country.

Let's build the app following these steps:

1. Let's go to `ContentView.swift` and define a list of possible countries we can choose from, and two `@State` variables, one to hold the selected country and the other to store the number of taps. Add this code immediately after the `ContentView` declaration:

```
static let countries = ["USA", "Spain", "France", "Italy"]
@State private var selectedCountry: String = Self.countries[0]
@State private var counter = 0
```

2. Replace the contents of the `body` variable with the following code, which will display three text views—one at the top with a text asking the user to choose a country, a second one in the middle with the selected country name, and a third one at the bottom with the number of taps:

```
var body: some View {
    VStack {
        Text("What is your country of origin?")
        HStack(spacing: 12) {
            // Buttons to choose a country
        }
        Spacer()
        Text(selectedCountry)
            .font(.system(size: 40))
        Spacer()
        HStack {
            Text("Number of taps: \((counter)")
            Spacer()
            // Button to reset the counter
        }
        .padding()
    }
}
```

3. Now, add a list of buttons to select the countries—one for each country. Replace the `HStack` after the first text view with the following:

```
HStack(spacing: 12) {
```

```
ForEach(Self.countries, id:\.self) { country in
    let isSelected = selectedCountry == country
    Button {
        selectedCountry = country
        counter += 1
    } label: {
        Text(country)
            .frame(width: 80, height: 40)
            .background((isSelected ? Color.red : .blue).
            opacity(0.6))
            .cornerRadius(5)
            .foregroundStyle(.white)
    }
}
```

4. Finally, let's add the `Reset` button embedded in the last `HStack`. Replace the comment with a `Button` to reset the selected country and the counter. The code should look like this:

```
HStack {
    Text("Number of taps: \((counter)")
    Spacer()
    Button("Reset") {
        selectedCountry = Self.countries[0]
        counter = 0
    }
    .buttonStyle(.bordered)
}
```

5. The app is now finished. Use the live preview of the canvas and interact with the app. Notice how the app starts with **USA** as the elected country and the number of taps is 0. As we start tapping the buttons, we see that the selected country text changes and the counter with the number of taps increases. Finally, tap on the **Reset** button to go back to the initial state. Here are some screenshots of the app:

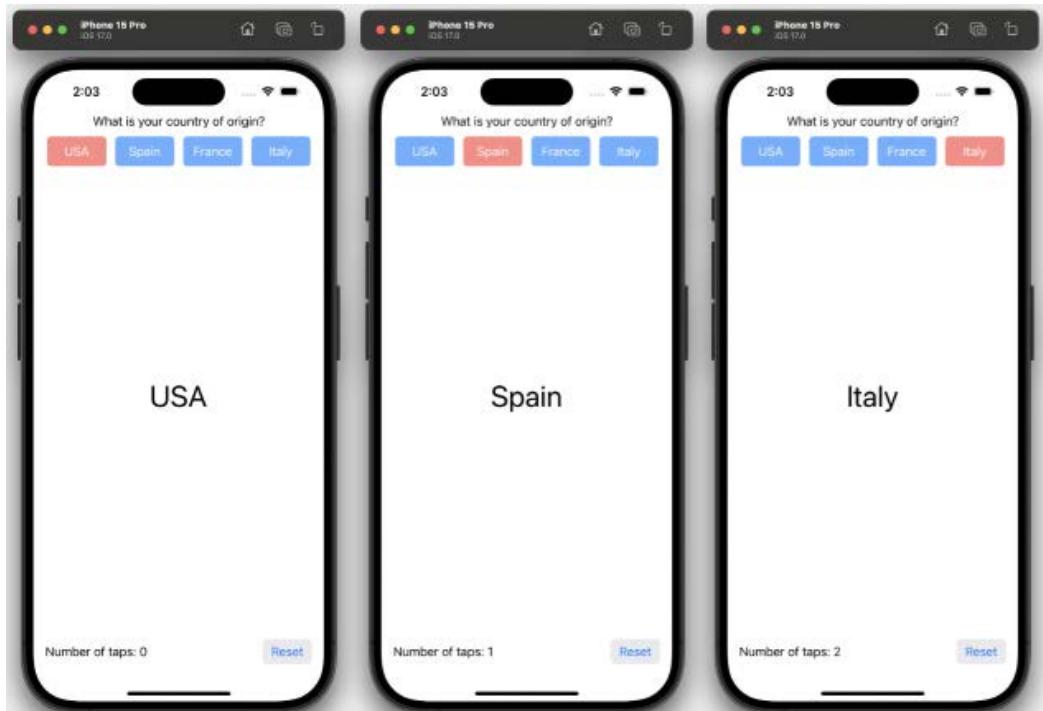


Figure 17.2: MyCountry app screenshots

6. We are ready to start our journey on automated tests. We will implement UI tests, which test the app using the **user interface (UI)** and simulating user interactions. Xcode accomplishes this purpose using the **Accessibility** features of iOS. These **Accessibility** features are in place to support users with vision, mobility, hearing, speech, and cognitive needs. For example, a user with vision needs could enable **VoiceOver** to hear what is on the screen.

Give it a try and enable VoiceOver from the iOS Settings app on your iPhone, as shown in *Figure 17.3*, so you can understand how it works:

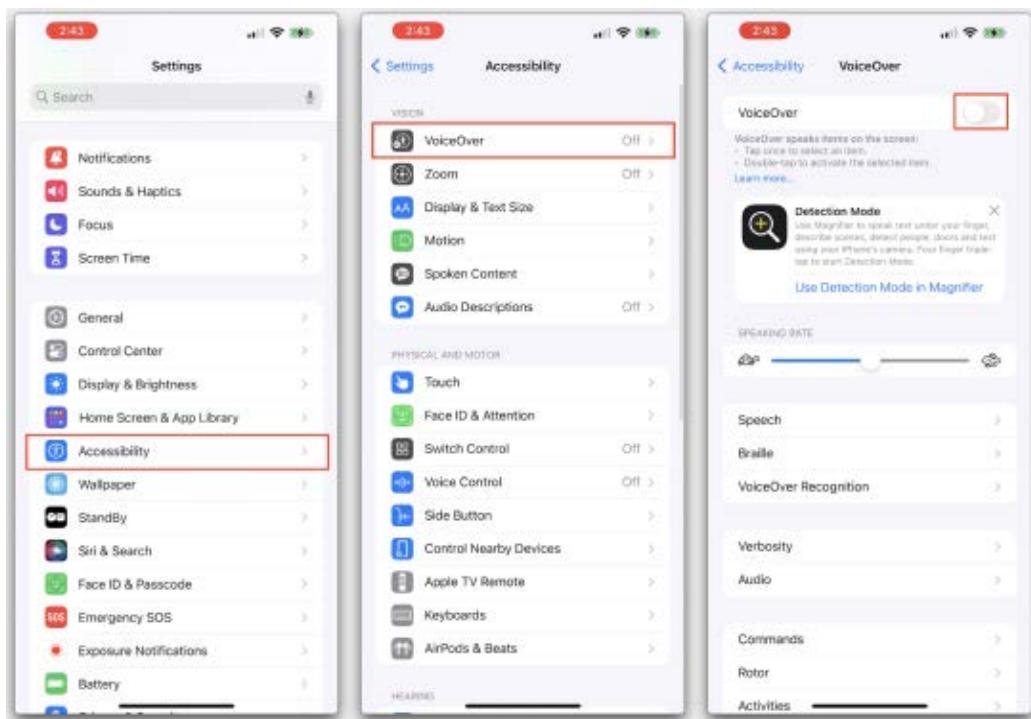


Figure 17.3: Enabling VoiceOver

7. Testing the app via its UI is the most end-to-end way to implement an automated test, and it will give us the biggest confidence that the app works as expected. XCTest uses the accessibility attributes of SwiftUI views to interact with them. A good practice while implementing UI tests is to attach accessibility identifiers to the views we are interested in testing. Accessibility identifiers allow us to assign a String of our choice to uniquely identify a view.
8. Switch to `ContentView.swift` and add accessibility identifiers to the three text views. We accomplish this by adding the `.accessibilityIdentifier()` view modifier. The three text views should look like this:

```
Text("What is your country of origin?")
    .accessibilityIdentifier("Question")
// ...
Text(selectedCountry)
```

```
.font(.system(size: 40))
.accessibilityIdentifier("SelectedCountry")
// ...
Text("Number of taps: \(counter)")
.accessibilityIdentifier("NumberOfTaps")
```

9. Let's add accessibility identifiers to the buttons of our app. For each of the buttons representing a country, we will use the first two letters of the country name as the accessibility identifier, and for the button used to reset the UI, we will use the word *Reset* as the accessibility identifier. The code for the buttons should look like this:

```
Button {
    selectedCountry = country
    counter += 1
} label: {
    Text(country)
        .frame(width: 80, height: 40)
        .background((isSelected ? Color.red : .blue).opacity(0.6))
        .cornerRadius(5)
        .foregroundStyle(.white)
}
.accessibilityIdentifier(String(country.prefix(2)).uppercased())
// ...
Button("Reset") {
    selectedCountry = Self.countries[0]
    counter = 0
}
.buttonStyle(.bordered)
.accessibilityIdentifier("Reset")
```

10. With all the accessibility identifiers in place, we can finally start with the UI tests. If you inspect the project structure using the Xcode navigator pane, you'll notice that we have two extra groups, each one with a target, as shown in *Figure 17.4*. The first group, *MyCountryTests*, is for unit tests, which are used to make sure that our code works as intended and is out of the scope of this recipe. If you want to learn more about unit tests and test automation, check the links provided at the end of the recipe.

The second group, `MyCountryUITests`, is for the UI tests we are going to be implementing.

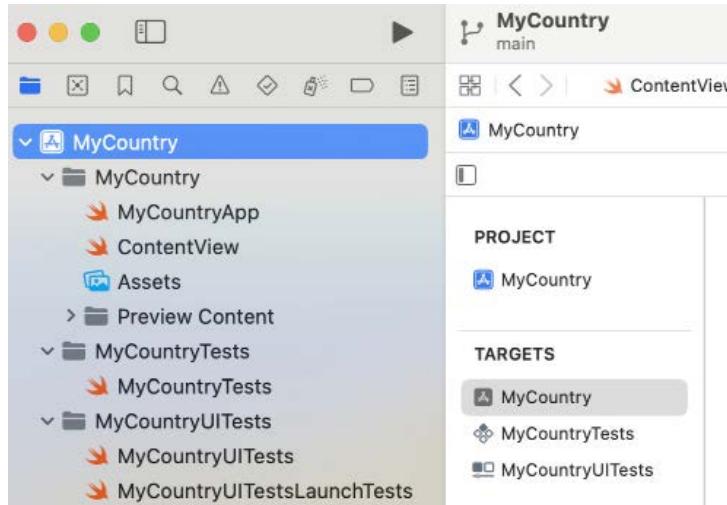


Figure 17.4: Test groups and targets

11. Switch to the `MyCountryUITests.swift` file, delete all the content inside the class declaration, and replace it with the following code:

```
import XCTest

final class MyCountryUITests: XCTestCase {

    override func setUpWithError() throws {
        continueAfterFailure = false
    }

    override func tearDownWithError() throws {
    }
}
```

12. Add a constant property to the class to refer to the `XCUITApplication` proxy, which is an object that can launch, monitor, and terminate our app under test. We will use this proxy in the test methods that we will write:

```
final class MyCountryUITests: XCTestCase {

    let app = XCUITApplication()

    // ...
}
```

13. In the `setUpWithError()` function, which is called before each of the automated tests runs, let's instruct the proxy to launch the app:

```
override func setUpWithError() throws {
    continueAfterFailure = false
    app.launch()
}
```

14. Now it's time to work on our first automated test. By convention, the function representing an automated test starts with the word `test`. Below the `tearDownWithError()` function, add the following function:

```
func testButtonLabels() throws {
    // ...
}
```

15. Each UI automated test follows the same pattern. We set the UI to an initial state, interact with the UI, and then verify that the UI is the new expected state. In test automation, we call `assertion` to this verification:

```
func testButtonLabels() throws {
    // 1. Set the initial state of the UI
    // 2. Interact with the UI
    // 3. Verify the UI is in the expected
}
```

16. In our case, we would like to test that the five buttons of the app are present and that each button label is set to the expected value. Since our app is simple and only has one screen, we get to our initial state by simply launching the app. The proxy performs this action in the `setUpWithError()` function, which is called right before our test method is called. We start the test by creating a dictionary to map each accessibility identifier of the button under test with its expected label, and then we iterate over the accessibility identifiers:

```
func testButtonLabels() throws {
    let expectedLabels = ["US": "USA", "SP": "Spain", "FR": "France",
    "IT": "Italy", "Reset": "Reset"]
    let identifiers = expectedLabels.keys
    for identifier in identifiers {
        // more code will follow
    }
}
```

17. For each button we get the expected label, we ask the proxy to retrieve the button from the UI using the accessibility identifier, and then we assert that the label of the button is equal to the expected label. The final code should be as follows:

```
func testButtonLabels() throws {
    let expectedLabels = ["US": "USA", "SP": "Spain", "FR": "France",
    "IT": "Italy", "Reset": "Reset"]
    let identifiers = expectedLabels.keys
    for identifier in identifiers {
        let expectedLabel = expectedLabels[identifier]!
        let button = app.buttons[identifier]
        let label = button.label
        XCTAssertEqual(expectedLabel, label, "Button \(expectedLabel) not
present")
    }
}
```

18. Time to run our first automated test and see how it works! Make sure you have an iOS simulator selected as the run destination from the Xcode toolbar, as shown in the following screenshot:



Figure 17.5: Destination to run UI tests

19. To run our test, we have several options, but for now, we are going to run only the test method that we want. You'll observe that next to the name of the function, there is a diamond icon. If you hover over the icon, you will see that it changes to a solid-gray diamond with an arrow, ◊. Click on the diamond icon and the test will start:



```
func testButtonLabels() throws {
    let expectedLabels = ["US": "USA", "SP": "Spain", "FR": "France", "IT": "Italy", "Reset": "Reset"]
    let identifiers = expectedLabels.keys
    for identifier in identifiers {
        let expectedLabel = expectedLabels[identifier]!
        let button = app.buttons[identifier]
        let label = button.label
        XCTAssertEqual(expectedLabel, label, "Button '\(expectedLabel)' not present")
    }
}
```

Figure 17.6: Button to run the specific test method

20. After clicking on the diamond icon, Xcode will launch an instance of the selected simulator and then run the test. You will see how the app launches and after completing the test, it goes back to the background. Make sure that the Xcode console is not hidden and take a look at its content. You will see the log statements of the test execution and the test result. Your log statements should be like the following:

```
2023-10-15 18:35:10.930461-0500 MyCountryUITests-Runner[80846:7463385]
[Default] Running tests...
Test Suite 'MyCountryUITests' started at 2023-10-15 18:35:11.468.
```

```
Test Case '-[MyCountryUITests.MyCountryUITests testButtonLabels]' started.  
t = 0.00s Start Test at 2023-10-15 18:35:11.469  
t = 0.02s Set Up  
t = 0.02s Open us.bitmakersSwiftUIBook.MyCountry  
t = 0.02s Launch us.bitmakersSwiftUIBook.MyCountry  
t = 0.63s Setting up automation session  
t = 0.97s Wait for us.bitmakersSwiftUIBook.MyCountry  
to idle  
t = 2.08s Find the "US" Button  
t = 2.11s Find the "SP" Button  
t = 2.12s Find the "FR" Button  
t = 2.12s Find the "IT" Button  
t = 2.13s Find the "Reset" Button  
t = 2.13s Tear Down  
Test Case '-[MyCountryUITests.MyCountryUITests testButtonLabels]' passed  
(2.362 seconds).  
Test Suite 'MyCountryUITests' passed at 2023-10-15 18:35:13.832.  
Executed 1 test, with 0 failures (0 unexpected) in 2.362 (2.364)  
seconds
```

21. After testing the buttons of our app, we would like to test that the three text views are present and contain the expected initial label. We want to create a new test method to achieve this. Add the following function after the function created in the previous step:

```
func testTextLabels() throws {  
    // ...  
}
```

22. The code to test the text views is very similar to the code we used to test the buttons. The test method should look like this:

```
func testTextLabels() throws {  
    let expectedLabels = ["Question": "What is your country of origin?",  
    "SelectedCountry": "USA", "NumberOfTaps": "Number of taps: 0"]  
    let identifiers = expectedLabels.keys  
    for identifier in identifiers {  
        let expectedLabel = expectedLabels[identifier]!  
        let text = app.staticTexts[identifier]  
        let label = text.label  
        XCTAssertEqual(expectedLabel, label, "Text '\(expectedLabel)'  
not present")  
    }  
}
```

23. With the previous two functions, testing the five buttons and the three texts of our app, we have tested that our UI is in the initial expected state. Now, we will perform some actions and test that the UI is in a new expected state after the actions have been completed. We will test the country buttons. Add the following function at the end of our class:

```
func testCountryButtonAction() throws {
    // ...
}
```

24. We will start testing that when we tap on a country button, the text in the middle of the screen changes to the country name, and that the text with the number of taps changes too. We will test the four buttons in a single function so we will cycle through them using a **for-in** loop. For each of the buttons, the test steps would be as follows:

- a. Get the button
- b. Tap on the button
- c. Verify that the country text view changed to the selected country name
- d. Verify that the text view with the number of taps changed to reflect the correct number of taps

The code to perform this test should be as follows:

```
func testCountryButtonAction() throws {
    var counter = 0
    let identifiers = ["SP", "FR", "IT", "US"]
    for identifier in identifiers {
        let button = app.buttons[identifier]
        button.tap()
        counter += 1
        XCTAssertEqual(app.staticTexts["SelectedCountry"].label, button.
label)
        XCTAssertEqual(app.staticTexts["NumberOfTaps"].label, "Number of
taps: \(counter)")
    }
}
```

25. The last test we need to implement is to test the functionality of the **Reset** button. What we would do is interact with the app and tap twice on a known country button, verify that the app is in the expected initial state, tap on the **Reset** button, and verify that the app is in the default initial state. The selected country should be the USA and the number-of-taps label should display zero taps. The test method should be as follows:

```
func testResetButtonAction() throws {
    app.buttons["Spain"].tap()
    app.buttons["Spain"].tap()
```

```

XCTAssertEqual(app.staticTexts["SelectedCountry"].label, "Spain")
XCTAssertEqual(app.staticTexts["NumberOfTaps"].label, "Number of
taps: 2")
app.buttons["Reset"].tap()
XCTAssertEqual(app.staticTexts["SelectedCountry"].label, "USA")
XCTAssertEqual(app.staticTexts["NumberOfTaps"].label, "Number of
taps: 0")
}

```

26. Now that we have finished with all the test methods, let's run the whole set of tests, which in test automation jargon is called a *test case* or *test suite*. To do so, click on the diamond next to the class name and all the test methods inside the class will run. When the tests are complete, you'll see the results in the console or, in a graphical way, by choosing the Report navigator in the Xcode navigator pane, as shown in the following screenshot:

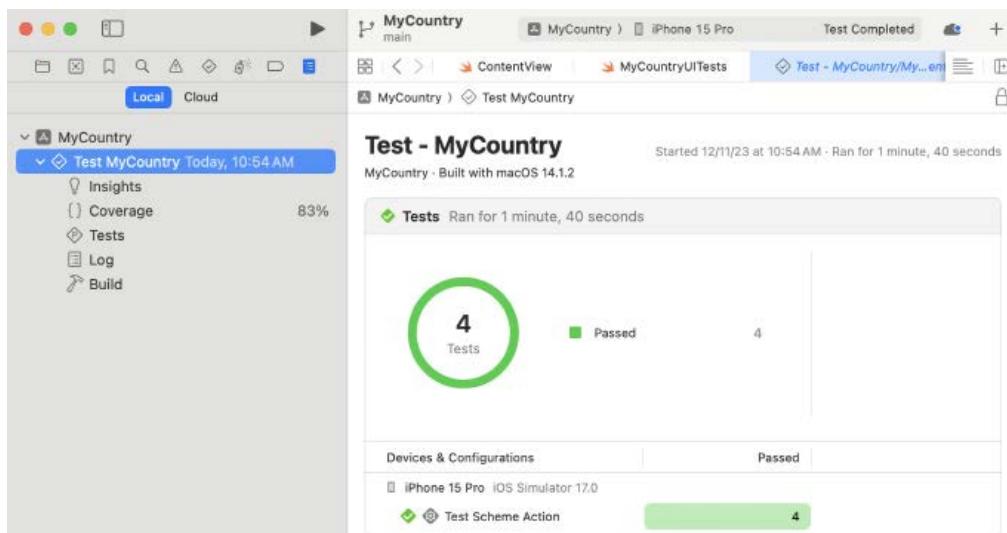


Figure 17.7: Report navigator showing UI test results

How it works...

When we want to implement automated tests in our app, we need to prepare our code for it. In our case, we started modifying the code by adding accessibility identifiers to the UI elements that we were interested in testing.

We used the `.accessibilityIdentifier(_:)` view modifier in the five buttons and the three text views of our app. The use of accessibility identifiers allows our test code to refer to the specific UI element by its identifier, instead of searching for the UI element by one of its attributes, such as the label. This is especially useful when the label of our element is dynamic and changes depending on our data. For example, in the text view at the bottom of the screen, which displays the number of taps, the label changes to reflect the number of taps, but the *Number of Taps* accessibility identifier is independent of our data and never changes. This makes it easier to refer to the view we want to test.

We implemented UI automated tests using Apple's XCTest framework. Our test case was implemented with the `MyCountryUITests` class, which inherits from `XCTestCase`, and included four test methods. Each method was implemented with a function that starts with the word `test`, which allows Xcode to identify the function as a test method and include a diamond icon to the right of the name (*Figure 17.6*).

Test cases make use of the `XCUIAutomation` object, which acts as a proxy to interact with the app under test. The proxy can access the UI using accessibility attributes and return `XCUIElement` instances representing UI elements, which we can interact with. After interacting with the UI elements, we check for certain attributes using test assertions. A test assertion checks for a certain condition and, if that condition is not met, then it fails the test case and displays an optional error message.

For example, let's look at the following code:

```
let app = XCUIApplication()
let button = app.buttons["Reset"]
button.tap()
XCTAssertEqual(app.staticTexts["NumberOfTaps"].label, "Number of taps: 0")
```

Here,

- We instantiate the app proxy and store it in the `app` variable.
- We ask the proxy for the button with the `Reset` accessibility identifier and then store the `XCUIElement` representing the button in the `button` variable.
- We ask the element to send a `tap` event to the button with the `tap()` function.
- We ask the proxy to get the text with the `NumberOfTaps` accessibility identifier and assert that the value of the label of the text is equal to the text “Number of taps: 0.”

There's more...

In the app of this recipe, we have used UI automation tests, which gives us very high confidence that the app works as expected because we interact with the UI with automated actions and read the UI state after these interactions. This high fidelity comes at the cost of execution time, as UI automated tests take longer to run than other kinds of tests. These tests take seconds to complete.

XCTest provides the ability to write tests with different levels of abstraction. Of special interest are unit tests, which test small units of our code in an automated way, and integration tests, which test how objects interact together. Unit tests and integration tests work very differently than the UI automation tests. Due to these differences, Xcode created a separate target for them. If you look at the targets for the project, you'll see we have three targets:

```
MyCountry: app target
MyCountryTests: target for unit and integration tests
MyCountryUITests: target for UI tests
```

With unit tests and integration tests, we have access to our app's code, and we can instantiate objects and interact with them to verify that they behave as expected.

Instantiating objects is a very fast process, and the execution time is in the milliseconds range, more than a thousand times faster than UI automation tests.

Let's look at unit tests with an example. Imagine we have a `Counter` class that models a counter with two methods: `increment` to increment the counter and `reset` to reset the counter to zero. The code for the class is as follows:

```
public class Counter {  
    private(set) var value: Int!  
    @discardableResult public func increment() -> Int {  
        value += 1  
        return value  
    }  
    @discardableResult public func reset() -> Int {  
        value = 0  
        return value  
    }  
}
```

A good test case class using unit tests would be this:

```
import XCTest  
@testable import MyApp  
class ModelTests: XCTestCase {  
    private var model: Counter!  
    override func setUpWithError() throws {  
        model = Counter()  
    }  
    func test_reset_setsCounterToZero() throws {  
        model.reset()  
        XCTAssertEqual(model.value, 0)  
    }  
    func test_try_incrementsCounterBy1() throws {  
        model.reset()  
        model.increment()  
        XCTAssertEqual(model.value, 1)  
    }  
}
```

The `@testable import MyApp` statement at the top of the test case class, `ModelTests`, gives full access to the internal elements of our code in the `MyApp` module. The advantage of unit tests is that we don't need to launch the app and wait for the UI; we can just instantiate the objects in the app and interact with them.

In our test method, `test_reset_setsCounterToZero()`, we instantiate the Counter class, we call the `reset()` method on an instance, and then we assert that the `value` variable is equal to zero. Notice how, in the Counter class, we declared the `value` variable with a `private(set)` access modifier. This access modifier makes the variable private to writing but internal for reading. In this way, we can read the variable from the test case class.

Unit test and integration test methods follow the same pattern:

1. We set up the object to an initial and known state.
2. We interact with the object using its public and internal methods.
3. We compare the object state with the expected state and then fail or pass the test with an assertion.

The above 3-step process is called *Arrange-Act-Assert* in test automation jargon.

A good testing strategy combines multiple types of tests to maximize the benefits of each. If you're interested in knowing more about test automation and a good testing strategy, check the link for *The Practical Test Pyramid*, referenced in the next section.

See also

- **The Practical Test Pyramid** is a reference document when it comes to test automation and implementing a testing strategy: <https://martinfowler.com/articles/practical-test-pyramid.html>.
- **Testing your apps in Xcode** is Apple's explanation of the test pyramid and the different types of tests: <https://developer.apple.com/documentation/xcode/testing-your-apps-in-xcode/>.
- **XCTest official documentation**: <https://developer.apple.com/documentation/xctest>.

Using custom fonts in SwiftUI

iOS comes with many preinstalled fonts that can be safely used in SwiftUI. However, the design of the app sometimes needs some custom fonts because they might be a part of the brand of the app maker.

In this recipe, we'll see how to import a couple of custom fonts for building a menu page for an Italian restaurant.

Getting ready

1. Create a SwiftUI app called `RestaurantMenu`.
2. Get 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-3rd-Edition/tree/main/Resources/Chapter17/recipe2>.

3. Drag the font files to the Xcode project navigator pane to add the fonts as resources to the project, as shown in the following figure:

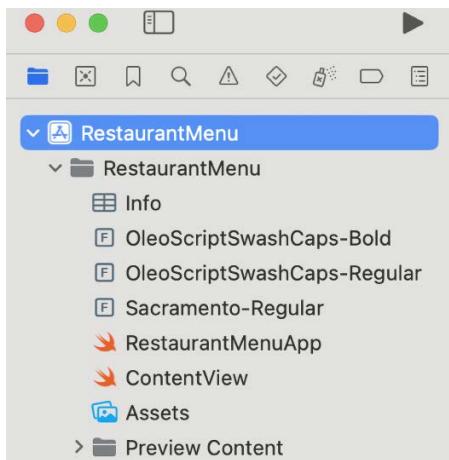


Figure 17.8: Custom fonts added to the project

4. Make sure the **Copy items if needed** and **RestaurantMenu** options are checked:

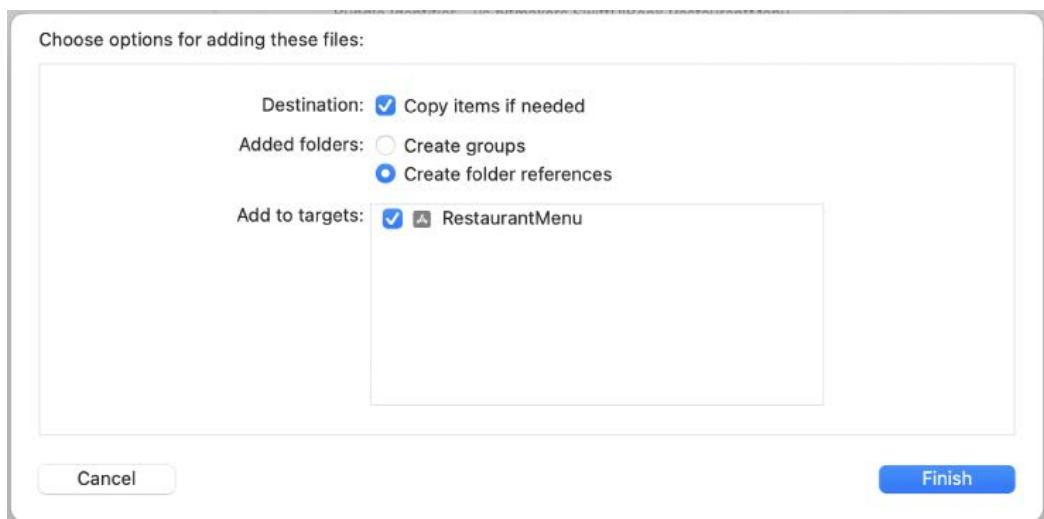


Figure 17.9: Adding custom font files to the project

5. Finally, we must register the fonts with the iOS app. For this, navigate to the **Info** tab on the app target, and disclose the **Custom iOS Target Properties** section, as shown in the following figure:

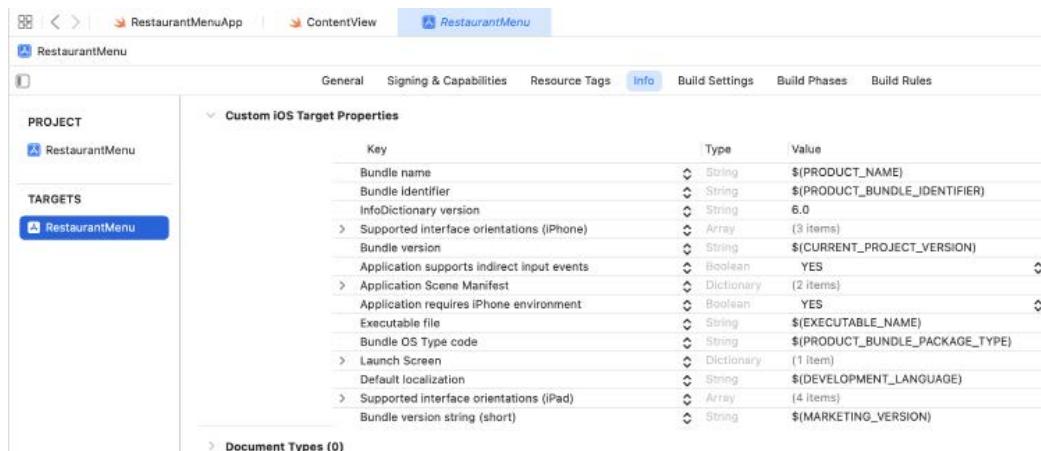


Figure 17.10: Custom iOS Target Properties

6. Hover over any of the keys, and when a small plus icon appears to the right of the key, click on it to add a new key. Start typing the name of the **Fonts provided by application** key and when you see the text in the drop-down menu, press the *Return* key twice, as shown in *Figure 17.11*:

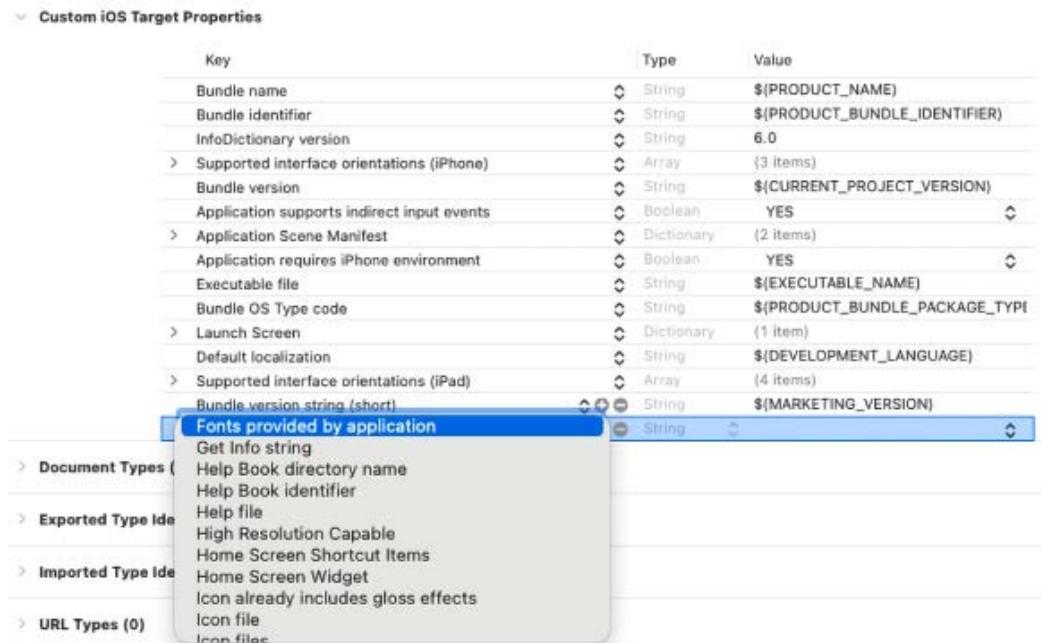


Figure 17.11: Adding fonts to Custom iOS Target Properties

7. Add the name of the three font files (including the file extension) as items to the array, as shown in the following figure:

Custom iOS Target Properties			
Key	Type	Value	
Bundle name	String	<code>\$(PRODUCT_NAME)</code>	
Bundle identifier	String	<code>\$(PRODUCT_BUNDLE_IDENTIFIER)</code>	
InfoDictionary version	String	6.0	
Supported interface orientations (iPhone)	Array	{3 items}	
Bundle version	String	<code>\$(CURRENT_PROJECT_VERSION)</code>	
Application supports indirect input events	Boolean	YES	
Application Scene Manifest	Dictionary	{2 items}	
Application requires iPhone environment	Boolean	YES	
Executable file	String	<code>\$(EXECUTABLE_NAME)</code>	
Fonts provided by application		Array {3 items}	
Item 0	String	OleoScriptSwashCaps-Bold.ttf	
Item 1	String	OleoScriptSwashCaps-Regular.ttf	
Item 2	String	Sacramento-Regular.ttf	
Bundle OS Type code	String	<code>\$(PRODUCT_BUNDLE_PACKAGE_TYPE)</code>	
Launch Screen	Dictionary	{1 item}	
Default localization	String	<code>\$(DEVELOPMENT_LANGUAGE)</code>	
Supported interface orientations (iPad)	Array	{4 items}	
Bundle version string (short)	String	<code>\$(MARKETING_VERSION)</code>	

▶ Document Types (0)

Figure 17.12: Custom fonts added to Custom iOS Target Properties

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 `Font.custom()` function.

We'll use it for our custom fonts. Follow these steps:

- At the top of `ContentView.swift`, implement an extension to the `Font` class. We add three 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)
    }
}
```

- Immediately after the extension to the `Font` class, create a custom view. This view will be used to display an entry on the menu page of the restaurant:

```
struct ItemView: View {
    let name: String
```

```
let price: String

var body: some View {
    HStack {
        Text(name)
        Spacer()
        Text(price)
    }
    .font(.sacramento(size: 40))
}
}
```

3. Finally, modify the content of the `body` variable of `ContentView` with the following code:

```
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. Using the live preview in the canvas, we see the sleek interface created by the custom fonts. It should look like in the following screenshot:



Figure 17.13: Our restaurant menu page

How it works...

SwiftUI provides a list of semantic fonts, such as `body`, `callout`, and so on, that allows the styling of different parts of a page with a lot of text.

The `Font` struct also has a static function called `custom()` whose parameters are the names 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 must use it can be error-prone, so it is better to create a static function in an extension of the `Font` struct. This way, it can be statically checked during the compilation, reducing the possibility of errors.

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 have provided a sample PDF document in the repository but feel free to use the PDF document of your choice:

<https://github.com/PacktPublishing/SwiftUI-Cookbook-3rd-Edition/blob/main/Resources/Chapter17/recipe3/PDFBook.pdf>

Copy the PDF document into the project:

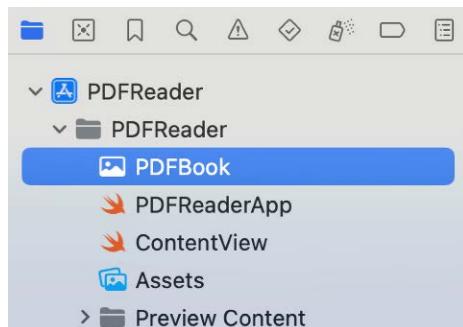


Figure 17.14: PDF document in the Xcode project

With the PDF document added as a resource to 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` struct 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. Go to `ContentView.swift` and, at the top of the file, define a struct called `PDFKitView` to bridge `PDFView` to SwiftUI:

```
import SwiftUI
import PDFKit

struct PDFKitView: UIViewRepresentable {
    let url: URL

    func makeUIView(context: UIViewRepresentableContext<PDFKitView>) -> PDFView {
        let pdfView = PDFView()
        pdfView.document = PDFDocument(url: self.url)
        return pdfView
    }

    func updateUIView(_ uiView: PDFView,
                      context: UIViewRepresentableContext<PDFKitView>) {
    }
}
```

2. The newly created component can then be used anywhere in our SwiftUI code. Add it to the `ContentView`. Replace the `struct` content with our custom code:

```
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 the live preview of the canvas, the PDF document is accurately rendered in our `PDFKitView` component:

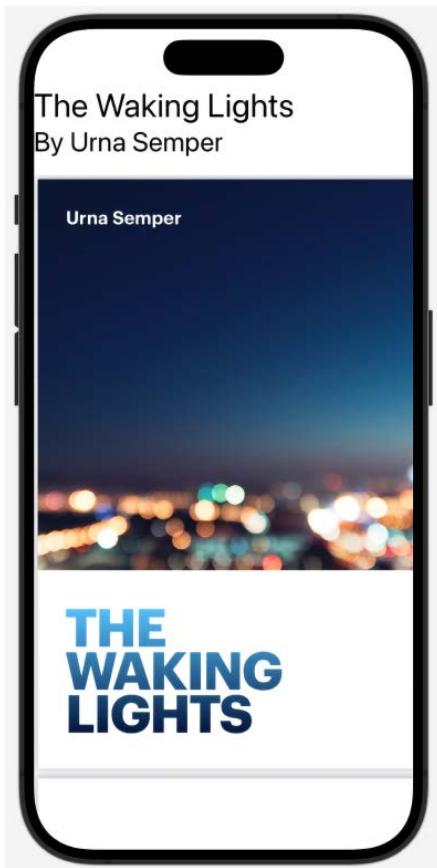


Figure 17.15: PDF document rendered in SwiftUI

How it works...

We first define a `PDFKitView` struct conforming to the `UIViewRepresentable` protocol, which creates and holds an instance of the `PDFView` UIKit class. After creating it, we set the URL of the document to present in our view. This `PDFKitView` view is then used in the `ContentView` view as a normal SwiftUI view.

The `PDFView` class renders a document from a URL. The sample PDF document we are using is in the app bundle, so we must provide the URL to the document in the main Bundle of the app.

Since `PDFView` 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 full-screen PDF reader, where we present a page in full-screen, and navigate through each page using custom buttons. It should be a matter of exposing the `PDFView` functions in our `PDFKitView` struct and connecting them to normal SwiftUI buttons.

Implementing a Markdown editor with preview functionality

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 for Markdown tags in the `Text` views.

In this recipe, we'll experiment with Markdown, implementing a simple text editor.

Getting ready

This recipe doesn't require any preliminary operations. Create a new 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 view and a preview view, where the text is rendered as rich text, with bold, italics, and so on.

The text editor view will have a tab bar above the editor itself for adding Markdown tags for the bold, italics, strikethrough, and code formats.

Let's get started. We will be working in the `ContentView.swift` file, so go to the file and follow these steps:

1. At the top, start by creating a `FormatButton` view, customizing the button's appearance, and passing a closure to be called when the button is tapped:

```
struct FormatButton: View {  
    let label: String  
    let onTap: () -> Void  
  
    var body: some View {  
        Button {  
            onTap()  
        } label: {
```

```
        Text(.init(label))
            .frame(width:30)
    }
    .buttonStyle(.bordered)
}
}
```

2. Immediately after this, create a `FormatBar` view, containing a list of buttons that will format the text in various styles:

```
struct FormatBar: View {
    @Binding var text: String
    var body: some View {
        HStack {
            FormatButton(label: "***B***") {
                text += "***"
            }
            FormatButton(label: "*I*") {
                text += "*"
            }
            FormatButton(label: "***B***") {
                text += "***"
            }
            FormatButton(label: "~~S~~") {
                text += "~~"
            }
            FormatButton(label: "'C'") {
                text += "'"
            }
        }
    }
}
```

3. Now we create the `EditorView`, where we will use the `FormatBar` to edit our text:

```
struct EditorView: View {
    @Binding var text: String
    var body: some View {
        VStack(alignment: .leading) {
            FormatBar(text: $text)
            TextEditor(text: $text)
                .font(.title)
        }
    }
}
```

```
        }
    }
}
```

4. Create a `PreviewView` to render the rich text from the Markdown text entered 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, for the `ContentView`, replace its content with the following code. We will have a picker to choose between two views: the `Editor` view to enter our text, and the `Preview` view to visualize our formatted text:

```
struct ContentView: View {
    @State private var isEditMode = true
    @State private var text = ""

    var body: some View {
        VStack {
            Picker("Choose mode", selection: $isEditMode) {
                Text("Editor").tag(true)
                Text("Preview").tag(false)
            }
            .pickerStyle(.segmented)
            if isEditMode {
                EditorView(text: $text)
            } else {
                PreviewView(text: text)
            }
        }
        .padding(.horizontal)
    }
}
```

6. Using the live preview of the canvas, try the app. We can enter a text with Markdown tags in the editor view and visualize it as rich text in the preview view as shown in the figure:



Figure 17.16: A Markdown editor with a text preview

How it works...

Markdown is a simple language that, in a simple way, allows us to describe the style of text by 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

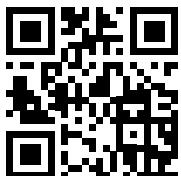
EditorView presents a list of buttons to add an initial or final tag to the `text` variable. However, the tag can be simply written as plain text and everything works.

From iOS 15, the Text view 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` using the code `Text(.init(text))`. Thanks to using this code, the Markdown tags are correctly rendered.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

<https://packt.link/swiftUI>





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

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:

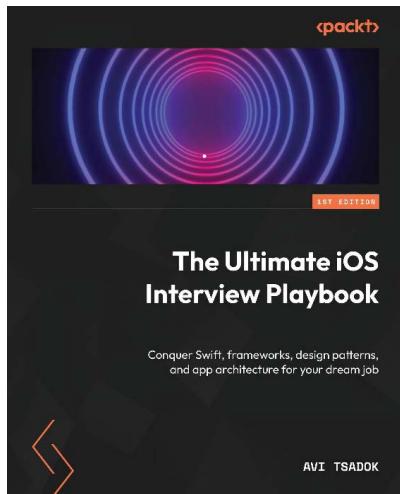


iOS 17 Programming for Beginners, Eighth Edition

Ahmad Sahar

ISBN: 9781837630561

- Discover the world of Xcode 15 and Swift 5.9, laying the foundation for your iOS development journey
- Implement the latest iOS 17 features through a hands-on example app, ensuring your apps remain innovative and engaging
- Build and deploy iOS apps using industry-standard design patterns and best practices.
- Implement the Model-View-Controller (MVC) design pattern to create robust and organized applications
- Expand your app's reach by effortlessly converting it for iPad, Mac, and visionOS
- Dive into UIKit, the essential framework for large-scale iOS projects

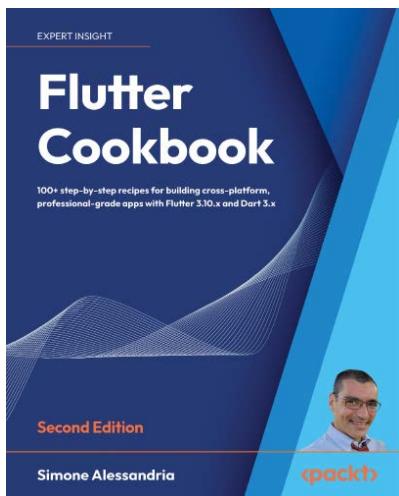


The Ultimate iOS Interview Playbook

Avi Tsadok

ISBN: 9781803246314

- Gain insights into how an interview process works
- Establish and capitalize on your iOS developer brand
- Easily solve general Swift language questions
- Solve questions on data structures and code management
- Prepare for questions involving primary frameworks such as UIKit, SwiftUI, and Combine Core Data
- Identify the “red flags” in an interview and learn strategies to steer clear of them

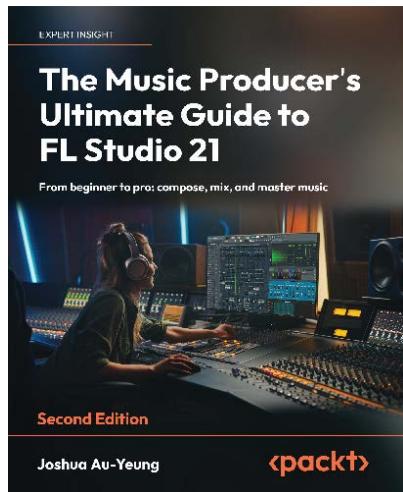


Flutter Cookbook, Second Edition

Simone Alessandria

ISBN: 9781803245430

- Familiarize yourself with Dart fundamentals and set up your development environment
- Efficiently track and eliminate code errors with proper tools
- Create various screens using multiple widgets to effectively manage data
- Craft interactive and responsive apps by incorporating routing, page navigation, and input field text reading
- Design and implement a reusable architecture suitable for any app
- Maintain control of your codebase through automated testing and developer tooling
- Develop engaging animations using the necessary tools
- Enhance your apps with ML features using Firebase MLKit and TensorFlow Lite
- Successfully publish your app on the Google Play Store and the Apple App Store

**The Music Producer's Ultimate Guide to FL Studio 21, Second Edition**

Joshua Au-Yeung

ISBN: 9781837631650

- Get up and running with FL Studio 21
- Compose melodies and chord progressions on the piano roll
- Mix your music effectively with mixing techniques and plugins, such as compressors and equalizers
- Record into FL Studio, pitch-correct and retime samples, and follow advice for applying effects to vocals
- Create vocal harmonies and learn how to use vocoders to modulate your vocals with an instrument
- Create glitch effects, transform audio samples into playable instruments, and sound design with cutting-edge effects
- Develop your brand to promote your music effectively
- Publish your music online and collect royalty revenues

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.

Share your thoughts

Now you've finished SwiftUI Cookbook, Third Edition, we'd love to hear your thoughts! If you purchased the book from Amazon, please [click here](#) to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.

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

Index

A

AddContacts app 547-553
advanced pickers
 using 25-28
alert buttons
 actions, adding to 201-203
 reference link 201
alerts
 presenting 198-201
AlertsWithActions app 201-203
Alignment
 reference link 195
Animate symbols
 reference link 37
AnimateTriangleShape project 315-318
Apple 1
app navigation titles
 reference link 235
async await function
 completion block function, converting to 468
 infinite scrolling, implementing with 473-478
AsyncAwaitSwiftUI app 456-459

B

banner
 creating, with spring animation 318-321
BannerWithASpringAnimation project 318-321

BarMark documentation

 reference link 584

basic animations

 creating 310-314

BasicAnimations app 310-314

@Binding

 using, to pass state variable to child Views 366-370

built-in shapes 280

 using 280-283

BuiltInShapes project 280-283

buttons

 adding 19-23

 navigating with 24

Buttons project 20-24

Button view 19

C

candlestick chart 637

 reference link 666

Canvas API

 using 290-293

CanvasPreview app 124-137

cells 193

chained animations

 with PhaseAnimator 330-334

ChainedAnimations project 330-334

ChartContentBuilder result builder
 documentation
 reference link 584

ChartContent protocol documentation
 reference link 584

ChartCustomizations app 584-599

ChartProxy documentation
 reference link 666

charts
 customizing 584-599

ChartSelection app 637-666

Chart view documentation
 reference link 584

ColorPicker views 44

Combine 361
 publishing capabilities 412
 subscription 412, 413

CombineCoreLocationManager project
 creating 405-411

completion block function
 converting, to async await 468

components
 laying out 4-8

confirmation dialogs
 presenting 204-207
 reference link 207

context menus 217
 creating 217-219
 reference link 219

controls 44-49

Core Data
 integrating, with SwiftUI 536-541

Core Data objects
 adding, from SwiftUI view 547-552
 deleting, from SwiftUI view 556-560
 showing, with @FetchRequest 541-547

Core Data requests
 filtering, with predicate 553-556

CoreLocation wrapper
 implementing, as @ObservedObject 370-375

Counter app 376-380

curved custom shape
 drawing 287-290

CustomAnimation protocol 3

custom animations
 with KeyframeAnimator 334-339

CustomAnimations project 334-339

custom fonts
 using 734-739

custom list row
 using 58-62

CustomRows app 58-62

custom shape 283
 drawing 284-287

custom types, encoding and decoding
 reference link 599

custom view transitions
 creating 339-342

D

data
 pulling and refreshing asynchronously, in
 SwiftUI 464

data visualization 575

DateBins structure 630

DatePicker views 25

DebuggingCombine app
 creating 440-445

Decimal type
 reference link 637

delay, applying to animation view modifier
 sequence of animations, creating 321-323

delay, applying to withAnimation function
 sequence of animations, creating 324-327

DelayedAnimations project 321-327

DeleteContacts app 556-559

disclosure groups

- used, for displaying content 105-109
- used, for hiding content 105-109

DisclosureGroups project 105-109

DisclosureGroup view 105-109

DisplayingContextMenu project 217-219

DisplayingPopovers project 214-217

Divider 4, 8

donut charts 630

- data, plotting in 630-636

DragGesture documentation

- reference link 666

Drawing app 290-293

DynamicTodoList project 366-370

dynamic type sizes

- reference link 138

E

easing functions

- reference link 314

editable collections

- creating 75, 76

EditableListsField project 75, 76

editable List view

- creating 69, 70

EditButton view 24

@EnvironmentObject 361

- state objects, sharing with multiple Views 380-387

environment values

- reference link 138

expanding lists

- hierarchical content, displaying in 101-104

ExpandingLists project 101-104

F

FakePosts app 459-463

FetchContact app 541-547

@FetchRequest

- Core Data objects, showing with 541-547

FilterContacts app 553-556

Firebase 489

- integrating, into SwiftUI project 489
- working 503

FirebaseNotes app 517-533

Firebase Remote Config 504

FocusAndSubmit project 169-172

Form 3

FormattedText project 9-12

FormFieldDisable project 165-168

forms 160

- filling out, with Focus and Submit 169-172
- items, disabling 166-168
- items, enabling 166-168
- reference link 165
- sections, displaying 160-164
- sections, hiding 160-164

FormValidation app

- creating 419-428

Form views 27, 159

full-screen covers 207

- used, for presenting new views 207-214

G

GeometryReader

- reference link 138

GithubUsers app

- creating 446-452

GoogleLogin app

- creating 505-516

gradient view
rendering 302-307

GradientViews app 302-307

Grid 3, 189
reference link 195
using 190-195

GridRow
reference link 195

groups of styles
applying, with ViewModifier 28-31

H

Heart app 287-290

hero transition 342

hero view transition
creating, with
.matchedGeometryEffect 342-349

hierarchical content

displaying, in expanding lists 101-104

histograms 623

with data bins 623-629

Histograms app 623-629

Hosting View Controller 44

HStack 4, 8

human interface guidelines, alerts
reference link 201

human interface guidelines, modality
reference link 214

human interface guidelines, popovers
reference link 217

I

IBSegueAction 44

Image
reference link 19

images
using 12-18

Image view 8

infinite scrolling 473
implementing, with async await 473-478

Inspector 3

interactive charts
scrollable content 667-675
selection 637-666

iOS app

creating, in SwiftUI 678-687
macOS version, creating with new
target 688-697
multiplatform version, creating with same
target 697-707
watchOS version, creating 708-717

items

disabling, in forms 166-168
enabling, in forms 166-169

J

JavaScript Object Notation (JSON) 153

K

KeyframeAnimator
for custom animations 334-339

KeyPathComparator
reference link 189

L

LabeledContent view 3

Layout protocol 3

lazy grids 85, 89

LazyGrids app 89-92

LazyHGrid view 89
tabular content, displaying with 90-92

LazyHStack view 86

using 86-89

lazy stacks 85

- LazyStacks app** 86-89
LazyVGrid view 89
 tabular content, displaying with 90-92
LazyVStack view 86
 using 86-89
legacy UIKit app
 SwiftUI, adding to 40-44
list
 rows, adding to 63-66
 rows, deleting from 67, 68
 sections, adding to 73-75
list of dynamic items 55
list of static items 55
 creating 56-58
ListRowAdd project 63-66
ListRowDelete app 67, 68
ListRowEdit project 69, 70
List views 51, 55
 rows, moving 71, 72
ListWithSections app 73-75
live preview canvas
 using, in Xcode15 124-137
- M**
- macOS version, iOS app**
 creating, with new target 688-696
- Markdown** 743
- Markdown editor**
 implementing, with preview
 functionality 743-747
- MarkdownEditor project** 743-747
- marks**
 using, to plot different charts 599-622
- .matchedGeometryEffect**
 hero view transition, creating 342-349
- MeasurementFormatter**
 reference link 189
- Menu**
 reference link 25
- MenuButtonView** 24
- Menu views** 44
- mock data**
 using, for previews 153-157
- modality** 197
- modal presentations**
 reference link 214
- model data**
 managing, with Observation 387
- Model-View-Controller (MVC)** 404
- Model-View-ViewModel (MVVM)** 404
- ModernNavigation app** 236-252
- MoreViewsAndControls project** 44-49
- MovingListRows project** 71, 72
- multi-column lists**
 creating, with Table 172-188
- multi-column navigation**
 with NavigationSplitView 260-268
- MultiColumnTable project** 173-188
- MultiDatePicker control** 3
- Multiplatform app** 678-687
- multiplatform version, iOS app**
 creating, with same target 697-707
- multiple animations**
 applying, to view 327-329
- MultipleAnimations project** 328, 329
- MyCountry project** 720-732
- N**
- navigation** 221
- navigation bars**
 reference link 235
- navigation containers** 221

- NavLink buttons** 24
 - reference link 25
- NavigationView** 2, 259
 - for multi-column navigation 260-268
- NavigationStack** 2
 - for simple navigation 222-235
 - for typed data-driven navigation 236-252
 - for untyped data-driven navigation 252-259
 - view, previewing in 138-142
- NavigationView Controller** 44
- NumberFormatter**
 - reference link 189
- O**
 - Observable macro** 3
 - ObservableObject protocol** 361
 - Observation** 361, 387
 - references 400
 - used, for managing model data 387-400
 - @ObservedObject** 361
 - CoreLocation wrapper, implementing as 370-375
- P**
 - PaletteGenerator app**
 - creating 468-473
 - Parameter Packs** 169
 - PasteButtons** 24
 - reference link 25
 - PDF**
 - showing 740-742
 - PDFKit** 740
 - PDFKitView** 741
 - PDFReader app** 740-742
 - PDFView** 740
 - PhaseAnimator**
 - for chained animations 330-334
 - PhaseAnimator struct** 3
 - PhotosPicker view** 3
 - pickers** 25
 - Picker views** 3, 25-28
 - PieAndDonutCharts app** 630-636
 - pie charts** 630
 - data, plotting in 630-636
 - Pixabay** 474
 - URL 474
 - Pixabay API**
 - using 474
 - popovers** 214
 - displaying 214-217
 - Practical Test Pyramid**
 - reference link 734
 - predicate**
 - Core Data requests, filtering with 553-556
 - documentation link 573
 - PresentingAlerts project** 198-200
 - PresentingConfirmationDialogs project** 205-207
 - PresentingSheets project** 207-214
 - Preview Content folder** 153
 - preview functionality**
 - Markdown editor, implementing with 743-747
 - PreviewingInNavigationStack app** 138-142
 - PreviewingWithTraits app** 143-145
 - preview macros, on Xcode** 15
 - reference link 149
 - PreviewOnDifferentDevices app** 146-148
 - previews**
 - mock data, using for 153-157
 - using, in UIKit 149-153
 - previews, in Xcode**
 - reference link 149
 - progress ring**
 - implementing 293-296

ProgressRings project 293-296

ProgressView 44

property wrappers 361

pull-to-refresh 464

R

random number generator

reference link 630

range area charts

reference link 622

range bar charts

reference link 622

Realm

reference link 536

RectangleMark

reference link 622

RefreshableCrypto app

creating 464-467

Reminders app 388-400

RemoteConfig app

creating 490-502

remote data

fetching, in SwiftUI 459

RestaurantMenu app 734-739

RhombusApp 283-287

rows

adding, to list 63-66

deleting, from list 67, 68

moving, in List view 71, 72

S

scope modifier 81

ScrollableCharts app 667-675

scrollable content 51

scrolling

programmatically 93-101

ScrollView 51, 92

ScrollViewReader 93

ScrollViewReaders project 93-101

scroll views 51, 52

reference link 55

using 52-55

searchable lists

using 77-81

using, with scopes 81-83

SearchableLists project 77-83

search feature, adding to SwiftUI app

reference link 84

SectionedContacts app 560-563

sectioned list

@SectionedFetchRequest, used for data presentation in 560-563

sections

adding, to list 73-75

displaying, in forms 160-164

hiding, in forms 160-164

Section views 27

SectorMark documentation

reference link 637

SecureField view 9

SF Symbols

reference link 37

using, for simple graphics 34-37

SF Symbols 5 3, 34

reference link 37

shapes

transforming 314-318

ShapeStyle extensions

gradient 3

shadow 3

ShareLink view 3

sheets 207

used, for presenting new views 207-214

- SignInWithApple app**
creating 483-488
working 488
- SignUp project** 160-164
- simple navigation**
with NavigationStack 222-235
- SimpleNavigation app** 222-235
- Slider views** 25
- social login** 504
- SongPlayer app** 381-387
- Spacer** 4, 8
- spring animation**
banner, creating with 318-321
- stacked area charts**
reference link 622
- @State** 361
using, to drive View's behavior 362-365
- State and Data Flow**
reference link 366
- @StateObject** 361
used, for preserving model's life cycle 376-380
- StaticList app** 55-58
- StaticTodoList app** 363-365
- Stepper views** 3, 25
- StopWatch app**
creating 414-419
- stretchable header** 349
implementing 350-354
- StretchableHeader app** 350-354
- Swift Charts** 2, 3
basics 576-584
- SwiftChartsBasics app** 576-584
- SwiftData** 3, 563
documentation link 573
using 563-573
- SwiftDataBasics app** 564-573
- SwiftData, for Core Data app**
reference link 573
- Swift Package Manager (SPM) package** 490
- SwiftUI 1**
features 1
new features 2, 3
- SwiftUI app, based on Combine**
unit testing 446
- SwiftUI apps**
testing, with XCTest 720-732
- SwiftUICoreDataStack app** 537-541
- SwiftUICoreLocation project** 370-375
- SwiftUI view**
Core Data objects, adding from 547-552
Core Data objects, deleting from 556-560
- SwiftUI widgets**
creating 109-121
- SwipeableCards app** 354-360
- swipeable stack of cards**
implementing 354-360
- Symbol Effects** 3
- T**
- Table 3**
multi-column lists, creating with 172-188
reference link 189
- tabular content**
displaying, with LazyHGrid 90-92
displaying, with LazyVGrid 90-92
- TabView** 269
reference link 274
tabs, switching programmatically on 274-276
used, for navigating between multiple
views 269-274
- TabViewWithGestures app** 275, 276
- text**
dealing with 8-12

TextField instances 3
TextField view 9
Text view 8-12
 reference link 12
TheStacks project 4-8
Tic-Tac-Toe game 297
 implementing 297-302
Toggle views 3, 25, 28
toolbars
 reference link 235
Transferable protocol 3
TwoDimensionalLayout project 189-195
typed data-driven navigation
 with NavigationStack 236-252

U

UIImage 13
UIKit
 integrating, into SwiftUI 38-40
 previews, using 149-153
UIKit and SwiftUI, integration
 reference link 40
UIKitToSwiftUI project 38-40
UITableViewController 51
unit testing
 app, based on Combine 446
 benefits 453
untyped data-driven navigation
 with NavigationStack 252-259
User Experience (UX) 159
UsingImages project 13-18
UsingPickers project 25-28
UsingSF Symbols project 34-37
UsingTabViews app 269
UsingViewBuilder project 31-33
UsingViewModifiers project 29, 30

V

view 44-49
 multiple animations, applying to 327-329
 previewing, in NavigationStack 138-142
 previewing, on different devices 146-148
 previewing, with different traits 143-145
ViewBuilder
 presentation, separating from content 31-33
 reference link 34
ViewModifier
 used, for applying groups of styles 28-31
view modifiers
 reference link 31
 VStack 4, 8

W

watchOS version, iOS app
 creating 708-717
Weather app
 creating 428-440
WeScroll project 52-55
WidgetKit
 reference link 121
WorkingWithMarks app 600-622
Worldwide Developer Conference (WWDC) 1

X

Xcode15
 live preview canvas, using 124-137
XCTest
 reference link 734
 used, for testing SwiftUI apps 720-732

Z

ZStack 4, 8

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere? Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily.

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below



<https://packt.link/free-ebook/9781805121732>

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

