EXPERT INSIGHT

SwiftUI Cookbook

Build Modern iOS Apps with Practical Recipes and Hands-On Sample Projects

Fourth Edition



Juan C. Catalan

packt

SwiftUI Cookbook

Fourth Edition

Build Modern iOS Apps with Practical Recipes and Hands-On Sample Projects

Juan C. Catalan



SwiftUI Cookbook

Fourth Edition

Copyright © 2026 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.

Portfolio Director: Ashwin Nair

Relationship Lead: Sohini Ghosh

Project Manager: Ruvika Rao

Content Engineer: Nithya Sadanandan

Technical Editor: Arjun Varma

Copy Editor: Nithya Sadanandan

Indexer: Manju Arasan

Proofreader: Nithya Sadanandan

Production Designer: Vijay Kamble

Growth Lead : Anamika Singh

First published: October 2020
Second edition: November 2021
Third edition: December 2023
Fourth edition: October 2026

Production reference: 1231026

Published by Packt Publishing Ltd.
Grosvenor House
11 St Paul's Square
Birmingham
B3 1RB, UK.

ISBN 978-1-80666-377-4

www.packtpub.com

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

– Juan C. Catalan

Contributors

About the author

Juan C. Catalan is a software engineer with more than 20 years of professional experience. He started mobile development back in the days of iOS 3 and has worked as a professional iOS developer in 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, by mentoring developers, and by reviewing technical books, and he is the author of the third and fourth editions of *SwiftUI Cookbook* . He lives in Austin, Texas, with his wife, Donna, where they spend time with their kids.

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, Stewart and Felipe, for their excellent work reviewing the chapters of this book and for the many suggestions that made it better.

To everyone at Packt Publishing who contributed to this book, with a special thanks to Nithya Sadanandan.

About the reviewers

Stewart Lynch is a retired educator and IT professional. During his 30-year career in education, he served as a teacher, administrator, and IT director before moving to the private sector, where he worked for Canada's largest software company.

Since retiring in 2014, Stewart has focused on iOS and macOS development and sharing his knowledge with others through his YouTube channel, blog posts, and speaking engagements. Stewart has also been reviewing books for Packt Publishing since 2023.

> *"I would like to thank the author of this book, Juan C. Catalan, for being so responsive and receptive to my feedback throughout the review process."*

Felipe O holds a bachelor's degree in computer science and has two years of experience with SwiftUI and MVVM-C. He is currently developing an investment app as a personal project and is also interested in data science. Felipe is passionate about portrait photography and digital image upscaling using ESRGAN.

Preface

SwiftUI is the modern way to build user interfaces for all Apple platforms. Its declarative syntax lets us describe what an interface should look like for any given state, and the framework keeps the screen in sync as that state changes. This book covers the foundations of SwiftUI, from views, layout, and navigation to state management, persistence, networking, drawing, animation, and data visualization, together with the features Apple introduced in iOS 26 and iOS 27, such as the Liquid Glass design, the native web view, rich text editing, reorderable containers, 3D charts, and the Foundation Models framework. This fourth edition is fully updated for iOS 27, Xcode 27, and Swift 6.4, and its text is entirely new. Of its 54 sample apps, only two come from the third edition, and both have been updated for iOS 27.

I have been building iOS apps since the days of iOS 3, and I spent most of those years writing user interfaces in UIKit. I started using SwiftUI as soon as Apple released it in 2019, and I have used it in production apps since 2022. Today I use it in every project I work on, and after seven years of annual updates, I consider it the best way to build apps for Apple platforms, not just the newest one.

Particular attention is given to the interaction between SwiftUI and the rest of an app's code: data flow with the Observation framework and Combine, persistence with SwiftData, networking with async/await, and debugging, profiling, and testing with Xcode and

Instruments. With 61 recipes built around working sample apps and packed with the practices I use in my own projects, you'll learn how to build visually compelling, accessible apps, from simple lists to interactive charts and features powered by Apple Intelligence.

When I planned this edition, I wanted it to work as well for someone writing their first SwiftUI view as for an experienced developer looking for a single recipe. That is why, although it is still a cookbook, I organized its 15 chapters into three progressive parts: the essentials of building views, the flow of data through an app, and the advanced APIs that make an app stand out. If you are new to SwiftUI, you can read the book from beginning to end, and each part builds on the one before it. If you are an experienced developer, you can jump straight to any chapter or recipe, and cross-references point back to the recipes that introduce the concepts it relies on. Each recipe focuses on a small set of APIs and builds a working app around them, so you can understand each concept thoroughly before combining several of them in your own apps. I teach SwiftUI the way I work with it: with the preview canvas open, changing one thing at a time and watching the result. Most recipes follow that rhythm, and it is a habit I hope you will take with you.

This edition also responds to what readers and reviewers told me about the previous ones. In earlier editions, some steps did not compile until a later step added the missing piece; in this edition, every step leaves the project in a state that compiles and runs. Reviewers asked to see at a glance which lines a step adds, so those lines are now set in bold in every snippet that shows code from earlier steps. And for readers who want to check their work, the companion repository now includes step files that show each app at

the milestones of its recipe, something the previous editions did not have.

I chose to focus this edition on iOS. The previous edition covered several platforms, and I have found that developers learn faster in one consistent environment. Once you master SwiftUI on iOS, adapting your skills to iPadOS, macOS, watchOS, tvOS, and visionOS is straightforward, because the framework and its declarative patterns are shared across all Apple platforms.

This edition also arrives at a time when AI is changing how we write code. Xcode 27 builds coding intelligence into the workspace: we can ask a model such as ChatGPT or Claude about our code, or hand a task to an agent that edits files, builds the app, and fixes its own errors. I use AI coding assistants in my own work, including in preparing this book, as I explain in the *Use of AI in the writing of this book* section later in this preface. They have made me faster, but they have not made understanding SwiftUI any less important. Someone still has to decide what to build, judge whether the generated code is correct, current, and accessible, and find out why a view updates at the wrong moment or a list stutters as it scrolls. That someone is you, and the recipes in this book give you the understanding to do it with confidence. AI also appears inside the apps we build: the last chapter adds Apple Intelligence features to SwiftUI apps with the Foundation Models and Image Playground frameworks.

Over the years, I have mentored many developers, and the questions they ask me are rarely about what an API does; they are about when to use it and why one approach is better than another. In writing this cookbook, I wanted to answer those questions the way I would

explain them to a teammate working on the same project. I hope it helps you enjoy building apps with SwiftUI as much as I do.

Who this book is for

This book is for developers who want to learn SwiftUI, whether they are new to Apple platforms or experienced iOS developers moving from UIKit. The book assumes knowledge of the Swift programming language, but no previous experience with SwiftUI. It is also for developers who already build apps with the help of AI coding assistants and want a solid understanding of the SwiftUI code those assistants produce. You'll also find this book to be a helpful reference whenever you need to implement a specific feature in SwiftUI or catch up with the changes Apple introduced in iOS 26 and iOS 27.

What this book covers

`Chapter 1`, *SwiftUI Essentials*, explains the basic building blocks of SwiftUI apps: text, images, and stacks for layout, buttons and navigation with `NavigationStack`, input controls such as pickers, toggles, and sliders together with SF Symbols, and the representable protocols that bring UIKit views and view controllers into SwiftUI.

`Chapter 2`, *Lists, Scroll Views, and Search*, explains how to present collections with `List` and `ScrollView`. You'll build grouped, editable lists, add search with tokens, scopes, and suggestions, implement drag and drop with the `Transferable` protocol, control scrolling with snapping and programmatic positions, and let the user reorder items in any container, a capability new in iOS 27.

`Chapter 3`, *Prototyping and Previews*, explains how to make the preview canvas part of your daily workflow with the `#Preview` macro. You'll create interactive previews, verify views under different traits and environment values, feed previews with reusable mock data, and preview UIKit views and view controllers.

`Chapter 4`, *User Input and View Composition*, explains how to build forms with keyboard focus management, ask the user for decisions with alerts and confirmation dialogs, organize actions with menus, disclosure groups, and toolbars in the Liquid Glass design, and keep growing screens readable by extracting reusable views and view modifiers.

`Chapter 5`, *State and Data Binding*, explains where state lives and how it flows between views, from `@State` and `@Binding` to `@Observable`

models built with the Observation framework, and how the older `ObservableObject` pattern and the Combine framework fit alongside them.

Chapter 6, *App-Wide Settings and Accessibility*, explains how to share models across an app through the environment, with both the Observation framework and `ObservableObject`, how to read the system's environment values and define your own, and how to make your apps usable by everyone with VoiceOver labels, custom actions, rotors, and accessibility focus.

Chapter 7, *Data Persistence and Local Storage*, explains how to pick the right persistence tool for your data: `@AppStorage` and `@SceneStorage` for values that live in a view, `UserDefaults` for values a model owns, SwiftData for records the app queries and relates, and JSON files for data you want to own end to end.

Chapter 8, *Networking and Remote Data*, explains how to map JSON to Swift types with `Codable`, fetch data from a remote API with async/await and with Combine, and download remote images with `AsyncImage`, taking control of their caching when the defaults are not enough.

Chapter 9, *Debugging, Testing, and Performance*, explains how to find out why a SwiftUI view updates, inspect a running app with breakpoints and the view hierarchy debugger, measure and fix slow updates with Instruments, and automate the checks of an app with UI tests.

Chapter 10, *Grids and Layout*, explains how to build layouts beyond stacks: two-dimensional layouts with `Grid`, sortable and

editable data tables with `Table`, and custom containers that conform to the `Layout` protocol, animating between them with `AnyLayout`.

Chapter 11, *Navigation, Sheets, and Deep Linking*, explains how to model navigation as data with `NavigationStack`, `NavigationSplitView`, and `NavigationPath`, build tab-based apps with `TabView` and the Liquid Glass tab bar, present and customize sheets, full-screen covers, and popovers, and route deep links to the right screen.

Chapter 12, *Drawing, Animations, and Visual Effects*, explains how to draw with shapes, paths, and `Canvas`, animate views with animations, transitions, and matched geometry, choreograph multistep motion with phase and keyframe animators, and apply Liquid Glass and material effects.

Chapter 13, *Swift Charts*, explains how to visualize data with Swift Charts, from bar, line, point, and area marks to fully customized axes, scales, and legends. You'll add selection and scrolling to interactive charts, plot functions and large datasets with vectorized plots, and display data in three dimensions with `Chart3D`.

Chapter 14, *Web Content and Rich Text*, explains how to embed web content with the native `WebView` and `WebPage`, build a rich text editor with `TextEditor` and `AttributedString`, render Markdown, use custom fonts, and export SwiftUI views to PDF.

Chapter 15, *On-Device Intelligence in SwiftUI*, explains how to add Apple Intelligence to your SwiftUI apps. You'll generate text and structured data with the Foundation Models framework, let the model call your app's code through tools, send images along with text in multimodal prompts, and create images with Image Playground.

To get the most out of this book

This book has been completely revised for iOS 27, Xcode 27, and Swift 6.4. To build all the apps in this book, you will need:

- A Mac with Apple silicon running macOS Tahoe 26.6 or later
- Xcode 27.0 or later, with the iOS 27 platform installed
- For `Chapter 15`, a Mac running macOS 27 with Apple Intelligence turned on, or an iPhone that supports Apple Intelligence

I wanted anyone with a Mac to be able to follow this book, so you don't need an iPhone for it. Every recipe runs in Device Hub, with one exception: the *Generating Images with Image Playground* recipe in `Chapter 15` needs an iPhone that supports Apple Intelligence, because Image Playground does not run on simulated devices. The other recipes in that chapter use the Mac's own language model, as long as the Mac meets the requirements listed above. An iPhone also makes a difference in the *Instruments and Performance* recipe in `Chapter 9`, because the SwiftUI instrument records nothing on a simulated device, but the recipe explains what a simulated device can still show, and you can download the Instruments recordings behind its numbers and explore them without a device. If you do want to run the apps on your own iPhone, it must run iOS 27, and all you need is your Apple Account: sign in with it in Xcode's settings, and Xcode creates a personal team that signs the apps for your devices. None of the recipes requires a membership in the Apple Developer Program.

The book uses Xcode's preview canvas as the main place to see the result of each step, often in its live mode, where we can interact with

a view while we build it. When a step needs something the preview canvas cannot do, such as a drag interaction or data that must survive a relaunch, we run the app in Device Hub, the window that replaces the standalone Simulator app in Xcode 27. Whenever a step says to run the app, press *Cmd + R* or choose **Product** > **Run** .

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

Use of AI in the writing of this book

I used AI tools throughout the writing of this book, mainly Anthropic's Claude through Claude Code, as an assistant for research, drafting, verification, and production. They made it possible to keep 61 recipes and their companion projects in sync while the book moved from iOS 26 to iOS 27. AI was used in the following ways:

- **Research** : Collecting and summarizing Apple's documentation and WWDC material on the APIs of each recipe, especially the new features of iOS 27. I checked every API the book uses against Apple's documentation and in Xcode.
- **Sample data and code formatting** : Generating the sample data for the apps, refactoring code, and formatting it to fit within 80-character lines on the printed page.
- **Consistency checks** : Comparing every code snippet in the book with the Xcode project it comes from, confirming that each step leaves the project in a state that compiles, and marking the lines that each step adds.
- **Step files** : Generating the Swift files in the companion repository that capture an app at each milestone of a recipe, so you can preview it at that point.
- **Figures** : Capturing many of the screenshots with UI tests that drive the apps on a simulated device, and cropping and stitching them for print.
- **Editorial review** : Analyzing each chapter with more than one model, Claude and GPT, to flag inconsistencies in voice,

terminology, and structure, and to check the text for originality.

- **Production** : Building the pipeline that converts the manuscript into Packt's Word template.

AI did not decide what this book teaches. I chose the topics, designed the apps, created every Xcode project, and reviewed every app before it reached the editor. The chapters were also reviewed by the book's technical reviewers and by Packt's editorial team. Any errors that remain are mine.

Download the example code files

The code bundle for the book is hosted on GitHub at https://github.com/PacktPublishing/SwiftUI-Cookbook-Fourth-Edition. The code is organized by chapter, with one folder per recipe holding its Xcode project. In each project, `ContentView.swift`, together with the `Models` and `Views` folders when the recipe has them, holds the finished app. A few recipes start from a prepared project instead of an empty one; their folders hold a `Starter` project to code along with and a `Complete` project with the finished app.

Almost every project also includes a `Steps` folder with step files, which are snapshots of the app at the milestones of its recipe. Each step file is a complete view with its own `#Preview`, named after the last book step it covers, so `Step08.swift` shows the app as it is at the end of step 8, and a comment at the top of the file lists every book step it covers. Steps that only add a model or change nothing visible share the file of the next milestone. Select a step file in the Project navigator, and the preview canvas shows the app at that point of the recipe. This lets you follow a recipe without typing it, or compare

your own code with the book's when a step does not behave as described. The `Instructions.md` file in the repository explains the structure of the projects and the step files in more detail.

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/9781806663774 .

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: "Computed properties like `headerView` are your first tool for organizing a growing view."

A block of code is set as follows:

```
struct ContentView: View {
  @State private var notificationsEnabled = true
  var body: some View {
    VStack(spacing: 24) {
      headerView
      Toggle("Enable Notifications", isOn: $notificationsEnabled)
        .padding(.horizontal)
    }
    .padding()
```

```
    }
    // ... headerView
  }
```

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

```
struct ContentView: View {
  @State private var notificationsEnabled = true
  var body: some View {
    VStack(spacing: 24) {
      headerView
      Toggle("Enable Notifications", isOn: $notificationsEnabled)
        .padding(.horizontal)
    }
    .padding()
  }
  // ... headerView
}
```

Most steps change code written in earlier steps. To keep each snippet short, it shows the new code in bold with just enough of the existing code to locate it, and a comment that starts with `// ...` stands in for each run of existing code that the step leaves unchanged. The comment names the code it replaces, in the order that code appears in the file. In the following snippet, the three comments stand for the view's state properties, the content of the **Library** tab, and the whole **Radio** tab, so the only line to add is the `.badge()` modifier:

```
struct ContentView: View {
  // ... the state properties
  var body: some View {
    TabView(selection: $selection) {
      Tab("Library", systemImage: "square.stack", value: .library) {
        // ... the Library content
```

```
        }
        .badge(Track.recent.count)
        // ... the Radio tab
      }
    }
  }
```

When you need to see the complete code, the step files described in the *Download the example code files* section show it at each milestone of the recipe.

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

```
ContentView: __hideBought changed.
FilterBarView: @self, _hideBought changed.
hideBought changed from false to true
```

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: "They do not come from our code, and right-clicking one and choosing an option under **Hide Similar Items** filters them out."



Warnings or important notes appear like this.



Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

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

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

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

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

Share your thoughts

Once you've read *SwiftUI Cookbook* , 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.



https://packt.link/r/1806663775

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

Free benefits with your book

This book comes with free benefits to support your learning. Activate them now for instant access (see the " *How to Unlock* " section for instructions).

Here's a quick overview of what you can instantly unlock with your purchase:







How to Unlock

Scan the QR code (or go to packtpub.com/unlock). Search for this book by name, confirm the edition, and then follow the steps on the page.



UNLOCK NOW

Note: Keep your invoice handy. Purchases made directly from Packt don't require one.

Part 1

Getting Started with SwiftUI

In this first part of the book, you will learn the building blocks that every SwiftUI app is made of. You will start with text, images, and stacks, move on to buttons, navigation, and the standard input controls, and bring UIKit views into SwiftUI when the framework has no equivalent. Next, you will present collections with lists and scroll views, add search, drag and drop, and reordering, and make the preview canvas part of your everyday workflow. The part closes with forms, alerts, menus, and toolbars in the Liquid Glass design, and with the reusable views and modifiers that keep a growing screen readable. By the end of this part, you will be able to build complete, interactive screens with SwiftUI.

This part of the book includes the following chapters:

- `Chapter 1` , *SwiftUI Essentials*
- `Chapter 2` , *Lists, Scroll Views, and Search*
- `Chapter 3` , *Prototyping and Previews*
- `Chapter 4` , *User Input and View Composition*

1

SwiftUI Essentials

When I first started building iOS apps, setting up even a simple screen meant writing dozens of lines of boilerplate code or spending hours in Interface Builder. SwiftUI changed that. Instead of telling the system how to construct each element step by step, we describe what we want on screen, and the framework takes care of the rest. After years of annual updates, SwiftUI has matured into the standard way to build apps across Apple's platforms, and I use it in every project I work on.

In practice, SwiftUI's declarative approach means we compose our screens from small, reusable pieces, arranged inside layout containers and fine-tuned with modifiers. When data changes, SwiftUI figures out which parts of the screen need to update and redraws them for us. The recipes in this chapter introduce these core pieces and show how they fit together in working apps.

In this first chapter, we will get hands-on with the essentials. We will start by learning how to display text and images, and how to arrange views using `HStack`, `VStack`, and `ZStack`. From there, we will explore buttons and navigation with `NavigationStack` and `NavigationLink`, discovering how to style buttons and move between screens. We will then dive into input controls such as `Picker`, `Toggle`, `Slider`, `Stepper`, and `DatePicker`, along with SF Symbols (San Francisco Symbols,

named after Apple's San Francisco system font), Apple's library of over 6,000 system icons that scale and color alongside text. Finally, we will learn how to bridge UIKit components into SwiftUI using `UIViewRepresentable` and `UIViewControllerRepresentable`, a technique that opens up the entire UIKit ecosystem from within our SwiftUI apps.

By the end of the chapter, you should feel comfortable with SwiftUI's core building blocks and, more importantly, with the way those pieces fit together when you build real screens.

We will cover these topics in the following recipes:

- Basic Views and Layout
- Buttons and Navigation
- Pickers, Toggles, and SF Symbols
- UIKit Interoperability Basics



Download the code bundle and the PDF version of this book

Your purchase includes a DRM-free PDF copy of this book, the code bundle, and a range of exclusive benefits. To unlock everything, follow the *Free benefits with your book* section in the *Preface* .

Technical requirements

The code in this chapter is based on Xcode 27.0 and iOS 27.0. You can download and install the latest version of Xcode from the App Store. All the code examples for this chapter can be found on GitHub at https://x.swiftuicookbook.com/ch01 .

Basic Views and Layout

The best way to understand SwiftUI is to build something with it. In this first recipe, we will construct a weather profile card from scratch, starting with plain text and progressively adding images, layout containers, modifiers, and a layered background. Along the way, we will see how views nest inside each other to form the screen, and how modifiers control their appearance and sizing.

The finished card displays a city name, a weather summary, temperatures, and an icon, but the real goal is bigger than the finished UI. This small example gives us a compact way to practice composition, nesting, and modifier order without introducing unnecessary complexity.

Getting ready

You will need a new SwiftUI iOS app named **BasicViews** . Follow these steps to create it:

1. Open Xcode and click on the **New Project** button. Alternatively, select **File > New > Project** from the menu bar.
2. From the displayed menu, select **Choose Template…** .
3. Select the **iOS** tab, choose the **App** template, and click **Next** .
4. Set the product name to **BasicViews** , make sure the interface is set to **SwiftUI** and the language is set to **Swift** , and leave **Testing System** and **Storage** set to **None** .
5. Click **Next** , choose a folder for the project (you can leave **Create Git Repository on my Mac** unchecked for these recipes), and click **Create** .

This is the same process you will follow to create an Xcode project for every recipe in this book, unless otherwise noted .

How to do it...

We will build the card the same way I usually teach SwiftUI on a team: start with the smallest visible pieces, verify each step, and only then layer on layout and styling. That approach keeps the view hierarchy understandable from the beginning. These are the steps:

1. Open the `ContentView.swift` file. `ContentView` is the root view Xcode generates for every new app, and the `App` struct displays it on launch, which is why we start here, and why you will meet this name in every recipe; it is as fundamental to a SwiftUI app as the `body` property is to a view. Replace the content of `body` with a `VStack` containing a couple of `Text` views. The `ContentView` should look like the following:

```
struct ContentView: View {
  var body: some View {
    VStack {
      Text("San Francisco")
      Text("Partly Cloudy")
    }
  }
}
```

2. Use the preview canvas in live mode to confirm the baseline layout. At this point, you should see two centered lines of text. I like verifying early, because once modifiers and nested containers start piling up, small mistakes become harder to spot.

You can also check the corresponding step files in the book's GitHub repository to compare your code at each stage.

3. Next, add a bit of hierarchy to the text with font and weight modifiers, so the city reads as the primary element and the weather condition reads as supporting information:

```
VStack {
  Text("San Francisco")
    .font(.title)
    .fontWeight(.bold)
  Text("Partly Cloudy")
    .font(.subheadline)
    .foregroundStyle(.secondary)
}
```

4. Now, let's add an image. Download the `cloud and sun.png` file from the `ch01/assets/` folder in the book's GitHub repository, then drag it into the asset catalog (`Assets.xcassets`) in the Xcode Project Navigator. Add the weather image above the text, so the card starts to feel like a composed interface rather than a loose stack of labels. The string in `Image("cloud and sun")` must match the asset's name exactly, including its spaces and capitalization:

```
VStack {
  Image("cloud and sun")
    .resizable()
    .aspectRatio(contentMode: .fit)
    .frame(width: 120)
  // ... existing Text views
}
```

5. Let's improve the layout by adding spacing and padding. We will also switch to the `ImageResource` -based `Image(.cloudAndSun)`

syntax, which is type-safe and provides compile-time checking for asset names:

```
VStack(spacing: 8) {
  Image(.cloudAndSun)
    .resizable()
    .aspectRatio(contentMode: .fit)
    .frame(width: 120)
  // ... existing Text views
}
.padding()
```

6. With the vertical structure in place, we can bring in an `HStack` to lay out the temperature information horizontally and make the card feel more like a real weather summary:

```
var body: some View {
  VStack(spacing: 8) {
    // ... existing Image and Text views
    HStack(spacing: 16) {
      VStack {
        Text("72°")
          .font(.system(size: 44, weight: .thin))
        Text("Current")
          .font(.caption)
          .foregroundStyle(.secondary)
      }
      VStack {
        Text("68°")
          .font(.system(size: 44, weight: .thin))
        Text("Feels like")
          .font(.caption)
          .foregroundStyle(.secondary)
      }
    }
  }
  .padding()
}
```

7. Add a `Divider` between the city info and the temperature row. A `Divider` draws a thin horizontal line when placed inside a `VStack`:

```
    Text("Partly Cloudy")
      .font(.subheadline)
      .foregroundStyle(.secondary)
    Divider()
    HStack(spacing: 16) {
```

8. Now, expand the temperature row to three columns (**Low** , **Current** , and **High**) and replace the fixed spacing with `Spacer` views, so the columns distribute themselves naturally across the available width. Replace the `HStack` with the following, and delete the `.padding()` modifier after the outer `VStack` . The row now carries its own horizontal and bottom padding, and leaving the `VStack` unpadded lets the header we build in step 10 reach the edges of the card:

```
VStack(spacing: 8) {
  // ... existing Image, Text, and Divider views
  HStack {
    VStack {
      Text("58°")
        .font(.system(size: 44, weight: .thin))
      Text("Low")
        .font(.caption)
        .foregroundStyle(.secondary)
    }
    Spacer()
    VStack {
      Text("72°")
        .font(.system(size: 44, weight: .thin))
      Text("Current")
        .font(.caption)
        .foregroundStyle(.secondary)
    }
```

```
      Spacer()
      VStack {
        Text("79°")
          .font(.system(size: 44, weight: .thin))
        Text("High")
          .font(.caption)
          .foregroundStyle(.secondary)
      }
    }
    .padding(.horizontal)
    .padding(.bottom, 8)
}
```

9. At this point, the content works, but it still floats on the screen. Wrapping it in a rounded background gives it the visual structure of a card and makes the composition feel intentional:

```
VStack(spacing: 8) {
  // ... all our content
}
.background {
  RoundedRectangle(cornerRadius: 20)
    .fill(Color(.systemBackground))
    .shadow(color: .black.opacity(0.15), radius: 10, x: 0,
}
.padding(.horizontal, 24)
```

10. To finish the card, wrap the whole composition in an outer `ZStack` layered over a full-screen background color, and give the header its own `ZStack` : a `LinearGradient` with the weather image on top, and a `location.fill` pin in the bottom-trailing corner. Because several layers now interact, here is the complete `body` :

```
var body: some View {
  ZStack {
    Color(.systemGroupedBackground)
```

```
        .ignoresSafeArea()
      VStack(spacing: 8) {
        ZStack(alignment: .bottomTrailing) {
          ZStack {
            LinearGradient(
              colors: [.blue, .cyan],
              startPoint: .topLeading,
              endPoint: .bottomTrailing
            )
            .frame(height: 140)
            .clipShape(
              UnevenRoundedRectangle(
                topLeadingRadius: 20,
                topTrailingRadius: 20
              )
            )
            Image(.cloudAndSun)
              .resizable()
              .aspectRatio(contentMode: .fit)
              .frame(width: 120)
              .shadow(radius: 3)
          }
          Image(systemName: "location.fill")
            .font(.caption)
            .foregroundStyle(.white.opacity(0.8))
            .padding(10)
        }
        Text("San Francisco")
          .font(.title)
          .fontWeight(.bold)
        Text("Partly Cloudy")
          .font(.subheadline)
          .foregroundStyle(.secondary)
        Divider()
        HStack {
          VStack {
            Text("58°")
              .font(.system(size: 44, weight: .thin))
            Text("Low")
              .font(.caption)
              .foregroundStyle(.secondary)
          }
          Spacer()
```

```
          VStack {
            Text("72°")
              .font(.system(size: 44, weight: .thin))
            Text("Current")
              .font(.caption)
              .foregroundStyle(.secondary)
          }
          Spacer()
          VStack {
            Text("79°")
              .font(.system(size: 44, weight: .thin))
            Text("High")
              .font(.caption)
              .foregroundStyle(.secondary)
          }
        }
        .padding(.horizontal)
        .padding(.bottom, 8)
      }
      .background {
        RoundedRectangle(cornerRadius: 20)
          .fill(Color(.systemBackground))
          .shadow(color: .black.opacity(0.15),
                  radius: 10, x: 0, y: 5)
      }
      .padding(.horizontal, 24)
    }
  }
```

11. With the card in place, spend a minute exploring it in the preview canvas using live mode. Switching between light and dark mode is especially useful here (use the appearance toggle in the preview canvas controls, or add `.preferredColorScheme(.dark)` to the `#Preview` macro), because it shows why semantic colors and system materials are worth using from the start. You will find the complete code in the book's GitHub repository. The following figure shows the finished weather profile card:



Figure 1.1: The final weather profile car d app

How it works...

Every element in our weather card, from the `San Francisco` title to the gradient header, is a `View`. A view is a lightweight struct that describes a piece of the interface. Views do not draw pixels directly, they express *what* should be drawn; instead, SwiftUI reads our view hierarchy and decides *how* to render it. This is why we can nest views

freely without worrying about performance: each one is a description, not a drawing command.

We used three layout containers in this recipe:

- `VStack` arranges children vertically. We used an optional `spacing` parameter (`VStack(spacing: 8)`) to have precise control over the gap between children.
- `HStack` arranges children horizontally. We used it for the temperature row.
- `ZStack` layers children on the z-axis. We used it to overlay the weather icon on the gradient and to place the card on a colored background. The `alignment` parameter controls positioning.

Our weather card illustrates how stacks compose through nesting. The outer `VStack` contains an `HStack` , which itself holds three `VStack` columns for the temperature data. The header is built the same way: a `ZStack` overlays the weather image on a `LinearGradient` , and that entire header sits inside another `ZStack` with `.bottomTrailing` alignment to position the location icon on top. This layered nesting is how we express view hierarchy in SwiftUI.

For two-dimensional layouts, see the *Grid and GridRow* section of *Chapter 10* . For fully custom layout algorithms, see the *Custom Layouts* section of *Chapter 10* .

`Spacer` and `Divider` round out the layout toolkit. `Spacer` expands to fill available space; we used two of them to distribute the temperature columns evenly across the row. `Divider` draws a thin line that adapts its orientation to its container (vertical inside an `HStack` , horizontal inside a `VStack`).

We customize views by chaining **modifiers** . Order matters here because each modifier wraps what came before it and produces a new view. A common surprise for newcomers: `.padding().background(.blue)` on a `Text` fills the padded area with blue, while `.background(.blue).padding()` fills only the text and leaves the padding transparent. Key modifiers we used:

- `.font()` with semantic styles (`.title` , `.subheadline` , `.caption`) that adapt to Dynamic Type. For precise sizing, we used `.font(.system(size: 44, weight: .thin))` . See the *Accessibility APIs* section of *Chapter 6* for more on accessible text.
- `.foregroundStyle(.secondary)` for visual hierarchy. SwiftUI defines primary, secondary, tertiary, and quaternary styles that adapt to light and dark mode.
- `.padding()` for spacing. Without arguments it uses the system default; with parameters like `.padding(.horizontal, 24)` we control direction and amount. Our card uses double-padding: inner `.padding()` between content and card edges, outer `.padding(.horizontal, 24)` between card and screen edges.
- `.resizable()` and `.aspectRatio()` for images, which must come in this order to work correctly.
- `.background()` with the closure syntax for card styling, `.frame()` for size constraints, and `.clipShape()` masks a view to a given shape. We combined it with `UnevenRoundedRectangle` to round only the top corners of the header gradient. To create your own custom shapes or combine built-in ones, see the *Shapes, Paths, and Drawing* section of *Chapter 12* .



When using asset catalog images, always apply the `.resizable()` modifier before `.frame()` or `.aspectRatio()`. Without `.resizable()`, the image will display at its original size, ignoring any size constraints.

When you drag an image into the asset catalog, Xcode creates an *image asset*. You can refer to it in two ways. The string-based `Image("cloud and sun")` matches the asset by its exact name, including capitalization and spaces, so a typo surfaces only as a missing image at runtime. Xcode also generates a type-safe `ImageResource` for every asset, stripping the spaces and converting the name to camel case, so `cloud and sun` becomes `cloudAndSun`. Typing `Image(.)` then lists every available asset for you to choose from. Prefer the type-safe `Image(.cloudAndSun)` form over `Image("cloud and sun")` whenever you can. In practice, this is the kind of small choice that pays off later: you get autocomplete, compile-time validation, and fewer silent runtime failures when assets change. For system icons, `Image(systemName:)` loads icons from SF Symbols. To know more, see *Pickers, Toggles, and SF Symbols*.

We used **semantic colors** like `Color(.systemBackground)` and `Color(.systemGroupedBackground)` throughout. These dynamic colors adapt to light mode, dark mode, and high-contrast settings automatically, so always prefer them over hard-coded values. To learn about previewing with different device settings, see the *Trait Variations and Device Simulations* section of *Chapter 3*. For more on the preview canvas and live previews, check out the *Live Previews with #Preview* section of *C hapter 3*.

See also

- `Text` view documentation: https://x.swiftuicookbook.com/hmwz7
- `Image` view documentation: https://x.swiftuicookbook.com/7y1jw
- `HStack` documentation: https://x.swiftuicookbook.com/dda52
- `VStack` documentation: https://x.swiftuicookbook.com/42oke
- `ZStack` documentation: https://x.swiftuicookbook.com/vyrc5
- View modifiers documentation: https://x.swiftuicookbook.com/yqm4n
- SF Symbols app: https://x.swiftuicookbook. com/0i5i1

Buttons and Navigation

Most apps need two things early on: a way to respond to taps and a way to move between screens. In SwiftUI, `Button` handles the first, while `NavigationStack` and `NavigationLink` handle the second. This recipe covers both, along with the styling and role system that lets us control how buttons look and what they communicate to the user.

In this recipe, we will build a small app named **ButtonShowcase** that uses a navigation menu to present three focused detail screens: styles, roles and tinting, and border shapes. Keeping each topic isolated makes the behavior of the button APIs easier to see and easier to remember.

If this is your first time working with SwiftUI, we recommend reading the *Basic Views and Layout* section before starting this recipe, as we will use `VStack`, `HStack`, `Text`, `Image(systemName:)`, `Spacer`, `Divider`, and several view modifiers throughout.

Getting ready

Create a new SwiftUI iOS app named **Butt onShowcase**.

How to do it...

We will start with the main screen and its navigation structure, then build each destination screen one at a time. That keeps the structure of the app visible from the beginning, which is especially helpful when using navigation. Let's get started:

1. Open the `ContentView.swift` file and replace the entire `ContentView` struct with the following. We create a `NavigationStack` as the root container, with a `List` of three `NavigationLink` items inside it. Each link uses an `HStack` label with an SF Symbol image and a text description. For now, we use a placeholder `Text` view as the destination for each link:

```
struct ContentView: View {
  var body: some View {
    NavigationStack {
      List {
        NavigationLink {
          Text("Coming soon")
        } label: {
          HStack(spacing: 12) {
            Image(systemName: "paintbrush.fill")
              .foregroundStyle(.blue)
```

```
            Text("Button Styles")
          }
        }
        NavigationLink {
          Text("Coming soon")
        } label: {
          HStack(spacing: 12) {
            Image(systemName: "theatermasks.fill")
              .foregroundStyle(.purple)
            Text("Roles & Tints")
          }
        }
        NavigationLink {
          Text("Coming soon")
        } label: {
          HStack(spacing: 12) {
            Image(systemName: "square.on.circle")
              .foregroundStyle(.orange)
            Text("Border Shapes")
          }
        }
      }
      .navigationTitle("Buttons & Nav")
    }
  }
}
```

2. Run the preview to confirm the navigation skeleton works before moving on. You should see three rows, each with an icon, a label, and the standard disclosure chevron that tells the user the row pushes to a detail screen. The detail screen will show the placeholder **Coming soon** text and a back button in the top-left corner. Alternatively, you can check the preview of the step files in the book's GitHub repository.

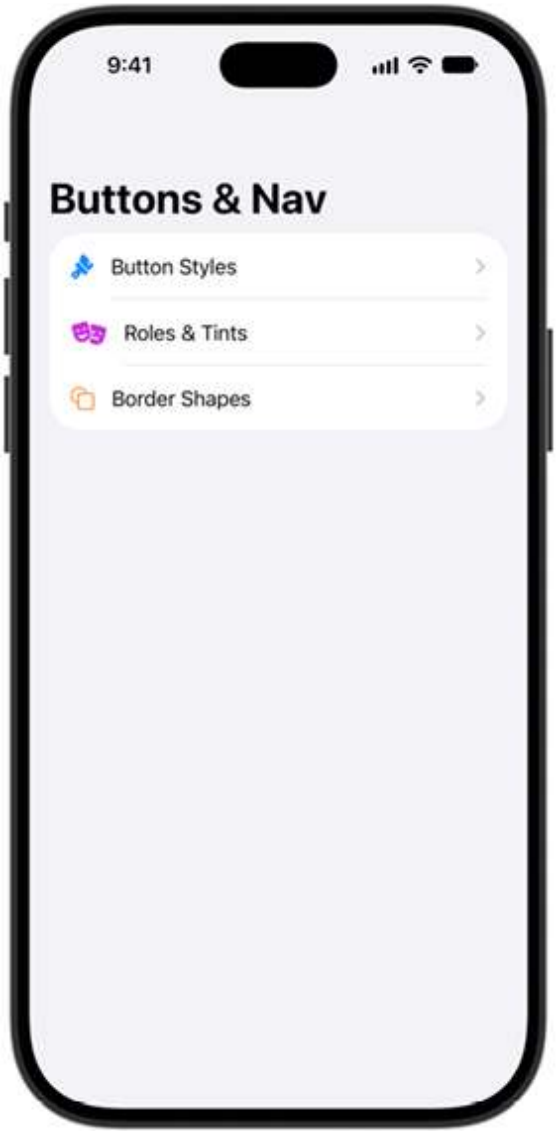


Figure 1.2: The main navigation screen with three options to choose from

3. Build the first destination screen, that will give us a compact playground for comparing the built-in button styles side by side. Create a new SwiftUI View file (**File** > **New** > **File** , or *Cmd* + *N* , then select **SwiftUI View**) named `ButtonStylesView.swift` . This view will display all seven built-in button styles and let the user tap each one to see its name. Replace the generated code with the following:

```
import SwiftUI
struct ButtonStylesView: View {
  @State private var selectedStyle: String?
  var body: some View {
    VStack(spacing: 20) {
      Text("Each button uses a different built-in style.")
        .font(.subheadline)
        .foregroundStyle(.secondary)
        .padding(.horizontal)
      Divider()
      VStack(spacing: 16) {
        Button("Automatic") { selectedStyle = "Automatic" }
          .buttonStyle(.automatic)
        Button("Bordered") { selectedStyle = "Bordered" }
          .buttonStyle(.bordered)
        Button("Bordered Prominent") {
          selectedStyle = "Bordered Prominent"
        }
          .buttonStyle(.borderedProminent)
        Button("Borderless") { selectedStyle = "Borderless"
          .buttonStyle(.borderless)
        Button("Plain") { selectedStyle = "Plain" }
          .buttonStyle(.plain)
        Button("Glass") { selectedStyle = "Glass" }
          .buttonStyle(.glass)
        Button("Glass Prominent") {
          selectedStyle = "Glass Prominent"
        }
          .buttonStyle(.glassProminent)
      }
      .padding()
      Divider()
      if let selectedStyle {
        Text("Style: \(selectedStyle)")
      } else {
        Text("Tap a button to see its style here")
      }
      Spacer()
    }
    .navigationTitle("Button Styles")
    .navigationBarTitleDisplayMode(.inline)
  }
}
```

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

Now, go back to `ContentView.swift` and update the first `NavigationLink` by replacing its `Text("Coming soon")` destination with `ButtonStylesView()`.

4. With the preview canvas in live mode, tap on **Button Styles** to open the detail screen. Seeing the styles stacked together is useful because the visual differences become obvious immediately, especially between `.borderedProminent`, `.plain`, and `.borderless`.
5. Next, create `ButtonRolesView.swift` to explore two ideas that matter a lot in real interfaces: semantics and emphasis. Roles tell the system what kind of action a button represents, while tinting helps communicate visual priority. Replace the generated code with the following:

```
import SwiftUI
struct ButtonRolesView: View {
  @State private var roleIcon: String?
  @State private var selectedColor: Color?
  var body: some View {
    VStack(spacing: 20) {
      Text("Roles give semantic meaning. Tints change accer
        .font(.subheadline)
        .foregroundStyle(.secondary)
        .padding(.horizontal)
      Divider()
      VStack(alignment: .leading, spacing: 16) {
        Text("Roles")
          .font(.headline)
```

```
        HStack(spacing: 12) {
          Button(role: .destructive) { roleIcon = "trash" }
            .buttonStyle(.bordered)
          Button(role: .cancel) { roleIcon = "xmark.circle"
            .buttonStyle(.bordered)
          Button(role: .close) { roleIcon = "xmark.circle.f
            .buttonStyle(.bordered)
          if let roleIcon {
            Image(systemName: roleIcon)
              .foregroundStyle(.secondary)
          }
        }
        Divider()
        Text("Tint Colors")
          .font(.headline)
        HStack(spacing: 12) {
          Button("Blue") { selectedColor = .blue }
            .buttonStyle(.borderedProminent)
            .tint(.blue)
          Button("Green") { selectedColor = .green }
            .buttonStyle(.borderedProminent)
            .tint(.green)
          Button("Orange") { selectedColor = .orange }
            .buttonStyle(.borderedProminent)
            .tint(.orange)
          Button("Purple") { selectedColor = .purple }
            .buttonStyle(.borderedProminent)
            .tint(.purple)
        }
        if let selectedColor {
          Image(systemName: "paintbrush.fill")
            .font(.system(size: 48))
            .foregroundStyle(selectedColor)
        }
      }
      .padding()
      Spacer()
    }
    .navigationTitle("Roles & Tints")
    .navigationBarTitleDisplayMode(.inline)
  }
}
#Preview {
```

```
    NavigationStack {
      ButtonRolesView()
    }
  }
```

Go back to `ContentView.swift` and update the second `NavigationLink` by replacing its `Text("Coming soon")` destination with `ButtonRolesView()` .

6. Back in the preview canvas, navigate to **Roles & Tints** . Notice that the **Delete** , **Cancel** , and **Close** buttons display their titles without us setting any label: each role supplies a standard, localized title, and `.destructive` automatically renders in red. Tap **Delete** to see a trash icon appear next to the buttons, then tap **Cancel** or **Close** to see it change. In the **Tint Colors** section, tap any color button to see a large paintbrush icon appear below, tinted with that color. That icon swap is driven by our own `roleIcon` state, not by the roles themselves: a role changes a button's appearance and meaning, but what happens on tap is entirely up to our code.
7. The last destination screen focuses on border shape customization. This is a smaller API surface, but it is worth seeing in isolation because shape meaningfully changes the character of a button. Create `ButtonShapesView.swift` and replace the generated code with the following:

```
import SwiftUI
struct ButtonShapesView: View {
  @State private var icon: String?
  var body: some View {
    VStack(spacing: 20) {
      Text("Border shapes control the outline of bordered b
```

```
          .font(.subheadline)
          .foregroundStyle(.secondary)
          .padding(.horizontal)
        Divider()
        VStack(spacing: 24) {
          VStack(spacing: 4) {
            Button("Capsule") { icon = "capsule" }
              .buttonStyle(.borderedProminent)
              .buttonBorderShape(.capsule)
            Text(".capsule")
              .font(.caption)
              .foregroundStyle(.secondary)
          }
          VStack(spacing: 4) {
            Button("Rounded Rectangle") { icon = "rectangle"
              .buttonStyle(.borderedProminent)
              .buttonBorderShape(.roundedRectangle)
            Text(".roundedRectangle")
              .font(.caption)
              .foregroundStyle(.secondary)
          }
          VStack(spacing: 4) {
            Button("Custom Radius") { icon = "button.roundedt
              .buttonStyle(.borderedProminent)
              .buttonBorderShape(.roundedRectangle(radius: 4)
            Text(".roundedRectangle(radius: 4)")
              .font(.caption)
              .foregroundStyle(.secondary)
          }
          VStack(spacing: 4) {
            Button {
              icon = "plus"
            } label: {
              Image(systemName: "plus")
            }
            .buttonStyle(.borderedProminent)
            .buttonBorderShape(.circle)
            Text(".circle")
              .font(.caption)
              .foregroundStyle(.secondary)
          }
        }
        .padding()
```

```
        Divider()
        if let icon {
          Image(systemName: icon)
            .font(.system(size: 48))
            .foregroundStyle(.blue)
        }
        Spacer()
      }
      .navigationTitle("Border Shapes")
      .navigationBarTitleDisplayMode(.inline)
    }
}
#Preview {
  NavigationStack {
    ButtonShapesView()
  }
}
```

Go back to `ContentView.swift` and update the third `NavigationLink` by replacing its `Text("Coming soon")` destination with `ButtonShapesView()`.

8. With all three destination views wired in, run through the app end to end and verify that each navigation path behaves as expected. At this point, the sample is less about complexity and more about building intuition for how SwiftUI composes navigation and button styling. If you get stuck, see the book's GitHub repository for the complete working code.

   We have finished building our **ButtonShowcase** app. Play with the app in the live preview or run it in Device Hub to navigate between the different screens and explore all the button variations. Device Hub, the Xcode window that runs apps on simulated devices, is introduced in the *Live Previews with #Preview* section of *Chapter 3*. The following screenshots

show the three destination screens: Button Styles, Roles & Tints, and Border Shapes.

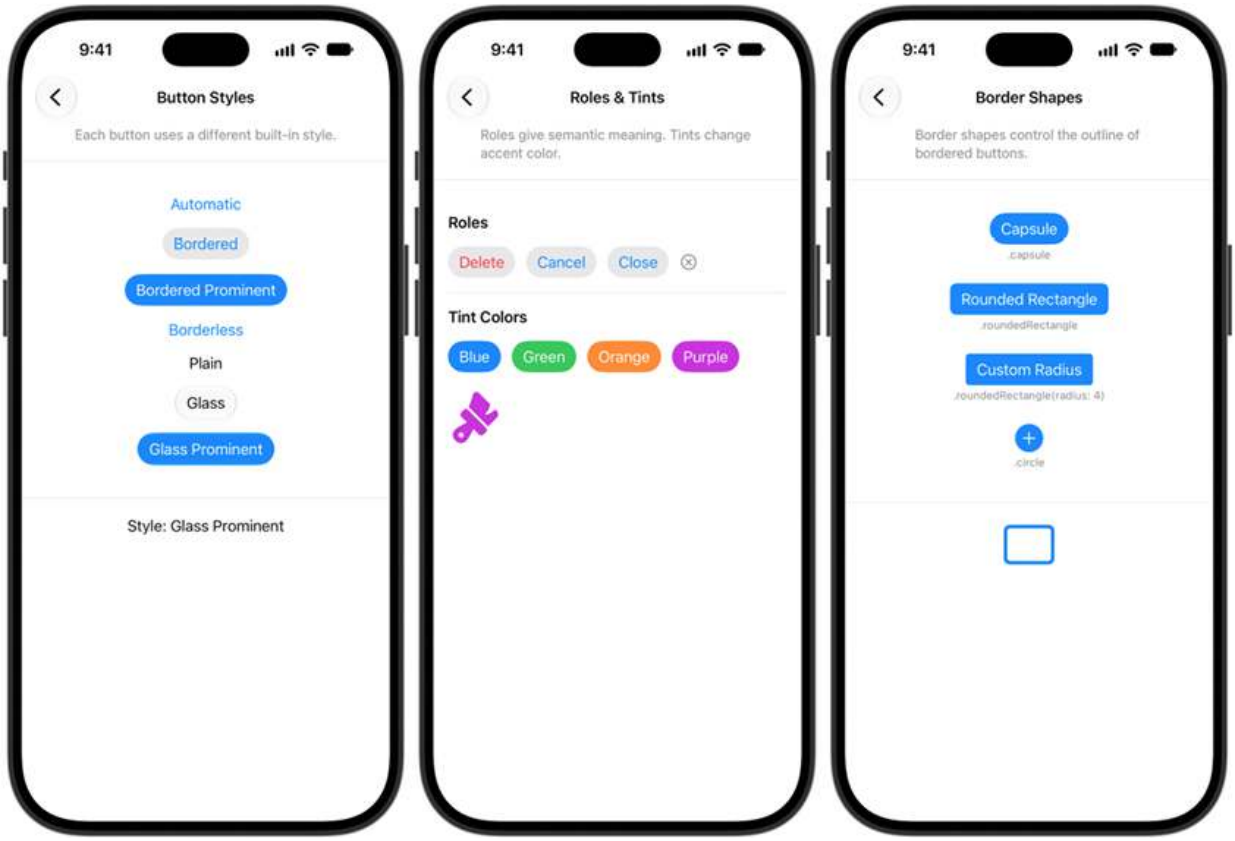


Figure 1.3: The three destination screens

How it works...

In our app, `NavigationStack` acts as the root container. It manages a stack of screens: the `List` sits at the bottom, and each `NavigationLink` pushes a destination view onto the stack when tapped. SwiftUI adds a back button and a disclosure chevron automatically. Notice that `.navigationTitle()` is applied to the `List`, not the `NavigationStack`, because each screen in the stack can set its own title. The `.navigationBarTitleDisplayMode(.inline)` modifier on the detail screens switches to a compact centered title, leaving more room for content.

This recipe uses view-driven navigation, where each `NavigationLink` embeds its destination directly. For programmatic navigation, including deep linking and state restoration, SwiftUI provides `NavigationLink(value:)` combined with `.navigationDestination(for:)` (see the *NavigationStack and NavigationPath* section of *Chapter 11*). To learn more about `List`, see the *Lists and Sections* section of *Chapter 2*.

All three destination screens use `@State` properties to track which button the user last tapped. When a `@State` value changes, SwiftUI automatically re-renders only the views that depend on it, so tapping a button instantly updates the label or icon at the bottom. SwiftUI's dependency tracking knows which parts of the view hierarchy read which state properties, so updates are efficient even with complex views. We will explore `@State`, `@Binding`, and the relationship between state and view updates in the *State and Binding Basics* section of *Chapter 5*. For sharing state across multiple views, see the *Sharing Data with the Environment Using Observation* section of *Chapter 6*.

The `.buttonStyle()` modifier controls visual treatment, with seven built-in options. In most apps, `.bordered` and `.borderedProminent` handle the bulk of your button styling needs. Use `.glass` and its high-emphasis companion `.glassProminent` when you're building for iOS 26 or later and want your buttons to feel native to the Liquid Glass design language (see the *Glass and Material Effects* section of *Chapter 12*). The remaining options `.automatic` (context-dependent), `.borderless` (no chrome), and `.plain` (fully stripped), are more situational but worth trying in the live preview to see how they compare.

Button roles (`.destructive` , `.cancel` , and the iOS 26 `.close`) give buttons semantic meaning that SwiftUI reflects through color and weight; when we omit the label, the role even supplies a standard, localized title, as our **Roles** buttons do. A destructive button renders in red to warn the user of the action's consequences. These roles play a critical part in confirmation dialogs (see the *Alerts and Confirmation Dialogs* section of *Chapter 4*). The `.tint()` modifier overrides the accent color for bordered and prominent buttons and propagates through the view hierarchy, so applying `.tint(.purple)` to a container tints all buttons inside it. The `.buttonBorderShape()` modifier controls the outline shape (`.capsule` , `.roundedRectangle` , `.roundedRectangle(radius:)` , `.circle`) and only affects buttons with visible borders. The `.circle` shape is designed for icon-only buttons, which use the `Button { action } label: { view }` closure syntax for full control over the label's appearance.

See also

- `Button` documentation: https://x.swiftuicookbook.com/7a3ak
- `NavigationStack` documentation: https://x.swiftuicookbook.com/1ocv4
- `NavigationLink` documentation: https://x.swiftuicookbook.com/a4a8u
- `buttonStyle(_:)` modifier: https://x.swiftuicookbook.com/u9cbk
- The SwiftUI cookbook for navigation (WWDC 2022): https://x.swiftuicookbook.com/davd8

- Build a SwiftUI app with the new design (WWDC 2025): https://x.s wiftuicookbook.com/wliha

Pickers, Toggles, and SF Symbols

Settings screens and forms are where users configure an app. SwiftUI has a dedicated control for each kind of input: selections, on/off switches, continuous values, discrete increments, and dates. They all follow the same pattern: bind to a `@State` property, and the control stays in sync with the data. This recipe also covers SF Symbols, the system icon library that scales with text and responds to the same color and styling modifiers we use on views.

We will build an app called **ControlsShowcase** around a settings-style `Form`. The first half of the recipe focuses on input controls and state binding. The second half uses the same screen to explore SF Symbols rendering, variants, and animation effects.

If this is your first time working with SwiftUI, we recommend reading the *Basic Views and Layout* and *Buttons and Navigation* sections before starting this recipe, as we will use concepts like `VStack`, `HStack`, `Label`, view modifiers, and `NavigationStack` throughout. If you want to learn more about forms, sections and textual user input, check out the *Forms* section of *Chapter 4*.

Getting ready

Create a new SwiftUI iOS app named **ControlsShowcase**.

How to do it...

Rather than introducing each control in isolation, we will place them inside a form that feels like a real settings screen. That gives the controls more natural context and makes the examples easier to transfer into production code. These are the steps:

1. Open the `ContentView.swift` file. We will create a `Form` inside a `NavigationStack` and add a first section with three `Toggle` controls. Each toggle uses a `Label` with an SF Symbol to display an icon alongside the text. `Label` is a built-in view that pairs a title with a system image, and many controls accept it as their content. Replace the `ContentView` struct with the following code:

```
struct ContentView: View {
  @State private var notificationsEnabled = true
  @State private var darkModeEnabled = false
  @State private var soundEnabled = true
  var body: some View {
    NavigationStack {
      Form {
        Section("Preferences") {
          Toggle(isOn: $notificationsEnabled) {
            Label("Notifications",
                  systemImage: "bell.fill")
          }
          Toggle(isOn: $darkModeEnabled) {
            Label("Dark Mode",
                  systemImage: "moon.fill")
          }
          Toggle(isOn: $soundEnabled) {
            Label("Sound",
                  systemImage: "speaker.wave.2.fill")
          }
        }
      }
      .navigationTitle("Settings")
    }
  }
}
```

2. Run the preview to confirm the base form renders correctly, then enhance the toggle rows with richer SF Symbol behavior. This is a good example of how small visual touches can make state changes feel much clearer. Replace the `Section("Preferences")` with the following:

```
Section("Preferences") {
  Toggle(isOn: $notificationsEnabled) {
    Label("Notifications", systemImage: "bell")
      .symbolVariant(notificationsEnabled ? .none : .slash)
      .symbolVariant(.fill)
  }
  Toggle(isOn: $darkModeEnabled) {
    Label("Dark Mode",
          systemImage: darkModeEnabled
            ? "moon.fill" : "sun.max.fill")
  }
  Toggle(isOn: $soundEnabled) {
    Label {
      Text("Sound")
    } icon: {
      Image(systemName: soundEnabled
        ? "speaker.wave.2.fill"
        : "speaker.slash.fill")
        .contentTransition(
          .symbolEffect(.replace))
    }
  }
}
```

   Use the live preview again and toggle each switch to watch the bell, moon, and speaker symbols respond to the change.

3. Now, let's add selection controls. Add the following `Theme` enum, `themeColors` array, and state properties above the `body`:

```
@State private var selectedTheme = Theme.system
@State private var selectedColor: Color = .blue
enum Theme: String, CaseIterable {
  case light = "Light"
  case dark = "Dark"
  case system = "System"
}
let themeColors: [Color] = [
  .blue, .purple, .green, .orange, .pink
]
```

Then, add a new **Appearance** section after **Preferences** . It contains a pair of selection controls: a segmented picker for a compact set of mutually exclusive choices, and a palette-style picker inside a menu for accent color:

```
Section("Appearance") {
  Picker("Theme", selection: $selectedTheme) {
    ForEach(Theme.allCases, id: \.self) { theme in
      Text(theme.rawValue).tag(theme)
    }
  }
  .pickerStyle(.segmented)
  Menu {
    Picker("Accent Color", selection: $selectedColor) {
      ForEach(themeColors, id: \.self) { color in
        Image(systemName: "circle.fill")
          .tint(color)
          .tag(color)
      }
    }
    .pickerStyle(.palette)
  } label: {
    HStack {
      Text("Select color 🌈 -> ")
      Image(systemName: "circle.fill")
        .foregroundStyle(selectedColor)
    }
  }
```

```
    .menuActionDismissBehavior(.enabled)
}
```

4. Let's add a `Slider` for continuous value selection and a `Stepper` for discrete increments. Add these state properties above the `body`:

    ```
    @State private var volume: Double = 0.7
    @State private var badgeCount = 3
    ```

    Then, add the following two sections to the form:

```
Section("Audio") {
  VStack(alignment: .leading) {
    Text("Volume: \(Int(volume * 100))%")
      .font(.subheadline)
    Slider(value: $volume) {
      Text("Volume")
    } minimumValueLabel: {
      Image(systemName: "speaker.fill")
    } maximumValueLabel: {
      Image(systemName: "speaker.wave.3.fill")
    }
  }
}
Section("Badge") {
  Stepper(value: $badgeCount, in: 0...9) {
    HStack {
      Image(systemName: "app.badge.fill")
        .symbolRenderingMode(.palette)
        .foregroundStyle(.white, .red)
      Text("Count: \(badgeCount)")
    }
  }
}
```

5. Add a `DatePicker` to the form. This control lets users select dates and times using a calendar or wheel interface. Add the following state property above the `body`:

```
@State private var reminderDate = Date()
```

   Then, add a new section:

```
Section("Reminder") {
  DatePicker("Date", selection: $reminderDate,
             displayedComponents: .date)
    .datePickerStyle(.graphical)
}
```

6. Add a section that renders the same symbol in multiple rendering modes side by side. Seeing them together is the fastest way to understand what each mode actually contributes visually:

```
Section("Symbol Rendering Modes") {
  HStack(spacing: 16) {
    VStack {
      Image(systemName: "cloud.sun.fill")
        .font(.title)
        .symbolRenderingMode(.monochrome)
        .foregroundStyle(.blue)
      Text("Mono")
        .font(.caption2)
    }
    VStack {
      Image(systemName: "cloud.sun.fill")
        .font(.title)
        .symbolRenderingMode(.hierarchical)
        .foregroundStyle(.blue)
      Text("Hierarchical")
        .font(.caption2)
    }
```

```
      VStack {
        Image(systemName: "cloud.sun.fill")
          .font(.title)
          .symbolRenderingMode(.palette)
          .foregroundStyle(.gray, .yellow)
        Text("Palette")
          .font(.caption2)
      }
      VStack {
        Image(systemName: "cloud.sun.fill")
          .font(.title)
          .symbolRenderingMode(.multicolor)
        Text("Multicolor")
          .font(.caption2)
      }
      VStack {
        Image(systemName: "cloud.sun.fill")
          .font(.title)
          .symbolRenderingMode(.multicolor)
          .symbolColorRenderingMode(.gradient)
        Text("Gradient")
          .font(.caption2)
      }
    }
    .frame(maxWidth: .infinity)
  }
```

7. Now, let's add some symbol effects. Add the following state properties above the `body`:

```
@State private var bellBounce = 0
@State private var isAnimating = false
@State private var isMuted = false
```

Finish the screen with symbol effects that demonstrate three different interaction patterns: one-shot feedback, ongoing animation, and animated symbol replacement. Add these two sections:

```
Section("Symbol Effects") {
  HStack(spacing: 20) {
    Button { bellBounce += 1 } label: {
      Image(systemName: "bell.fill")
        .font(.title)
        .symbolEffect(.bounce,
                      value: bellBounce)
        .foregroundStyle(.orange)
    }
    .buttonStyle(.plain)
    Image(systemName: "bell.badge")
      .font(.title)
      .symbolEffect(.wiggle,
                    isActive: isAnimating)
    Image(systemName: "arrow.clockwise")
      .font(.title)
      .symbolEffect(.rotate,
                    isActive: isAnimating)
    Image(systemName: "record.circle")
      .font(.title)
      .foregroundStyle(.red)
      .symbolEffect(.breathe,
                    isActive: isAnimating)
    Image(systemName: "checkmark.circle")
      .font(.title)
      .foregroundStyle(.green)
      .symbolEffect(.drawOn,
                    isActive: isAnimating)
  }
  .frame(maxWidth: .infinity)
  .padding(.vertical, 8)
  Toggle("Animate", isOn: $isAnimating)
}
Section("Magic Replace") {
  HStack {
    Image(systemName: isMuted
      ? "speaker.slash.fill"
      : "speaker.wave.2.fill")
      .font(.system(size: 44))
      .contentTransition(
        .symbolEffect(
          .replace.magic(fallback: .downUp)))
    Spacer()
```

```
      Button(isMuted ? "Unmute" : "Mute") {
        withAnimation { isMuted.toggle() }
      }
      .buttonStyle(.bordered)
    }
    .padding(.vertical, 8)
  }
```

8. Once the screen is complete, interact with every control and pay attention to how the state changes are reflected visually. This recipe is really about two things working together: SwiftUI bindings on the data side and SF Symbols on the presentation side. Refer to the book's GitHub repository if you want to compare your code with the final version. The following figure show s the completed app:



Figure 1.4: The completed ControlsShowcase app

How it works...

`Form` gives our screen the grouped, rounded-section look familiar from the Settings app, and we didn't need to configure that styling ourselves. Every input control in the form follows the same pattern: we pass it a `@State` property with the `$` prefix, and the control reads from and writes to that property automatically. We will explore

`@State` and bindings in the *State and Binding Basics* section of *Chapter 5*. Keep in mind that a control only updates its bound state; acting on that value is up to us. In this showcase, for instance, the **Appearance** controls demonstrate the picker and toggle APIs and their bindings rather than restyling the app itself. Wiring a theme selection to an actual color scheme with `.preferredColorScheme` is covered in the *Trait Variations and Device Simulations* section of *Chapter 3*.

`Picker` adapts its appearance to the style we choose: among many options we used `.segmented` for a small set of options, `.palette` for compact visual selections inside a `Menu`. By default, a `Picker` inside a `Menu` keeps the menu open after each selection so the user can try different options. We added `.menuActionDismissBehavior(.enabled)` to the `Menu` so it dismisses immediately after a tap, which is a better fit for a single-choice control like our accent color picker.

`DatePicker` has its own styles: `.graphical` shows a full calendar, and `.compact` shows a button that expands inline. The `.tag()` modifier on each picker option tells SwiftUI how to match the selection binding to each view. `Slider` provides continuous value selection with optional `minimumValueLabel` and `maximumValueLabel` closures; its label closure is for accessibility only (VoiceOver reads it, but it is not visible). `Stepper` provides discrete increment/decrement, with `in: 0...9` clamping the value to a valid range.

SF Symbols give us access to thousands of icons through `Image(systemName:)`, and because they behave like text, they scale with `.font()` and pick up color from `.foregroundStyle()`. The four rendering

modes (set via `.symbolRenderingMode()`) each offer a different level of control over how color is applied:

- `.monochrome` applies a single color uniformly. Best for toolbars and text-adjacent icons.
- `.hierarchical` uses a single base color with automatic opacity layers for depth.
- `.palette` lets us specify explicit colors per layer via `.foregroundStyle()` with up to three colors. This is how we gave the badge icon a white foreground on a red background.
- `.multicolor` uses the symbol's built-in color scheme. No color configuration needed.

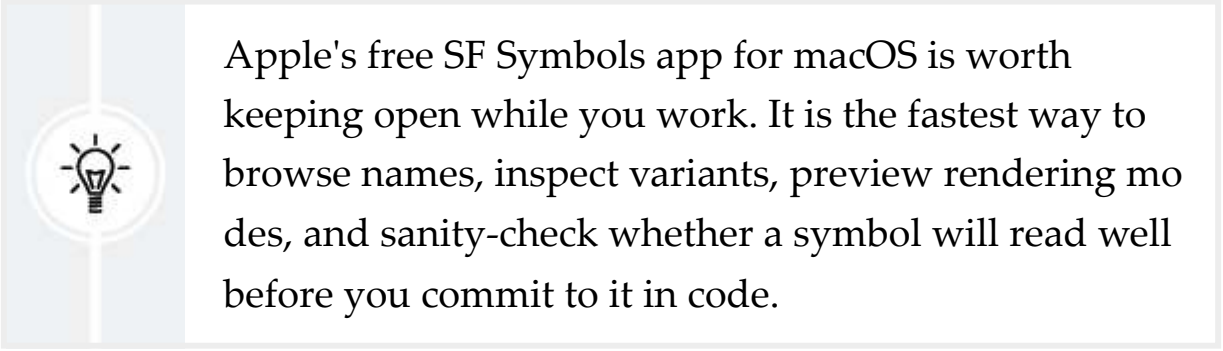
Separately, `.symbolColorRenderingMode(.gradient)` automatically generates a smooth linear gradient from a single source color, adding a sense of lighting and depth. This works alongside any of the four rendering modes above.

The `.symbolVariant()` modifier adds variants like `.slash` and `.fill` programmatically. Multiple calls stack: in our Notifications toggle, we use `"bell"` as the base symbol and apply `.symbolVariant(.fill)` alongside a conditional `.symbolVariant(.slash)`, so the icon shows a filled bell or a filled slashed bell depending on state. This avoids hardcoding multiple symbol names and lets SwiftUI animate the transition between variants.

When we tapped the orange bell icon in our app, it bounced once and stopped. That is a **discrete effect** . When we flipped the **Animate** toggle, the other icons started wiggling, rotating, breathing, and drawing continuously. Those are **indefinite effects** . Discrete effects like `.bounce` fire a one-shot animation each time their `value:`

parameter changes, making them ideal for tap feedback. Indefinite effects like `.wiggle`, `.rotate`, `.breathe`, and `.drawOn` run as long as `isActive` is `true`, useful for signaling ongoing states like "connected" or "recording". The draw effect animates a symbol's appearance stroke by stroke, and can also serve as a view transition with `.transition(.symbolEffect(.drawOn))`.

The three icon-swapping techniques in step 2 each suit different situations: `.symbolVariant(.slash)` adds/removes a slash overlay cleanly, ternary symbol names swap icons without animation, and `.contentTransition(.symbolEffect(.replace))` animates the full symbol swap. The `.magic(fallback: .downUp)` variant understands the semantic relationship between symbols, so when switching between `"speaker.wave.2.fill"` and `"speaker.slash.fill"`, it animates just the slash line while keeping the base shape in place. For animation techniques beyond SF Symbols, see the *Animations, Transitions, and Matched Geometry* section of *Chapter 12*.



Apple's free SF Symbols app for macOS is worth keeping open while you work. It is the fastest way to browse names, inspect variants, preview rendering mo des, and sanity-check whether a symbol will read well before you commit to it in code.

See also

- `Picker` documentation: https://x.swiftuicookbook.com/xl3lw

- `Toggle` documentation: https://x.swiftuicookbook.com/k3hor
- `Slider` documentation: https://x.swiftuicookbook.com/8ktlu
- `DatePicker` documentation: https://x.swiftuicookbook.com/nmfpo
- SF Symbols app: https://x.swiftuicookbook.com/0i5i1
- Animate symbols in your app (WWDC 2023): https://x.swiftuicookbook.com/1fnh5
- What's new in SF Symbols 6 (WWDC 2024): https://x.swiftuicookbook .com/bvhv3
- What's new in SF Symbols 7 (WWDC 2025): https://x.swiftuicookbook.com/hmree

UIKit Interoperability Basics

When SwiftUI was first introduced, many UIKit components had no native equivalent. Apple provided two bridging protocols to ensure full compatibility between the frameworks, and while SwiftUI has since covered most common use cases, these protocols remain essential for accessing specialized UIKit components and third-party libraries that still rely on UIKit. The reverse direction is also possible with `UIHostingController`, which lets a UIKit app present a SwiftUI view as if it were a regular view controller. Teams working on established UIKit apps often use this approach to adopt SwiftUI feature by feature.

`UIViewRepresentable` wraps individual UIKit views, and `UIViewControllerRepresentable` wraps view controllers. Both work the

same way: we write a `make` method that creates and configures the UIKit component once, and an `update` method that SwiftUI calls whenever our state changes so we can push new data into the component. When the UIKit side needs to talk back, through delegate callbacks, for instance, we provide a **Coordinator** object to bridge those events into SwiftUI.

In this recipe, we focus on bringing UIKit into SwiftUI, since that is the more common direction when building a new SwiftUI app. We will build a sample app called **TextPreviewer** . The app will display a `UILabel` wrapped in a `UIViewRepresentable` struct that updates live as we type and adjust a font size slider. We will also wrap `UIFontPickerViewController` using `UIViewControllerRepresentable` with a Coordinator that implements the font picker's delegate protocol, allowing the user to choose any installed font. By the end, we wi ll have a working example of both interoperability protocols a nd the Coordinator pattern.

Getting ready

Create a new SwiftUI iOS app named **TextPreviewer** .

How to do it...

Our app will have a single view with a text field, a UIKit-powered label for previewing text, a slider for font size, and a toolbar button that opens a UIKit font picker. We will build the SwiftUI layout first, then create the UIKit wrappers one at a time. These are the steps:

1. Open `ContentView.swift` and replace the `ContentView` struct with the following, leaving the existing `#Preview` macro in place. We

create a `NavigationStack` with a `VStack` containing a `TextField` for input, a temporary `Text` placeholder where our UIKit label will go, a `Slider` for font size, and a caption:

```
struct ContentView: View {
  @State private var text = "Hello, SwiftUI + UIKit!"
  @State private var fontSize: CGFloat = 24
  var body: some View {
    NavigationStack {
      VStack(spacing: 20) {
        TextField("Enter text", text: $text)
          .textFieldStyle(.roundedBorder)
        Text(text)
          .font(.system(size: fontSize))
        Spacer()
        Slider(value: $fontSize, in: 12...72)
        Text("\(Int(fontSize)) pt")
          .font(.caption)
          .foregroundStyle(.secondary)
      }
      .padding()
      .navigationTitle("Text Previewer")
    }
  }
}
```

2. Now, let's create our first UIKit wrapper. We start with a `UILabel` embedded in SwiftUI, which shows the representable lifecycle without adding delegate complexity. Add `import UIKit` at the top of the file, then add the following `AttributedLabel` struct above `ContentView`:

```
import UIKit
struct AttributedLabel: UIViewRepresentable {
  var text: String
  var font: UIFont
  func makeUIView(context: Context) -> UILabel {
```

```
    let label = UILabel()
    label.numberOfLines = 0
    label.textAlignment = .center
    label.setContentCompressionResistancePriority(
      .defaultLow, for: .horizontal)
    label.setContentCompressionResistancePriority(
      .defaultLow, for: .vertical)
    return label
  }
  func updateUIView(_ uiView: UILabel, context: Context) {
    uiView.text = text
    uiView.font = font
  }
}
```

3. Back in `ContentView`, replace the temporary `Text(text)` placeholder and its `.font(.system(size: fontSize))` modifier with our new `AttributedLabel`:

```
AttributedLabel(
  text: text,
  font: .systemFont(ofSize: fontSize)
)
```

   Run the preview or check the step files in the book's GitHub repository to verify. The UIKit label updates in real time as we type, and moving the slider changes the font size instantly. Both interactions flow through `updateUIView`, which keeps the UIKit label fully driven by SwiftUI state.

4. Now, let's wrap `UIFontPickerViewController`. This wrapper introduces a delegate callback and therefore requires a Coordinator. Add the following `FontPickerView` struct below `AttributedLabel`:

```
struct FontPickerView: UIViewControllerRepresentable {
  @Binding var selectedFont: UIFont
  @Environment(\.dismiss) var dismiss
  func makeUIViewController(
    context: Context
  ) -> UIFontPickerViewController {
    let config = UIFontPickerViewController.Configuration()
    config.includeFaces = true
    let picker = UIFontPickerViewController(configuration:
    picker.delegate = context.coordinator
    return picker
  }
  func updateUIViewController(
    _ uiViewController: UIFontPickerViewController,
    context: Context
  ) {}
  func makeCoordinator() -> Coordinator {
    Coordinator(self)
  }
  class Coordinator: NSObject, UIFontPickerViewControllerDe
    let parent: FontPickerView
    init(_ parent: FontPickerView) {
      self.parent = parent
    }
    func fontPickerViewControllerDidPickFont(
      _ viewController: UIFontPickerViewController
    ) {
      if let descriptor = viewController.selectedFontDescri
        let size = parent.selectedFont.pointSize
        parent.selectedFont = UIFont(descriptor: descriptor
      }
      parent.dismiss()
    }
    func fontPickerViewControllerDidCancel(
      _ viewController: UIFontPickerViewController
    ) {
      parent.dismiss()
    }
  }
}
```

5. Now, connect the picker to the main view with state and sheet presentation. This is where the sample starts to feel realistic; SwiftUI owns the presentation state, while UIKit supplies the specialized UI. Add two new state properties to `ContentView`:

```
@State private var selectedFont =
  UIFont.systemFont(ofSize: 24)
@State private var showFontPicker = false
```

   Then, add a `.toolbar` with a button and a `.sheet` modifier after the `.navigationTitle("Text Previewer")` line. These modifiers attach to the `VStack`, not the `NavigationStack`, so we can show the font picker when the button is tapped:

```
.toolbar {
  Button("Choose Font") {
    showFontPicker = true
  }
}
.sheet(isPresented: $showFontPicker) {
  FontPickerView(selectedFont: $selectedFont)
}
```

6. To finish our app, update the `AttributedLabel` to use the selected font and update the caption to show the font name:

```
AttributedLabel(
  text: text,
  font: selectedFont.withSize(fontSize)
)
```

```
Text("\(Int(fontSize)) pt \u{2014} \(selectedFont.familyName)")
  .font(.caption)
  .foregroundStyle(.secondary)
```

Make sure the preview canvas is in live mode, then tap **Choose Font** in the toolbar. Select a font from the picker, then adjust the slider. The preview label reflects both the chosen font family and the current size. The following screenshot shows the completed app.



Figure 1.5: The completed TextPreviewer app

Experiment with the app: type different text, resize it, and switch across font families. The sample is small, but it demonstrates the c ore interoperability loop you will use whenever SwiftUI does not expose the UIKit feature you need.

How it works...

In our `AttributedLabel`, `makeUIView` runs once to configure the label's line count, alignment, and compression resistance. After that, `updateUIView` fires on every keystroke and slider movement, pushing the current text and font into the `UILabel`. The rule of thumb is: expensive setup goes in `make`, lightweight data updates go in `update`. From the outside, `AttributedLabel` behaves like any other SwiftUI view: it participates in layout, responds to state changes, and accepts view modifiers.

In our font picker, SwiftUI calls `makeCoordinator()` before `makeUIViewController(context:)`, so the coordinator is ready to be assigned as the delegate immediately. The coordinator holds a reference to its parent struct, giving it access to `@Binding` properties and `@Environment` values. It reads the selected font descriptor, writes it to `selectedFont` via the binding (which propagates the change back to `ContentView`'s `@State`, triggering a re-render that calls `updateUIView` on the label), and calls `dismiss()` to close the sheet.

The `updateUIViewController` method is empty because we do not need to push SwiftUI state into the font picker, because it is a self-contained modal. Not every representable will have an empty update method; for instance, wrapping a `MKMapView` would require updating the map region whenever the user's location changes. For more on

`@Binding` and data flow, see the *State and Binding Basics* section of *Chapter 5*.



Before wrapping a UIKit component, check whether a native SwiftUI equivalent meets your needs: `ProgressView` replaces `UIActivi tyIndicatorView`, `Map` replaces `MKMapView`, `PhotosPicker` replaces `UIImagePickerController`, among others.

See also

- `UIViewRepresentable` documentation: https://x.swiftuicookbook.com/ba2oz
- `UIViewControllerRepresentable` documentation: https://x.swiftuicookbook.com/s3i25
- `UIHostingController` documentation: https://x.swiftuicookbook.com/vukt5
- Interfacing with UIKit (Apple tutorial): https://x.swiftuicookbook.com/atk91
- Integrating SwiftUI (WWDC 2019): https://x.swiftuicookbook.com/qzwfw
- Use SwiftUI with UIKit (WWDC 2022): https://x.swiftuicookbook.com/rsp12

2

Lists, Scroll Views, and Search

I have yet to work on an iOS project that didn't need a scrollable list somewhere on the first screen. Whether it is a list of messages, a feed of photos, or a catalog of products, presenting collections is one of the most common tasks in app development. In UIKit, getting a scrollable list meant adopting data source and delegate protocols, managing cell reuse, and wiring up editing gestures by hand. In SwiftUI, most of that plumbing disappears: we give `List` an array of items and a view for each row, and the framework takes care of scrolling, cell reuse, and accessibility without any extra setup.

The previous chapter focused on individual views, navigation, and input controls. Now, the problem changes: once a screen needs to display a collection of items, we need to decide whether the framework should manage scrolling, editing, and row layout for us, or whether we want to control those ourselves. `List` handles the first case, while `ScrollView` gives us full control for free-form, visual content.

This chapter builds in a deliberate sequence. We start with an app that displays travel destinations organized by continent, using `List`, `Section`, and `DisclosureGroup` to build grouped, expandable rows, with `.onDelete()` and `.onMove()` for editing gestures. We then layer

`.searchable()` with token-based filtering, search scopes, and suggestions on top of the same app, so the user can find content as the list grows. From there, we add drag-and-drop using the `Transferable` protocol, letting the user drag destinations into a trip planner that reacts the moment a drag begins, thanks to `.onDragSessionUpdated()`. Next, we switch to a photo gallery built with `ScrollView`, where we explore snapping with `.scrollTargetBehavior()`, position tracking with `ScrollPosition`, and programmatic scrolling, capabilities that `List` simply does not offer. The same recipe brings swipe actions, once a `List` exclusive, to the scroll view's rows with `.swipeActionsContainer()`. Finally, we build a trip-planning board whose tiles the user drags into visiting order, marking the grid's items with `.reorderable()`, scoping the gesture with `.reorderContainer(for:)`, and applying each `ReorderDifference` back to the data source.

By the end, you should feel comfortable choosing between `List` and `ScrollView` for any collection-based screen, and confident layering search, filtering, and drag-and-drop interactions on top.

In this chapter, we will cover the following recipes:

- Lists and Sections
- Search and Filtering
- Drag and Drop
- Scroll Views
- Reorderable Containers

Technical requirements

The code in this chapter is based on Xcode 27.0 and iOS 27.0. You can download and install the latest version of Xcode from the App Store. All the code examples for this chapter can be found on GitHub at https://x.swiftuicookbook.com/ch02 .

Lists and Sections

Whenever an app needs to present a collection of items, `List` is usually the first tool we reach for. It handles cell reuse, scrolling, and accessibility out of the box, and it supports built-in gestures for deleting and reordering rows. I think of `List` the way I used to think of `UITableView` , except that the boilerplate around data sources, delegates, and cell registration is gone.

In this recipe, we will build a travel destinations app that groups destinations by continent using `Section` , lets the user expand each row with `DisclosureGroup` to see country and tag details, and supports adding, deleting, and reordering through `.onDelete()` , `.onMove()` , and an `EditButton` in the toolbar. The app is designed to carry forward into the next two recipes, where we layer search and drag-and-drop on top of the same codebase, so the foundation we build here needs to be solid enough to extend without rewriting .

Getting ready

Create a new SwiftUI iOS app named **TravelDestinations** .

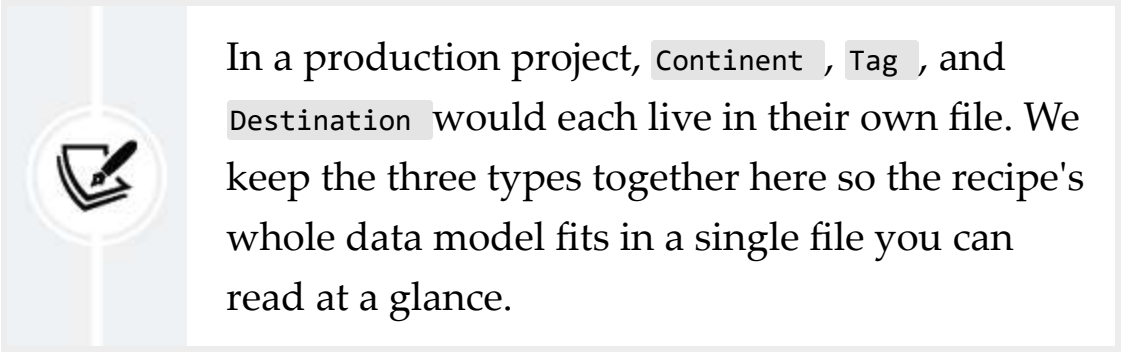
How to do it...

We will start by defining our data model in a separate file, then build the list in `ContentView` one layer at a time: flat rows first, then grouped sections, expandable details, and finally editing and styling. These are the steps:

1. Create a new Swift file named `Destination.swift`. Add a `Continent` enum, a `Tag` enum, and a `Destination` struct that holds a name, country, continent, and a list of tags. The `samples` array starts with two destinations per continent, and `extras` holds additional destinations the user can add later:

```
import SwiftUI
enum Continent: String, CaseIterable, Identifiable {
  case northAmerica = "North America"
  case europe = "Europe"
  case asia = "Asia"
  case southAmerica = "South America"
  case oceania = "Oceania"
  var id: String { rawValue }
}
enum Tag: String, CaseIterable, Identifiable {
  case beach = "Beach"
  case adventure = "Adventure"
  case cultural = "Cultural"
  var id: String { rawValue }
  var icon: String {
    switch self {
      case .beach: "sun.max.fill"
      case .adventure: "figure.hiking"
      case .cultural: "building.columns.fill"
    }
  }
}
struct Destination: Identifiable, Hashable {
  let id = UUID()
  var name: String
  var country: String
  var continent: Continent
```

```
  var tags: [Tag]
}
extension Destination {
  static let samples: [Destination] = [
    .init(name: "New York City", country: "USA",
      continent: .northAmerica, tags: [.cultural]),
    .init(name: "Cancun", country: "Mexico",
      continent: .northAmerica, tags: [.beach]),
    .init(name: "Barcelona", country: "Spain",
      continent: .europe, tags: [.cultural]),
    .init(name: "Santorini", country: "Greece",
      continent: .europe, tags: [.beach]),
    .init(name: "Kyoto", country: "Japan",
      continent: .asia, tags: [.cultural]),
    .init(name: "Jaipur", country: "India",
      continent: .asia, tags: [.cultural]),
    .init(name: "Rio de Janeiro", country: "Brazil",
      continent: .southAmerica, tags: [.beach, .cultural])
    .init(name: "Machu Picchu", country: "Peru",
      continent: .southAmerica, tags: [.adventure, .cultur
    .init(name: "Sydney", country: "Australia",
      continent: .oceania, tags: [.beach, .cultural]),
    .init(name: "Auckland", country: "New Zealand",
      continent: .oceania, tags: [.adventure]),
  ]
  static let extras: [Destination] = [
    .init(name: "Grand Canyon", country: "USA",
      continent: .northAmerica, tags: [.adventure]),
    .init(name: "Swiss Alps", country: "Switzerland",
      continent: .europe, tags: [.adventure]),
    .init(name: "Bali", country: "Indonesia",
      continent: .asia, tags: [.beach, .adventure]),
    .init(name: "Buenos Aires", country: "Argentina",
      continent: .southAmerica, tags: [.cultural]),
    .init(name: "Queenstown", country: "New Zealand",
      continent: .oceania, tags: [.adventure]),
  ]
}
```



In a production project, `Continent`, `Tag`, and `Destination` would each live in their own file. We keep the three types together here so the recipe's whole data model fits in a single file you can read at a glance.

2. Open `ContentView.swift`. Add a `@State` property initialized with the sample data, wrap the view in a `NavigationStack`, and display the destinations in a `List` using `ForEach`:

```
struct ContentView: View {
  @State private var destinations = Destination.samples
  var body: some View {
    NavigationStack {
      List {
        ForEach(destinations) { dest in
          Text(dest.name)
        }
      }
      .navigationTitle("Travel Destinations")
    }
  }
}
```

3. Group the destinations by continent using `Section`. We add a computed property that groups destinations by their continent, and then iterate over the continents to create a section for each one:

```
struct ContentView: View {
  @State private var destinations = Destination.samples
  var grouped: [(Continent, [Destination])] {
    let dict = Dictionary(grouping: destinations, by: \.co
```

```
    return Continent.allCases.compactMap {
      guard let items = dict[$0] else { return nil }
      return ($0, items)
    }
  }
  var body: some View {
    NavigationStack {
      List {
        ForEach(grouped, id: \.0) { continent, items in
          Section(continent.rawValue) {
            ForEach(items) { dest in
              Text(dest.name)
            }
          }
        }
      }
      .navigationTitle("Travel Destinations")
    }
  }
}
```

The `compactMap` over `Continent.allCases` ensures sections always appear in the enum's declaration order rather than the random order of a dictionary.

4. Replace each plain `Text` row with a `DisclosureGroup` so the user can tap to expand a destination and see its country and tags. Keeping the details hidden behind a disclosure indicator lets the list stay compact while still making the information accessible with a single tap. Rather than growing the `body`, we build the row in a small private method, `destinationRow(for:)`, and call it from the `ForEach`:

```
ForEach(items) { dest in
  destinationRow(for: dest)
}
```

Add the method below the `body`. Returning `some View` lets Swift infer the concrete view type for us. This is our first reusable subview; we will explore the technique in depth in the *Reusable Views and Modifiers* section of *Chapter 4*:

```
private func destinationRow(for dest: Destination) -> some View
  DisclosureGroup(dest.name) {
    LabeledContent("Country") {
      Text(dest.country)
    }
    LabeledContent("Tags") {
      HStack {
        ForEach(dest.tags) { tag in
          Image(systemName: tag.icon)
        }
      }
    }
  }
}
```

5. Add swipe-to-delete (to remove rows by swiping left) and drag-to-reorder (to rearrange rows with drag handles in edit mode) by attaching `.onDelete()` and `.onMove()` to the inner `ForEach`, not to the `List` itself. This is a common point of confusion: the modifiers need to know which specific `ForEach` collection to act on. Each closure forwards to a private method, keeping the view code focused on layout:

```
ForEach(items) { dest in
  destinationRow(for: dest)
}
.onDelete { offsets in
  deleteItems(at: offsets, in: items)
}
```

```
.onMove { from, to in
  moveItems(from: from, to: to, in: items)
}
```

Add the two methods below the `body`. They translate the section-local indices SwiftUI hands us into positions in the flat `destinations` array:

```
private func deleteItems(at offsets: IndexSet, in items: [Desti
  let ids = offsets.map { items[$0].id }
  destinations.removeAll { ids.contains($0.id) }
}
private func moveItems(from: IndexSet, to: Int, in items: [Dest
  let ids = items.map(\.id)
  var flat = destinations
  let indices = ids.compactMap { id in
    flat.firstIndex { $0.id == id }
  }
  let mapped = from.map { indices[$0] }
  let target = to < indices.count ? indices[to]
    : indices[indices.count - 1] + 1
  flat.move(fromOffsets: IndexSet(mapped), toOffset: target)
  destinations = flat
}
```



When a row is expanded, deleting it removes the entire destination, not just the country or tag line you happen to be looking at, because `.onDelete()` operates on the elements of the `ForEach`, the destination rows themselves.

Add the `EditButton` in the toolbar's leading position. Unlike `.onDelete()` and `.onMove()`, which went on the `ForEach`, the

`.toolbar()` modifier attaches to the `List` itself:

```
.toolbar {
  ToolbarItem(placement: .topBarLeading) {
    EditButton()
  }
}
```

6. Now, we finally need the `extras` pool we defined in step 1. Add a `@State` property for it right below `destinations`:

   ```
   @State private var extras = Destination.extras
   ```

   Then, add a toolbar button that lets the user add new destinations from the pool. Each tap moves the first available extra into the main list, and the button disables when the pool is empty:

   ```
   ToolbarItem(placement: .topBarTrailing) {
     Button {
       if let next = extras.first {
         destinations.append(next)
         extras.removeFirst()
       }
     } label: {
       Image(systemName: "plus")
     }
     .disabled(extras.isEmpty)
   }
   ```

7. Handle the empty state. An empty `List` renders as a blank screen with no guidance, so if the user deletes every destination, we show a `ContentUnavailableView` that tells them how to start

adding again. Since we are restructuring the `body` anyway, this is also the moment to move the `List` into a computed property, `destinationList`, the same trick we used for the rows in step 4:

```
Group {
  if destinations.isEmpty {
    ContentUnavailableView("No Destinations",
      systemImage: "airplane.departure",
      description: Text("Tap + to add destinations"))
  } else {
    destinationList
  }
}
```

Add the property below the `body`, holding the `List` exactly as we built it in steps 3 through 5:

```
private var destinationList: some View {
  List {
    ForEach(grouped, id: \.0) { continent, items in
      Section(continent.rawValue) {
        ForEach(items) { dest in
          destinationRow(for: dest)
        }
        .onDelete { offsets in
          deleteItems(at: offsets, in: items)
        }
        .onMove { from, to in
          moveItems(from: from, to: to, in: items)
        }
      }
    }
  }
}
```



With the rows in `destinationRow(for:)` and the list in `destinationList`, the `body` reads as an outline: a navigation stack, an empty-state switch, a title, and a toolbar. Extracting named subviews like this is the standard cure for deeply nested SwiftUI code. As a challenge, once you finish this recipe, try moving the tag icons `HStack` from step 4 into a subview of your own.

8. Add `.listRowBackground()` and `.listRowSeparator(.hidden)` to the `ForEach` inside `destinationList`, after the `.onDelete()` and `.onMove()` modifiers. This gives every row a subtle tinted background and removes the default separators for a cleaner look:

```
ForEach(items) { dest in
  destinationRow(for: dest)
}
.onDelete { offsets in
  // ... delete logic
}
.onMove { from, to in
  // ... move logic
}
.listRowBackground(Color.blue.opacity(0.05))
.listRowSeparator(.hidden)
```

9. Run the app or use the preview canvas to test the finished list. Tap **+** to add destinations from the pool, expand a row to see its details, swipe left to delete, and tap **Edit** to reorder with drag handles. Delete all destinations to see the empty state. The complete `ContentView` should look like the following:

```
struct ContentView: View {
  @State private var destinations = Destination.samples
  @State private var extras = Destination.extras
  var grouped: [(Continent, [Destination])] {
    // ... grouping logic (step 3)
  }
  var body: some View {
    NavigationStack {
      Group {
        if destinations.isEmpty {
          // ... ContentUnavailableView (step 7)
        } else {
          destinationList
        }
      }
      .navigationTitle("Travel Destinations")
      .toolbar {
        // ... EditButton + add button (steps 5-6)
      }
    }
  }
  // ... destinationList (steps 7-8) and destinationRow (s
  // ... deleteItems and moveItems methods (step 5)
}
```

For the complete code, see the book's GitHub repository.



Drag-to-reorder may not work in Xcode's live preview due to a known limitation with complex list interactions. Run the app in Device Hub to verify reordering.

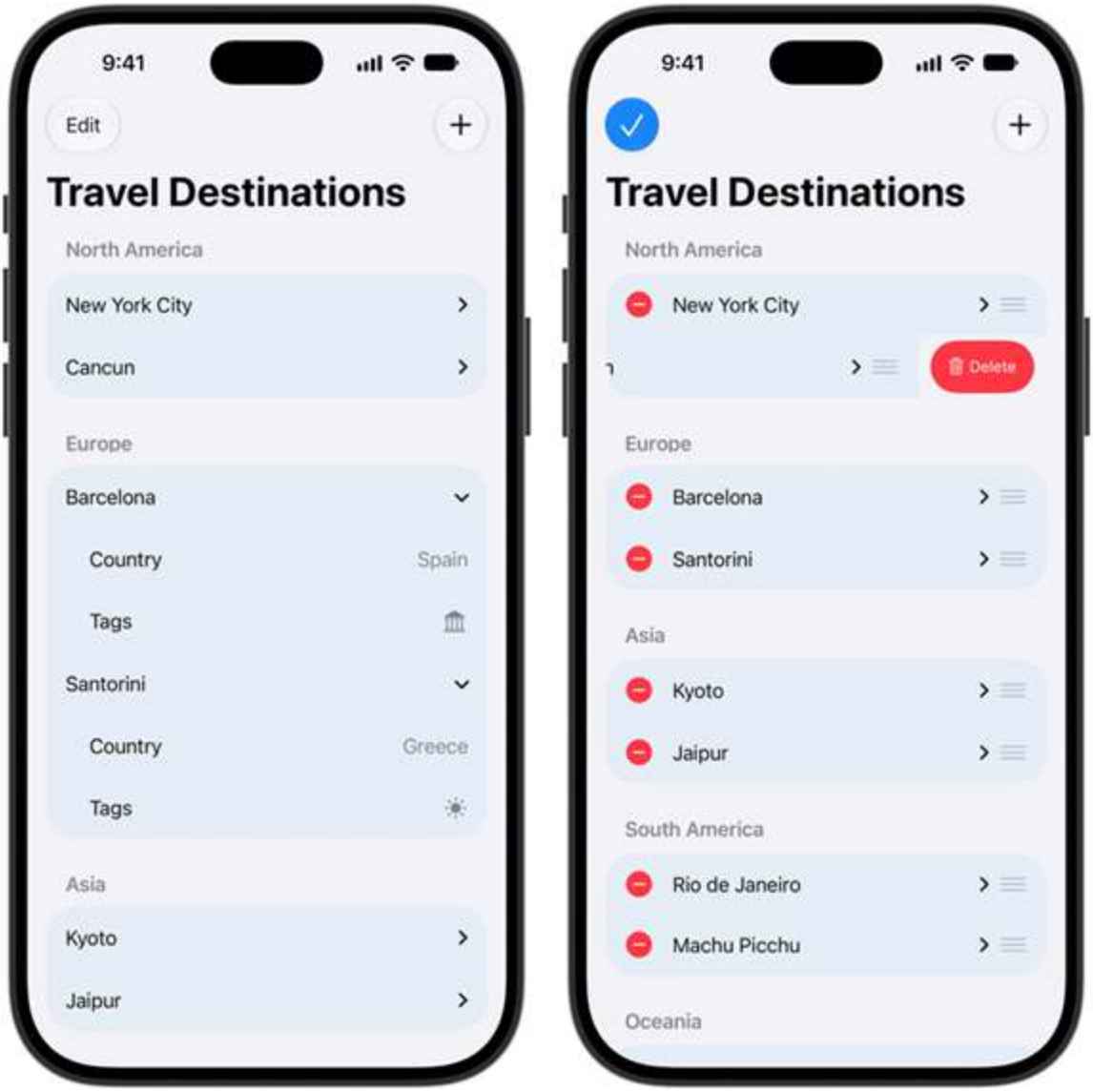


Figure 2.1: Left: the list with continent sections and two expanded disclosure groups showing country and tags. Right: edit mode with delete icons, drag handles, and the two-step delete button revealed on one row

How it works...

I reach for `List` when the screen is fundamentally row-driven, and I want the platform to handle the mechanics rather than coding them from scratch. The `List` in our travel app is a good example: scrolling, cell recycling, editing gestures, and platform-appropriate styling all

come built in, without extra plumbing. Inside the `List`, `ForEach` generates rows from a collection and requires each element to be `Identifiable` (or for us to provide a key path via the `id:` parameter) so SwiftUI can track which rows to insert, remove, or move when the data changes. We will see `ForEach` appear throughout the book whenever we need to iterate over data in a view. We also wrap our content in a `NavigationStack` so the list gets a navigation bar and a title. For free-form scrollable layouts where we need more control over positioning, we will use `ScrollView` in the *Scroll Views* section later in this chapter.

`Section` groups rows under a header, giving the list a clear visual structure. The `DisclosureGroup` view adds a disclosure indicator that toggles between collapsed and expanded states. We used the uncontrolled form here, where SwiftUI manages the expansion state internally. If we needed to control it programmatically, we could pass an `isExpanded` binding instead. We will explore `DisclosureGroup` further in the *Menus and Disclosure Groups* section of *Chapter 4*. Inside the disclosure group, `LabeledContent` pairs a label with a value in a single row; we will cover it in more detail in the *Forms* section of *Chapter 4*.

Two private members keep the `body` readable as the recipe grows. `destinationRow(for:)` builds one row, and returning `some View` (an opaque type) spares us from spelling out the concrete view type SwiftUI composes. `destinationList` holds the whole `List`, so the `body` stays an outline: a navigation stack, the empty-state switch, a title, and a toolbar. Extracting computed properties and small methods like these costs nothing at runtime, because SwiftUI sees exactly the same view tree, and it is the first tool to reach for when indentation

gets deep. We will build a full vocabulary of reuse techniques in the *Reusable Views and Modifiers* section of `Chapter 4`.

The `.onDelete()` and `.onMove()` modifiers attach to `ForEach`, not to `List`, and each one adds a complete interaction to the rows. `.onDelete()` gives the user two ways to remove a row: swiping left reveals a **Delete** button directly, and when the user taps the **Edit** button in the toolbar, the list enters edit mode, where a red circle icon appears on the leading edge of each row. Tapping that icon discloses the same **Delete** button as a two-step confirmation, which is useful for preventing accidental deletions. Either way, the delete removes the whole destination row, even when the row is expanded to show its details. `.onMove()` also activates in edit mode: drag handles appear on the trailing edge of each row, letting the user rearrange items within their section by dragging.

When SwiftUI calls the closures for these two modifiers, it passes indices relative to the section, not to our flat `destinations` array. Since each section displays only a filtered slice, we need to map those indices back to the source array. That mapping lives in the `deleteItems(at:in:)` and `moveItems(from:to:in:)` methods, so each closure stays a single line and the `body` keeps reading as layout. In `deleteItems(at:in:)`, we convert the offsets to item IDs and remove the matching entries. `moveItems(from:to:in:)` follows a similar pattern, translating section-local offsets into positions in the flat array. One detail worth noting: when the user drags a row to the very end of a section, the `to` parameter equals the section's item count, which would be out of bounds. We handle this by computing `target` as the index just past the last item when `to` reaches that limit.

The toolbar also carries an add button on the trailing side that pulls the next destination from the `extras` pool. In a real-world app, this button would present a form for the user to enter a new destination (we will build forms in the *Forms* section of *Chapter 4*), but since our focus here is on lists, we keep it simple by drawing from a predefined pool. We use `ToolbarItem` with `placement:` to control where each button appears; we will explore toolbar configurations in depth in the *Toolbars and the Liquid Glass Design* section of *Chapter 4*. Because both arrays are `@State` properties, SwiftUI re-renders the list automatically when we append to `destinations` or remove from `extras`. Once the pool is empty, the `.disabled(extras.isEmpty)` modifier grays out the button so the user gets clear feedback that there are no more destinations to add.

If the user deletes every destination, `ContentUnavailableView` replaces the list with a centered message and icon, guiding them to tap + to start adding again. We will use `ContentUnavailableView` again in the *Async/await with Remote APIs* section of *Chapter 8* to handle empty network responses. We wrap the `if` / `else` in a `Group` because the two branches return different view types (`ContentUnavailableView` vs `List`), and SwiftUI needs a single type to apply the `.navigationTitle()` and `.toolbar( )` modifiers to.

See also

- `List` documentation: https://x.swiftuicookbook.com/pcda1
- `Section` documentation: https://x.swiftuicookbook.com/q61l6

- `DisclosureGroup` documentation: https://x.swiftuicookbook.com/f0e7a
- `ForEach` documentation: https://x.swiftuicookbook.com/a1ike

Search and Filtering

Once a list has more than a screenful of items, scrolling alone stops being a practical way to find something. I have seen teams build custom filter views from scratch when `.searchable()` already handles most of what they need, including scoped filtering, token chips, and inline suggestions.

In this recipe, we keep the same travel app and layer search on top of it without changing the underlying list structure. We will wire up `.searchable()` with token-based filtering so the user can combine free text with structured tag chips, add `.searchScopes()` for continent-level filtering, and provide `.searchSuggestions()` for guided discovery when the search bar is empty. We also collapse the search bar with `.searchToolbarBehavior(.minimize)` and place its compact button in the bottom toolbar, an iOS 26 pattern that keeps the view focused on content while search stays within the thumb's reach.

Getting ready

Continue with the **TravelDestinations** project from the previous recipe. If you prefer to start fresh, download the completed project from the book's Git Hub repository.

How to do it...

We will add search capabilities to our existing list without changing its structure. The list, sections, disclosure groups, and editing features stay in place; we just wire up filtering on top. These are the steps:

1. Open `Destination.swift` and add a `shortLabel` computed property to `Continent`. The full continent names are too long for a segmented control, so we use two-letter abbreviations:

```
enum Continent: String, CaseIterable, Identifiable {
  // ... existing cases
  var id: String { rawValue }
  var shortLabel: String {
    switch self {
    case .northAmerica: "NA"
    case .europe: "EU"
    case .asia: "AS"
    case .southAmerica: "SA"
    case .oceania: "OC"
    }
  }
}
```

2. Open `ContentView.swift` and add four new `@State` properties for the search state. We need text for free-text input, an array for selected tokens, an optional continent for the scope, and a boolean to track whether search is active:

```
@State private var searchText = ""
@State private var selectedTokens: [Tag] = []
@State private var selectedContinent: Continent?
@State private var isSearching = false
```

3. Add a `filtered` computed property that applies the scope, tokens, and text filters in sequence. Then, update the existing `grouped` property so it groups `filtered` instead of `destinations`:

```
var filtered: [Destination] {
  var result = destinations
  if let continent = selectedContinent {
    result = result.filter { $0.continent == continent }
  }
  for token in selectedTokens {
    result = result.filter { $0.tags.contains(token) }
  }
  if !searchText.isEmpty {
    result = result.filter {
      $0.name.localizedCaseInsensitiveContains(searchText)
  }
  return result
}
var grouped: [(Continent, [Destination])] {
  let dict = Dictionary(grouping: filtered, by: \.continer
  return Continent.allCases.compactMap {
    guard let items = dict[$0] else { return nil }
    return ($0, items)
  }
}
```

4. Apply `.searchable(text:tokens:isPresented:prompt:)` to the `NavigationStack`, after `.navigationTitle()`. We use the `tokens` parameter so the user can build structured filters alongside free text; each token renders as a chip in the search bar. The trailing closure defines the appearance of each chip as a `Label` with the tag's name and icon:

```
.searchable(text: $searchText, tokens: $selectedTokens,
            isPresented: $isSearching,
```

```
            prompt: "Search destinations") { token in
  Label(token.rawValue, systemImage: token.icon)
}
```

Try it in the preview canvas: tap the search icon, then type a destination name and watch the list filter as you type. The token chips come into play in step 7, where suggestions let the user insert tags as structured filters.

5. Add `.onChange(of: isSearching)` right after `.searchable()` to reset the scope and tokens when the user dismisses the search bar. Without this, the list would stay filtered even after search is closed:

```
.onChange(of: isSearching) {
  if !isSearching {
    selectedContinent = nil
    selectedTokens = []
  }
}
```

6. Add `.searchScopes()` to show a segmented control with continent abbreviations when search is active. We use `shortLabel` (two-letter codes) instead of the full continent names because the full names overflow the segmented control on smaller screens:

```
.searchScopes($selectedContinent) {
  Text("All").tag(Continent?.none)
  ForEach(Continent.allCases) { continent in
    Text(continent.shortLabel).tag(Continent?.some(contine
  }
}
```

7. Add `.searchSuggestions()` to guide the user when the search bar is empty. Without suggestions, the user has to know what to type. We show two kinds of guided options: tag suggestions that insert a token chip when tapped (using `.searchCompletion(tag)` with a `Tag` value), and destination name suggestions that fill the text field (using `.searchCompletion(name)` with a `String`). Tags already selected are filtered out so the user does not see duplicates:

```
.searchSuggestions {
  if searchText.isEmpty {
    let remainingTags = Tag.allCases
      .filter { !selectedTokens.contains($0) }
    if !remainingTags.isEmpty {
      Section("Tags") {
        ForEach(remainingTags) { tag in
          Label(tag.rawValue, systemImage: tag.icon)
            .searchCompletion(tag)
        }
      }
    }
    if selectedTokens.isEmpty {
      Section("Destinations") {
        ForEach(["Barcelona", "Kyoto", "Sydney"],
                id: \.self) { name in
          Text(name).searchCompletion(name)
        }
      }
    }
  }
}
```

8. Apply `.searchToolbarBehavior(.minimize)` to collapse the search bar into a compact icon instead of letting it stretch across the toolbar. Then, add a second `.toolbar` block that positions that

icon in the bottom bar using `DefaultToolbarItem`, an iOS 26 API. The result is a single, thumb-friendly search entry point at the bottom edge, and the navigation bar stays free for the title:

```
.searchToolbarBehavior(.minimize)
.toolbar {
  DefaultToolbarItem(kind: .search, placement: .bottomBar)
  ToolbarSpacer(.flexible, placement: .bottomBar)
}
```

9. Run the app or use the preview canvas. Tap the search icon to activate search, try typing a destination name, switch scopes, and tap a tag suggestion to add a token. Dismiss the search bar and verify the full list returns. The complete `ContentView` should look like the following:

```
struct ContentView: View {
  @State private var destinations = Destination.samples
  @State private var extras = Destination.extras
  @State private var searchText = ""
  @State private var selectedTokens: [Tag] = []
  @State private var selectedContinent: Continent?
  @State private var isSearching = false
  var filtered: [Destination] {
    // ... scope + tokens + text (step 3)
  }
  var grouped: [(Continent, [Destination])] {
    // ... grouping filtered (step 3)
  }
  var body: some View {
    NavigationStack {
      Group {
        // ... ContentUnavailableView / destinationList
      }
      .navigationTitle("Travel Destinations")
      .searchable(// ... step 4)
```

```
        .onChange(of: isSearching) {
          // ... reset filters (step 5)
        }
        .searchScopes(// ... step 6)
        .searchSuggestions {
          // ... tags + destinations (step 7)
        }
        .searchToolbarBehavior(.minimize)
        .toolbar {
          // ... EditButton + add button
        }
        .toolbar {
          // ... search bottom bar (step 8)
        }
      }
    }
  }
```

For the complete code, see the book's GitHub repository.

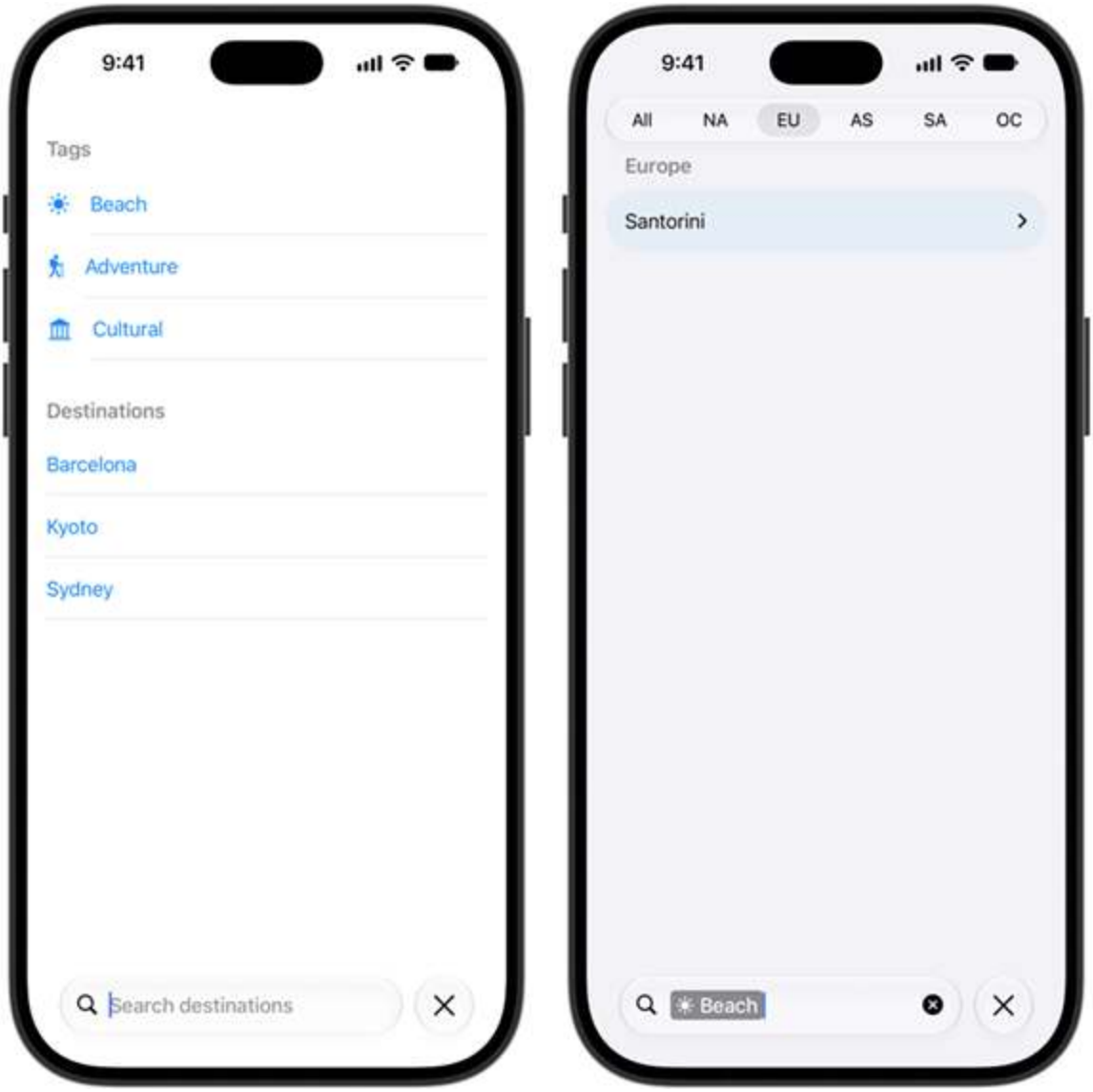


Figure 2.2: Left: search suggestions showing tag and destination options. Right: results filtered by the Beach token with the E urope scope selected

How it works...

The search bar at the top of our list is more than a text field; it is the entry point for the whole filtering model. We use the `searchable(text:tokens:isPresented:prompt:)` variant, which combines free-text input with structured token chips in the same search field.

The `isPresented` binding tracks whether the search bar is active. I like resetting the scope and tokens when the user dismisses search, which is what our `.onChange(of: isSearching)` handler does, because otherwise the search bar disappears but the list stays filtered, with no visible indication of why some destinations are missing. We will cover `.onChange()` in depth in the *State and Binding Basics* section of *Chapter 5*.

The `.searchScopes()` modifier adds a segmented control above the search results when search is active. Each scope maps to a `Continent` case, and we tag the **All** option with `Continent?.none` so SwiftUI can match it to our optional binding.

The suggestions panel is context-sensitive rather than a static list. Tags drop out as the user selects them, and destination names appear only while no tokens are active, so the panel always offers what the user can still add. Both kinds of suggestion funnel through `.searchCompletion()`: passing a `Tag` inserts a token chip, while passing a `String` fills the text field.



SwiftUI's built-in token support works well for single-token filtering. The suggestions view dismisses after a token is selected, and there is no built-in way to keep it open for selecting additional tokens. For scenarios that require more complex filter combinations, consider building a custom search and filter UI.

The `filtered` computed property applies all three filters in sequence: scope first (the coarsest filter), then tokens, then text. Because `grouped`

now groups the `filtered` array instead of `destinations`, the list updates automatically whenever any search state changes. The filtering order does not affect the final result, but applying the coarsest filter first reduces the number of items the subsequent filters process.

On this view, search matters, but it is not the first thing the user needs to see all the time. `.searchToolbarBehavior(.minimize)` keeps that capability close without letting the search field dominate the toolbar. Placing a `DefaultToolbarItem(kind: .search)` in the bottom bar puts the minimized button at the bottom edge, within the thumb's reach. Note that this is search's only entry point: once minimized and placed in the bottom bar, the navigation bar does not show a second search control. `DefaultToolbarItem` and `ToolbarSpacer` are iOS 26 APIs; we will explore them along with the Liquid Glass toolbar design in the *Toolbars and the Liquid Glass Design* section of *Chapter 4*.

See also

- `.searchable()` documentation: https://x.swiftuicookbook.com/p55h6
- `.searchScopes()` documentation: https://x.swiftuicookbook.com/1m49n
- `.searchSuggestions()` documentation: https://x.swiftuicookbook.com/ccr1z

Drag and Drop

Drag and drop feels natural on a touch device, but I have seen projects avoid it because the old UIKit implementation was heavy: drag delegates, drop delegates, session management, and custom preview providers. SwiftUI's approach is simpler. The `Transferable` protocol defines how a type serializes itself for transfer, and a pair of modifiers, `.draggable()` and `.dropDestination()`, handle the rest.

In this recipe, we extend the travel app so the user can drag destinations out of the continent sections and into a trip-planner drop zone above the list. We keep the `Transferable` representation intentionally small (just a UUID string) since the goal here is the drag interaction itself, not data serialization. That keeps the recipe focused on the two things that matter most: marking rows as draggable and building a drop zone that responds when items land on it. We finish with iOS 27's `.onDragSessionUpdated()`, which lets the drop zone light up t he moment a drag begins.

Getting ready

Open the **TravelDestinations** project from the previous recipe. Verify that it builds and that search, scopes, and suggestions all work befo re adding drag-and-drop.

How to do it...

We need two changes: make `Destination` transferable so it can participate in drag operations, then add a drop zone and mark each row as draggable. These are the steps:

1. Open `Destination.swift`. Add `Sendable` to the struct declaration and add a `Transferable` conformance that exports the destination's UUID as a string. This is the lightest representation because we only need to identify which item was dragged, not serialize the entire object:

```
struct Destination: Identifiable, Hashable, Sendable {
  let id = UUID()
  var name: String
  var country: String
  var continent: Continent
  var tags: [Tag]
}
extension Destination: Transferable {
  static var transferRepresentation: some TransferRepresen
    ProxyRepresentation { $0.id.uuidString }
  }
}
```

2. In `ContentView.swift`, add two new state properties for the trip items and the drop-zone highlight:

```
@State private var tripItems: [Destination] = []
@State private var isTripTargeted = false
```

3. In the `body`, replace `destinationList` inside the `else` branch with a `VStack` that places a `tripPlannerDropZone` above the list. We put the drop zone outside the `List` because `.dropDestination()` does not respond reliably when applied to views inside a `List`:

```
VStack(spacing: 12) {
  tripPlannerDropZone
```

```
    destinationList
}
```

Then, inside `destinationList`, add `.draggable(dest.id.uuidString)` to the `destinationRow(for:)` call so the user can long-press any row to start a drag:

```
ForEach(items) { dest in
  destinationRow(for: dest)
    .draggable(dest.id.uuidString)
}
```



Xcode will show an error on `tripPlannerDropZone` because we have not defined it yet. This is expected; the error will go away after step 5.

4. Also update the `ContentUnavailableView` condition to account for trip items:

```
if destinations.isEmpty && tripItems.isEmpty {
```

5. Add the `tripPlannerDropZone` computed property at the bottom of the struct. It uses `.dropDestination(for: String.self)` to receive UUID strings, looks up each destination by ID, and moves it from `destinations` to `tripItems`. The `isTargeted` closure drives the green highlight:

```
private var tripPlannerDropZone: some View {
  VStack(alignment: .leading, spacing: 8) {
    Label("Trip Planner", systemImage: "airplane")
      .font(.headline)
```

```
      if tripItems.isEmpty {
        Text("Drag destinations here")
          .foregroundStyle(.secondary)
          .frame(maxWidth: .infinity, alignment: .center)
      } else {
        ForEach(tripItems) { dest in
          HStack {
            Text(dest.name)
            Spacer()
            Text(dest.country)
              .foregroundStyle(.secondary)
            Button {
              tripItems.removeAll { $0.id == dest.id }
              destinations.append(dest)
            } label: {
              Image(systemName: "xmark.circle.fill")
                .foregroundStyle(.secondary)
            }
            .buttonStyle(.plain)
          }
        }
      }
    }
    .padding()
    .background(
      RoundedRectangle(cornerRadius: 12)
        .fill(isTripTargeted
          ? Color.green.opacity(0.15)
          : Color.blue.opacity(0.05))
    )
    .padding(.horizontal)
    .padding(.top, 8)
    .dropDestination(for: String.self) { ids, _ in
      for id in ids {
        guard let uuid = UUID(uuidString: id),
          let index = destinations.firstIndex(
            where: { $0.id == uuid })
        else { continue }
        tripItems.append(destinations.remove(at: index))
      }
      return true
    } isTargeted: { targeted in
      isTripTargeted = targeted
```

```
    }
  }
```

6. Let the drop zone announce itself as soon as a drag begins, before the user ever hovers over it. Add one more `@State` property next to the two from step 2:

```
@State private var isDragActive = false
```

Attach `.onDragSessionUpdated()` to the `VStack` from step 3. The closure receives a `DragSession` whose `phase` tells us whether a drag is in flight anywhere inside that hierarchy:

```
.onDragSessionUpdated { session in
  withAnimation {
    isDragActive = session.phase == .active
  }
}
```

Then, give `tripPlannerDropZone` a dashed border, right after its `.background()`, so the drop zone lights up while a drag is active:

```
.overlay {
  RoundedRectangle(cornerRadius: 12)
    .strokeBorder(isDragActive ? Color.green : Color.clear,
                  style: StrokeStyle(lineWidth: 2, dash: [6]))
}
```



New in iOS 27. SwiftUI can now observe drag sessions with `.onDragSessionUpdated()`. The session's `phase` moves from `.initial` through



`.active` to `.ended(_:)`, and `draggedItemIDs(for:)` reports the identifiers in flight, so any view in the hierarchy can react to a drag, not just the drop target.

7. Run the app in Device Hub by pressing *Cmd + R* . We use Device Hub because drag interactions are one of the features the preview canvas does not support reliably. Long-press a destination row, then drag it up to the trip-planner area. As soon as the drag starts, the dashed border appears around the trip planner, hinting at where the row can go. The background turns green when the dragged item hovers over the drop zone, and the destination moves into the trip list when you release. Tap the **X** button next to a trip item to return it to the main list.



Figure 2.3: The TravelDestinations app with drag-and-drop: the left screen shows the initial list; the right screen shows two destinations dragged in to the Trip Planner drop zone

How it works...

The drag-and-drop system relies on three pieces working together: a data representation, a drag source, and a drop target.

Our `Destination` struct conforms to the `Transferable` protocol from CoreTransferable, which tells the system how the type can be

serialized for transfer. It uses `ProxyRepresentation` to export its UUID as a plain `String`. This is the simplest representation: we only need to identify which item was dragged, and the UUID string is enough for an in-app lookup. `Destination` also conforms to `Sendable` because transferable data may cross concurrency boundaries during the drag operation.

The `.draggable()` modifier marks each destination row as a drag source. We compose it on the `destinationRow(for:)` call site inside `destinationList`, which keeps the row builder itself free of drag concerns. When the user long-presses a row, SwiftUI creates a drag session and calls the `Transferable` conformance to produce the transfer data. We pass `dest.id.uuidString` directly, matching the proxy representation.

The `.dropDestination(for: String.self)` modifier on the trip-planner view defines the drop target. Its action closure receives an array of UUID strings. For each one, we look up the destination by ID in the `destinations` array, remove it, and append it to `tripItems`. Note that `remove(at:)` does double duty here: it deletes the element from `destinations` and returns it, which is what lets us hand the result straight to `tripItems.append()`. The `isTargeted` closure fires whenever a dragged item enters or exits the drop zone, toggling the green background highlight.



Apple marks `dropDestination(for:action:isTargeted:)` as deprecated in favor of `dropDestination(for:isEnabled:action:)`, whose action closure receives a `DropSession` instead of returning a `Bool`. Xcode 27 does not warn about the older variant,

> and we stay with it deliberately: its `isTargeted:` closure drives the highlight in one line. If you do not need hover feedback, prefer the newer variant.

The `.onDragSessionUpdated()` modifier gives us the drag-side counterpart to that hover feedback. SwiftUI calls the closure every time the session changes, and we map its `phase` to the `isDragActive` flag: `.active` while the user is dragging, and one of the ending phases once the item is released. Because the modifier sits on the `VStack` that contains both the list and the drop zone, it observes drags that start anywhere inside that hierarchy. The result is a two-stage affordance: the dashed border invites the user toward the trip planner the moment they pick up a row, and the green fill confirms the target while they hover over it.

We placed the trip planner outside the `List` because `.dropDestination()` works reliably on regular views but does not respond to drops when applied to a `Section` inside a `List`. Extracting it into a standalone `VStack` with its own background and padding makes it both a clear visual target and a functional drop receiver.

Each trip item has a dismiss button that reverses the operation, removing the destination from `tripItems` and appending it back to `destinations`. This keeps the total set of destinations constant across both collections.

> For production apps that need to transfer complete objects between apps or processes, use



`CodableRepresentation` with a custom `UTType` instead of `ProxyRepresentation`. This encodes the full object as JSON and requires registering the type in Info.plist under `UTExportedTypeDeclarations`. We kept the proxy approach here to focus on the drag-and-drop interactio n rather than data serialization.

See also

- `Transferable` documentation: https://x.swiftuicookbook.com/qxfhe
- `View.draggable()` documentation: https://x.swiftuicookbook.com/t6dnq
- `View.dropDestination(for:action:isTargeted:)` documentation: https://x.swiftuicookbook.com/i39b4
- `View.dropDestination(for:isEnabled:action:)` documentation: https://x.swiftuicookbook.com/1jkme
- `View.onDragSessionUpdated()` documentation: https://x.swiftuicookbook.com/r3ad9
- *Adopting drag and drop using SwiftUI* article: https://x.swiftuicookbook.com/0xlq2

Scroll Views

The previous three recipes used `List` because the screen was fundamentally row-driven: structured data, sections, editing gestures. A photo gallery is a different kind of problem. We want horizontal swiping, full-width cards that snap into place, and a

vertical feed that knows where the user has scrolled. None of that maps well to `List`, which is why `ScrollView` exists.

In this recipe, we use `ScrollView` for two jobs that `List` would fight us on: a horizontal carousel that snaps one card at a time, and a vertical feed where we watch the scroll position closely enough to show a scroll-to-top button after the user has scrolled far enough. This also gives us a good excuse to work through the scroll APIs I end up using most in practice: `ScrollPosition` for tracking and control, `.containerRelativeFrame()` for sizing cards to the available space, `.scrollTargetBehavior()` for snapping, and `.onScrollGeometryChange()` for responding to movement. We close with iOS 27's `.swipeActionsContainer()`, which brings swipe actions, previously a `List` exclusiv e, to the rows of our scroll view.

Getting ready

Create a new S wiftUI iOS app named **PhotoGallery** .

How to do it...

We will scaffold the view structure with placeholders first, then fill in each section. These are the steps:

1. Create a new Swift file named `Photo.swift` in a `Models` folder and add the following data model:

```
import SwiftUI
struct Photo: Identifiable {
  let id: Int
  let title: String
  let color: Color
```

```
}
extension Photo {
  static let samples: [Photo] = (1...30).map { i in
    Photo(id: i, title: "Photo \(i)",
      color: Color(hue: Double(i)/30, saturation: 0.7, bri
  }
}
```

The model stays intentionally simple: an integer ID for stable iteration, a title for labeling each card, and a computed color so we can build a visually distinct gallery without spending time on image assets.

2. In `ContentView.swift`, replace everything except the `#Preview` at the bottom with the view structure. We use `Text` placeholders for the three sections we will build next:

```
import SwiftUI
struct ContentView: View {
  @State private var photos = Photo.samples
  @State private var scrolledID: Int?
  @State private var isScrolling = false
  @State private var showScrollToTop = false
  @State private var position = ScrollPosition(edge: .top)
  var body: some View {
    NavigationStack {
      VStack(spacing: 0) {
        Text("Carousel")
        Text("Photo List")
      }
      .navigationTitle("Photo Gallery")
      .overlay(alignment: .bottomTrailing) {
        if showScrollToTop {
          Text("Scroll to top")
        }
      }
    }
```

```
  }
}
```

The photos live in a `@State` property rather than a plain constant because a later step adds swipe-to-remove, which mutates the array.

3. Add the `carousel` computed property below the `body`. We use `LazyHStack` instead of a regular `HStack` so items are created only as they scroll into view, which matters once the collection grows. `.containerRelativeFrame(.horizontal)` sizes each card to fill the scroll view width without hard-coding a number, and `.scrollTargetBehavior(.viewAligned(limitBehavior: .alwaysByOne))` makes the carousel snap exactly one card per swipe:

```
var body: some View {
  // ... view structure (step 2)
}
private var carousel: some View {
  ScrollView(.horizontal) {
    LazyHStack(spacing: 0) {
      ForEach(photos) { photo in
        ZStack {
          RoundedRectangle(cornerRadius: 16)
            .fill(photo.color.gradient)
          VStack {
            Image(systemName: "photo.fill")
              .font(.system(size: 40))
              .foregroundStyle(.white.opacity(0.8))
            Text(photo.title)
              .font(.title3.bold())
              .foregroundStyle(.white)
          }
        }
        .padding(.horizontal, 20)
        .containerRelativeFrame(.horizontal)
```

```
            }
        }
        .scrollTargetLayout()
    }
    .scrollTargetBehavior(.viewAligned(limitBehavior: .alw
    .scrollPosition(id: $scrolledID)
    .scrollIndicators(.hidden)
    .frame(height: 220)
}
```

In the `body`, replace `Text("Carousel")` with `carousel`. Then, scroll the carousel horizontally in the preview canvas: each swipe should snap exactly one card into place.

4. Add the `photoList` computed property. This is where we explore scroll observation: `.onScrollPhaseChange()` tells us whether the user is actively scrolling, `.onScrollGeometryChange()` watches the content offset so we can show a scroll-to-top button once the user scrolls past a threshold, and `.onScrollVisibilityChange()` fires when individual rows enter or leave the visible area. We also apply `.scrollEdgeEffectStyle(.soft)`, the iOS 26 Liquid Glass edge blur:

```
private var photoList: some View {
  ScrollView {
    LazyVStack(spacing: 12) {
      ForEach(photos) { photo in
        HStack(spacing: 12) {
          RoundedRectangle(cornerRadius: 8)
            .fill(photo.color.gradient)
            .frame(width: 60, height: 60)
            .overlay {
              Image(systemName: "photo")
                .foregroundStyle(.white)
            }
          Text(photo.title)
```

```
                .font(.headline)
              Spacer()
            }
            .padding()
            .background(.background, in: .rect(cornerRadius:
            .shadow(color: .black.opacity(0.05), radius: 4)
            .onScrollVisibilityChange(threshold: 0.5) { visi
              if visible {
                print("\(photo.title) visible")
              }
            }
          }
        }
        .padding()
      }
      .scrollPosition($position)
      .scrollBounceBehavior(.basedOnSize)
      .scrollEdgeEffectStyle(.soft, for: .all)
      .onScrollPhaseChange { _, newPhase in
        withAnimation { isScrolling = newPhase.isScrolling }
      }
      .onScrollGeometryChange(for: Bool.self) { geo in
        geo.contentOffset.y > 200
      } action: { _, shouldShow in
        withAnimation {
          showScrollToTop = shouldShow
        }
      }
    }
```

In the `body`, replace `Text("Photo List")` with `photoList`. In the preview canvas, scroll the photo list vertically and the carousel horizontally: the two scroll views move independently.

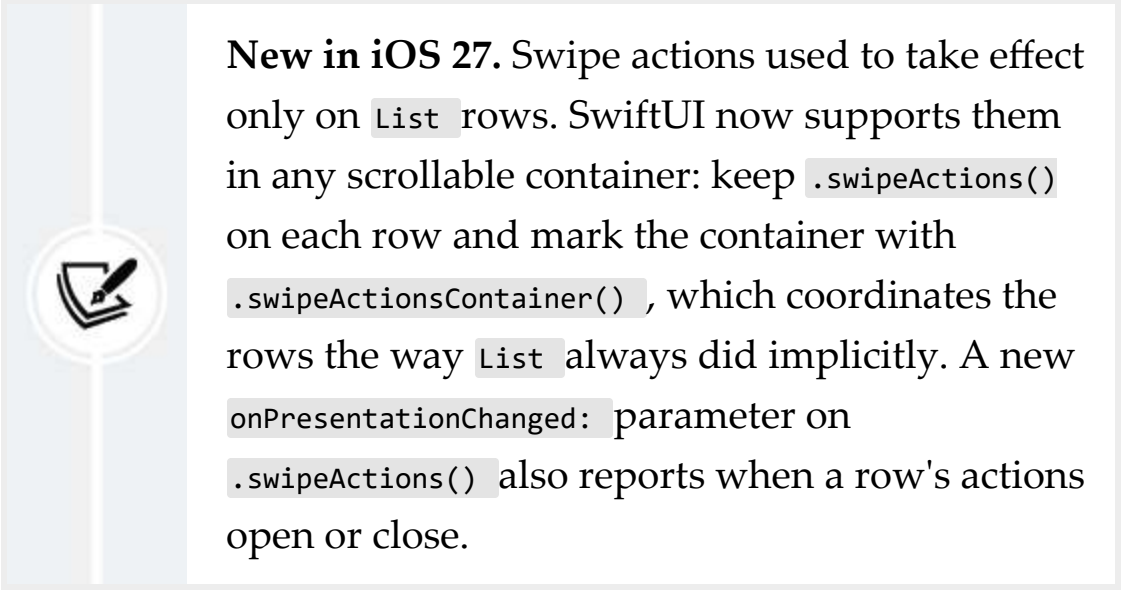
5. Give each photo row a swipe action so the user can remove it, something that used to require a `List`. Attach `.swipeActions()` to the row's `HStack`, right after `.onScrollVisibilityChange()`:

```
.swipeActions {
  Button(role: .destructive) {
    photos.removeAll { $0.id == photo.id }
  } label: {
    Label("Delete", systemImage: "trash")
  }
}
```

Then, mark the scroll view as the container that coordinates those swipes by adding `.swipeActionsContainer()` at the end of `photoList`'s modifier chain, after the `.onScrollGeometryChange()` block:

```
.swipeActionsContainer()
```

In the preview canvas, swipe a photo row to the left and tap **Delete** to remove it.



New in iOS 27. Swipe actions used to take effect only on `List` rows. SwiftUI now supports them in any scrollable container: keep `.swipeActions()` on each row and mark the container with `.swipeActionsContainer()`, which coordinates the rows the way `List` always did implicitly. A new `onPresentationChanged:` parameter on `.swipeActions()` also reports when a row's actions open or close.

6. Add the `scrollToTopButton` computed property. This button uses the `ScrollPosition` binding to scroll programmatically:

```
private var scrollToTopButton: some View {
  Button {
    withAnimation {
      position.scrollTo(edge: .top)
    }
  } label: {
    Image(systemName: "arrow.up.circle.fill")
      .font(.system(size: 44))
      .foregroundStyle(.white, .blue)
      .shadow(radius: 4)
  }
  .padding()
  .transition(.scale.combined(with: .opacity))
}
```

In the `body`, replace `Text("Scroll to top")` with `scrollToTopButton`. In the preview canvas, scroll the photo list down a few screenfuls to make the button appear, then tap it to jump back to the top. Note that the button scrolls only the vertical photo list, since the `position` binding is attached to that scroll view; it has no effect on the carousel's `LazyHStack`.

7. Run the app in Device Hub, or use the preview canvas in live mode. Swipe through the carousel to see photos snap one at a time. Swipe a photo row left to delete it. Scroll the list down past several rows to see the scroll-to-top button appear in the bottom-right corner, then tap it to jump back to the top.

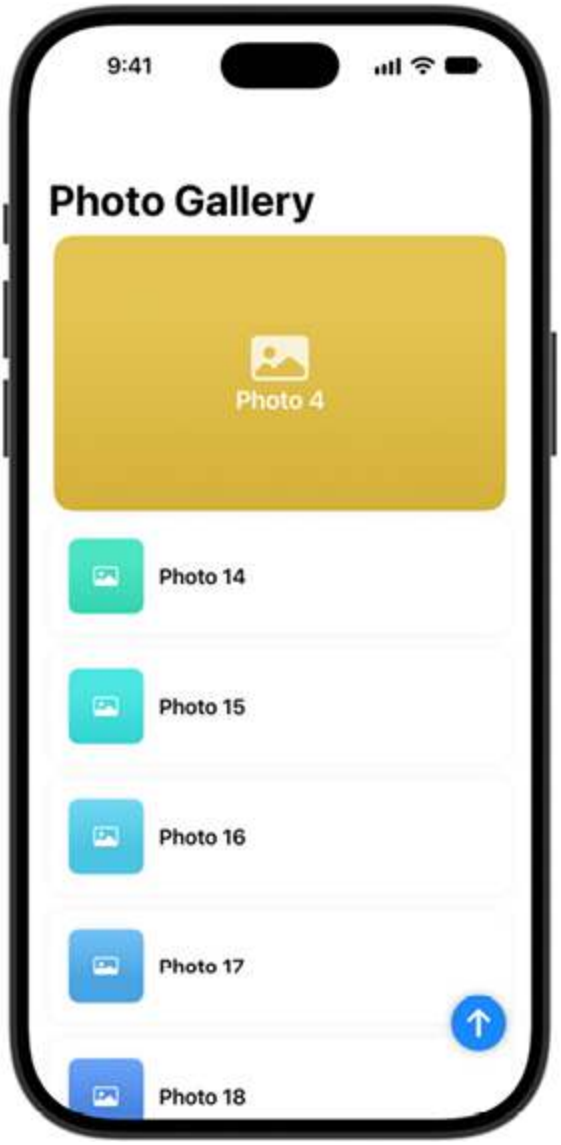


Figure 2.4: The PhotoGallery app showing the horizontal carousel at the top and the vertical photo list below

How it works...

This screen uses two scroll views for two different jobs. The carousel at the top is there to show controlled horizontal movement and snapping; the photo list underneath is where we watch scroll state, visibility, and programmatic movement. Putting both on one screen

makes the trade-off easy to see: same container family, very different interaction goals.

The carousel uses `.scrollTargetLayout()` on the `LazyHStack` to mark its children as snap targets, and `.scrollTargetBehavior(.viewAligned(limitBehavior: .alwaysByOne))` on the `ScrollView` to snap to one card at a time. The `.scrollPosition(id:)` modifier binds `scrolledID` to the currently visible photo, keeping the view in sync with the scroll position. We hide the scroll indicator with `.scrollIndicators(.hidden)` since the snapping behavior already communicates scrollability. Each card uses `.containerRelativeFrame(.horizontal)` to fill the full width of the scroll view without hard-coding a frame size.

The photo list stacks rows inside a `LazyVStack`, which creates views only as they approach the visible area. Each row attaches `.onScrollVisibilityChange(threshold: 0.5)`, an iOS 18 modifier that fires when at least 50% of the row becomes visible inside the scroll view. In a production app this is where we would trigger lazy image loading or analytics tracking.

Three modifiers on the list's `ScrollView` observe scrolling behavior. `.onScrollPhaseChange()` reports gesture phases: `.idle`, `.tracking`, `.interacting`, `.decelerating`, and `.animating`. We use it to track whether the user is actively scrolling. `.onScrollGeometryChange(for:of:action:)` efficiently monitors the scroll offset by transforming the full `ScrollGeometry` into a simple `Bool`. The transform closure runs on every frame, but the action closure fires only when the boolean changes, keeping state updates minimal.

When the content offset exceeds 200 points, the scroll-to-top button appears.

The scroll-to-top button uses `ScrollPosition`, the enhanced iOS 18 type, to scroll programmatically. Calling `position.scrollTo(edge: .top)` inside `withAnimation` produces a smooth animated scroll. The button itself uses `.transition(.scale.combined(with: .opacity))` so it scales and fades in when `showScrollToTop` becomes true. We will cover animations and transitions in detail in the *Animations, Transitions, and Matched Geometry* section of *Chapter 12*.

Two additional modifiers round out the scroll configuration. `.scrollBounceBehavior(.basedOnSize)` prevents bouncing when the content fits entirely within the visible area. `.scrollEdgeEffectStyle(.soft, for: .all)` applies the iOS 26 Liquid Glass edge effect, which adds a subtle blur at the top and bottom edges as content scrolls under translucent bars.

The swipe-to-delete on the photo rows uses the same `.swipeActions()` modifier we would attach to a `List` row. What is new in iOS 27 is that it works inside a plain `ScrollView`: the `.swipeActionsContainer()` modifier on the scroll view coordinates the gesture across the rows it contains, closing one row's actions when another row starts to swipe, the behavior `List` always provided implicitly. Deleting mutates the `photos` array, which is why the model moved into a `@State` property in step 2.

See also

- `ScrollView` documentation: https://x.swiftuicookbook.com/6z6vc

- `LazyVStack` documentation: https://x.swiftuicookbook.com/cph2d
- `scrollTargetBehavior(_:)` documentation: https://x.swiftuicookbook.com/0rt7a
- `onScrollGeometryChange(for:of:action:)` documentation: https://x.swiftuicookbook.com/16icz
- `scrollEdgeEffectStyle(_:for:)` documentation: https://x.swiftuicookbook.com/whzup
- `swipeActionsContainer()` documentat ion: https://x.swiftuicookbook.com/0096h

Reorderable Containers

Rearranging apps on the Home Screen is one of the first gestures every iPhone user learns: press and hold, drag the icon, and everything else shifts out of the way. Offering that same interaction inside our own apps used to be surprisingly hard in SwiftUI. `List` gave us `.onMove()` and drag handles in edit mode, but the moment a design called for a grid or a custom layout, we were left hand-rolling drag gestures and offset math. iOS 27 closes that gap with a pair of modifiers that make any container reorderable.

In this recipe, we build a trip-planning board: a scrolling grid of colored tiles, one for each stop on a ten-city European tour, that the user drags into the order they want to travel. Each tile shows a day number computed from the stop's position in the backing array, so every reorder visibly renumbers the itinerary. We mark the grid's items with `.reorderable()`, scope the gesture with `.reorderContainer(for:)`, and apply the `ReorderDifference` the

framework hands back to our array. We finish by gating the whole interaction behind an **Arrange** mode, using the modifier's `isEnabled` parameter, so casual browsi ng never rearranges the trip by accident.

Getting ready

Crea te a new SwiftUI iOS app named **TripBoard** .

How to do it...

Our app will have just one view, holding the board, plus a data layer in a separate file. We will build it in the same order we would on a real project: get the grid rendering first, then make it reorderable, and only then decide when reordering should be allowed. These are the steps:

1. Create a new Swift file named `Stop.swift` in a `Models` folder. Each stop carries a name, an SF Symbol, and a fixed hue, so a tile keeps its color no matter where the user drags it. The `samples` array lists the ten cities of our tour in the initial visiting order:

```
import SwiftUI
struct Stop: Identifiable {
  let id = UUID()
  var name: String
  var symbol: String
  var hue: Double
  var color: Color {
    Color(hue: hue, saturation: 0.65, brightness: 0.85)
  }
}
extension Stop {
  static let samples: [Stop] = [
    .init(name: "Lisbon", symbol: "sailboat.fill", hue: 0.
```

```
      .init(name: "Madrid", symbol: "sun.max.fill", hue: 0.0
      .init(name: "Barcelona", symbol: "building.columns.fil
      .init(name: "Paris", symbol: "fork.knife", hue: 0.6),
      .init(name: "Amsterdam", symbol: "bicycle", hue: 0.3),
      .init(name: "Berlin", symbol: "music.note", hue: 0.75)
      .init(name: "Prague", symbol: "crown.fill", hue: 0.12)
      .init(name: "Vienna", symbol: "pianokeys", hue: 0.85),
      .init(name: "Venice", symbol: "water.waves", hue: 0.5)
      .init(name: "Rome", symbol: "laurel.leading", hue: 0.4
  ]
}
```



The colored tiles are a deliberate simplification: a fixed hue keeps every stop visually distinct without bringing image assets into the recipe. In a production app, each tile would more likely show a photo of the city it represents.

2. In `ContentView.swift`, replace everything except the `#Preview` at the bottom with the view's structure: the state, the column definition, a `body` that wraps a `board` subview in a `NavigationStack` and a `ScrollView`, and the `board` property itself. For now, each grid cell is a plain `Text` with the city name; we will turn it into a real tile in the next step. Two flexible `GridItem` columns give the board its two-column layout; grids get a full treatment in the *Grid and GridRow* section of *Chapter 10*:

```
import SwiftUI
struct ContentView: View {
  @State private var stops = Stop.samples
  private let columns = [GridItem(.flexible()), GridItem(.
  var body: some View {
    NavigationStack {
```

```
        ScrollView {
          board
        }
        .navigationTitle("Trip Board")
      }
    }
    private var board: some View {
      LazyVGrid(columns: columns, spacing: 12) {
        ForEach(stops) { stop in
          Text(stop.name)
        }
      }
      .padding(.horizontal)
    }
  }
```

Check the preview canvas: the ten city names appear in two plain columns.

3. The tile is about to grow a color gradient, a symbol, and a day badge, and inlining all of that in the `ForEach` would bury the grid's structure. Let's refactor the cell into a small function instead, `stopTile(for:)`, the same extraction technique we used for `destinationList` in the *Lists and Sections* section. It derives the tile's day number from the stop's current position in the `stops` array rather than storing it, and that detail is what will make reordering satisfying to watch: move a tile, and every badge recomputes. The `@ViewBuilder` attribute lets the method's body mix that `let` computation with view content without a `return` keyword; we will look closer at result builders in the *Reusable Views and Modifiers* section of *Chapter 4*. Add the function below the `board` property and update the `ForEach` to call it:

```
  private var board: some View {
    LazyVGrid(columns: columns, spacing: 12) {
      ForEach(stops) { stop in
        stopTile(for: stop)
      }
    }
    .padding(.horizontal)
  }
  @ViewBuilder
  private func stopTile(for stop: Stop) -> some View {
    let day = (stops.firstIndex { $0.id == stop.id } ?? 0)
    ZStack {
      RoundedRectangle(cornerRadius: 16)
        .fill(stop.color.gradient)
      VStack(spacing: 8) {
        Image(systemName: stop.symbol)
          .font(.system(size: 36))
        Text(stop.name)
          .font(.headline)
        Text("Day \(day)")
          .font(.caption.bold())
          .padding(.horizontal, 8)
          .padding(.vertical, 2)
          .background(.white.opacity(0.25), in: .capsule)
      }
      .foregroundStyle(.white)
    }
    .frame(height: 140)
  }
```

Check the preview canvas: the ten stops render as a colorful two-column grid, numbered Day 1 through Day 10 in the sample order.

4. Now for the recipe's centerpiece. Modify the `board` property: mark the tiles as candidates for the gesture by attaching `.reorderable()` to the `ForEach`, and declare the grid as the reorder container with `.reorderContainer(for:)`, whose `move` closure

receives a `ReorderDifference` describing where the dragged tiles landed:

```
private var board: some View {
  LazyVGrid(columns: columns, spacing: 12) {
    ForEach(stops) { stop in
      stopTile(for: stop)
    }
    .reorderable()
  }
  .reorderContainer(for: Stop.self) { difference in
    difference.apply(to: &stops)
  }
  .padding(.horizontal)
}
```

Xcode complains that `apply(to:)` does not exist. The framework describes the move for us, but applying it to our data is our job, and that is what we write next.



New in iOS 27. Drag-to-reorder used to require a `List` and its `.onMove()` modifier. SwiftUI now supports it in any container: mark the items with `.reorderable()` and the container with `.reorderContainer(for:)`, and the framework provides the lift, the placeholder gap, and the drop animation, handing your code a `ReorderDifference` to apply to the data source. The same pair works in grids, stacks, custom layouts, and `List` itself.

5. Back in `Stop.swift`, add an extension that applies a difference to our array. We remove the moved stops in a single pass, capturing them in their original order, then insert them at the destination the difference describes:

```
extension ReorderDifference where ItemID == Stop.ID,
  CollectionID == ReorderableSingleCollectionIdentifier {
  func apply(to stops: inout [Stop]) {
    let moving = Set(sources)
    var moved: [Stop] = []
    stops.removeAll { stop in
      guard moving.contains(stop.id) else { return false }
      moved.append(stop)
      return true
    }
    switch destination.position {
    case .before(let id):
      let index = stops.firstIndex { $0.id == id } ?? stop
      stops.insert(contentsOf: moved, at: index)
    case .end:
      stops.append(contentsOf: moved)
    }
  }
}
```

Run the app in Device Hub. Press and hold any tile, drag it around the board, and watch the other tiles part to make room. Drop it, and every day badge renumbers to match the new itinerary.



The preview canvas does not display the grid's tiles once `.reorderable()` is attached to the `ForEach` and the grid becomes a reorder container; the canvas shows only the navigation title and the

toolbar. The app itself is fine, as running it in Device Hub confirms, so from step 4 onward verify your progress there.

6. A board this easy to rearrange cuts both ways, because a stray long press while browsing can reshuffle a trip the user had already settled. Let's require an explicit arrange mode. Add a new `@State` property below `stops` and pass it to the container's `isEnabled` parameter:

```
  @State private var stops = Stop.samples
  @State private var isArranging = false
  // ... columns and body
  private var board: some View {
    LazyVGrid(columns: columns, spacing: 12) {
      // ... tiles (steps 2 to 4)
    }
    .reorderContainer(for: Stop.self, isEnabled: isArrangi
      difference.apply(to: &stops)
    }
    .padding(.horizontal)
  }
```

7. Give the user a way in and out of that mode with a toolbar button. Attach `.toolbar()` right after `.navigationTitle()`:

```
.toolbar {
  ToolbarItem(placement: .topBarTrailing) {
    Button(isArranging ? "Done" : "Arrange") {
      isArranging.toggle()
    }
  }
}
```

8. Run the app in Device Hub by pressing *Cmd + R* . The board is now locked: a long press still lifts a tile, but the others no longer make room for it, and nothing changes when you let go. Tap **Arrange** , drag Barcelona from Day 3 up to the first position, and Lisbon and Madrid shift down a day behind it. Tap **Done** to lock the itinerary again.



Figure 2.5: The TripBoard app: the left screen shows the initial itinerary; the right screen shows the board in ar range mode after dragging Barcelona to Day 1

How it works…

Reordering is split across two modifiers on purpose. `.reorderable()` goes on the `ForEach` and marks its content as movable, which is why it is declared on `DynamicViewContent` rather than `View`: it needs a collection of identified items, not a single view. `.reorderContainer(for:)` goes on the container and scopes the interaction: it declares which item type participates, whether the gesture is currently enabled, and what to do with the result. Everything visual in between, the lift under the finger, the gap that tracks the prospective drop position, and the settle animation, comes from the framework. Compare that with the *Lists and Sections* section, where `.onMove()` gave us reordering inside a `List` in edit mode. That modifier remains the simplest choice for a row-based `List`, but it never worked anywhere else; the new pair does, in grids, stacks, and custom layouts alike.

The `move` closure does not receive indices the way `.onMove()` did; it receives a `ReorderDifference` describing the change. Its `sources` property lists the IDs of the items being moved (an array, because containers can support multi-item drags), and `destination.position` is either `.before(id)` or `.end`. Our `apply(to:)` extension turns that description into an array mutation in two moves: one pass through `stops` that removes the moving items while capturing them in order, then a single insert at the destination. Until the closure applies the difference, nothing about the drag has touched our data; the drop is only a request. That is why the day badges are computed from array positions: the moment we apply the difference, the tiles renumber,

and if we applied nothing, the board would simply snap back to the order still stored in `stops`.

For the framework to track a tile from lift to drop, the items need stable identity, which `Stop` provides through `Identifiable` and its `UUID`. The `for: Stop.self` overload keys the difference on that `id`; if a type is not `Identifiable`, or a different key fits better, an `itemID:` key-path overload does the same job. The `isEnabled` parameter earns its place in the signature: a long press is easy to land by accident on a touch screen, so production boards usually gate reordering behind an edit mode or a permission check, exactly like our **Arrange** button. And when a screen has more than one collection, sections in a `List` for example, tag each `ForEach` with `.reorderable(collectionID:)` and declare the identifier type with `reorderContainer(for:in:)`; the difference then carries a `destination.collectionID` telling us which group received the drop.

Under the hood, the reorder container participates in the same drag-and-drop system we met in the *Drag and Drop* section: it acts as its own drag source and drop target, which is why we did not write a single `.draggable()` or `.dropDestination()` here. When a board needs more than reordering, those APIs compose with the container. Reach for `.dragContainer(for:)` when tiles need a custom payload or should travel to other views or apps. Add a container-level `.dropDestination(for:)` when items arriving from elsewhere should land at the visible gap, whose position the session reports through `reorderDestination(for:)`. Use a per-tile `.dropDestination(for:isEnabled:)` when dropping one item onto another should combine them instead of reordering. The reorder modifiers themselves run on iOS, iPadOS,

macOS, and visionOS, and they bring drag-to-reorder to watchOS for the first time, while tvOS sits this one out.

See also

- `reorderable()` documentation: https://x.swiftuicookbook.com/ajwqr
- `reorderContainer(for:isEnabled:move:)` documentation: https://x.swiftuicookbook.com/vbggw
- `ReorderDifference` documentation: https://x.swiftuicookbook.com/bht7w
- `LazyVGrid` documentation: https://x.swiftuicookbook.com/p37to
- Code-along: Build powerful drag and drop in SwiftUI (WWDC 2026): https://x.swiftuicookbook.com/qwj44

3

Prototyping and Previews

I really fell in love with previews when I started working with SwiftUI many years ago. They cut the time between writing code and seeing results from minutes to seconds. Instead of building, launching the simulator, and navigating to the screen I was working on, I could see changes in the preview canvas as I typed. That alone changed how I approach UI development, and it is a habit I want every reader of this book to pick up early.

This chapter is about getting previews into your everyday workflow. We start with `#Preview` and `@Previewable` so you can name multiple previews, drive bindings with live state, and wrap views in `NavigationStack` to match what actually ships. From there, we use `.preferredColorScheme()`, `.environment()`, and the `traits` parameter to verify our layouts under different conditions, we build a reusable `PreviewModifier` so those checks stay consistent across the project, and we turn one `#Preview` into a whole family of variants with the `arguments:` parameter before resizing the device preview to any size we want to test. We then tackle mock data with static samples, `@Previewable` `@State` for interactive components, and `makeSharedContext()` for cached data that any preview can share. Finally, we bring the same `#Preview` macro to a UIKit app, previewing both a full storyboard navigation flow and a single `UILabel` in isolation.

By the end of this chapter, you should be comfortable creating named, interactive previews, verifying your views under different traits and

environment conditions, feeding previews with reusable mock data, and previewing UIKit views and view controllers without leaving the preview canvas.

We'll explore these topics in the following recipes:

- Live Previews with #Preview
- Trait Variations and Device Simulations
- Previews with Mock Data
- UIKit Previews

Technical requirements

The code in this chapter is based on Xcode 27.0 and iOS 27.0. You can download and install the latest version of Xcode from the App Store. All the code examples for this chapter can be found on GitHub at `https://x.swiftuicookbook.com/ch03` .

Live Previews with #Preview

One of the things I appreciate most about SwiftUI development is the tight feedback loop between writing code and seeing results. The `#Preview` macro sits at the center of that loop: it renders our views in the Xcode preview canvas as we type, without building and launching the full app. Getting comfortable with previews early pays off in every chapter that follows.

In this recipe, we will build a notification settings app because it gives us both a top-level screen and a detail screen to preview. That lets us cover the preview behaviors that matter in real work: naming multiple previews, driving a binding with `@Previewable` , and embedding a screen in `NavigationStack` so the preview canvas matches the UI you actually ship.

Getting ready

Create a new SwiftUI iOS app named **PreviewExplorer** .

How to do it...

We will build the views first, then configure previews that let us exercise those views directly in the preview canvas. These are the steps:

1. Open `ContentView.swift` and replace the `ContentView` struct with the following, where the header lives in a `headerView` computed property. Extracting the header keeps the `body` short as we add more controls in the next step:

```
struct ContentView: View {
  var body: some View {
    VStack(spacing: 24) {
      headerView
    }
    .padding()
  }
  var headerView: some View {
    VStack(spacing: 8) {
      Image(systemName: "bell.badge.fill")
        .font(.system(size: 50))
        .foregroundStyle(.blue)
      Text("Notification Center")
        .font(.title2)
        .fontWeight(.bold)
      Text("Manage your alerts")
        .font(.subheadline)
        .foregroundStyle(.secondary)
    }
    .padding()
  }
}
```

> Computed properties like `headerView` are your first tool for organizing a growing view. Give each group of



built-in SwiftUI views its own computed property so the `body` reads like an outline, and move views that stand on their own into separate files to keep the project tidy. In step 3, we extract a whole screen into its own `DetailView` struct.

2. Add a `@State` property and a `Toggle` below the header. Make sure the preview canvas is in **interactive** (or **live**) mode by clicking the Live button (the play-button icon), the first of the four mode buttons at the bottom left of the preview canvas: Live, Resizable, Selectable, and Variants. Live is the default mode for new previews; we use the selectable mode later in this recipe and explore the other two modes in the next recipe. Tap the toggle and the preview responds immediately because it runs our view with real SwiftUI state:

```
struct ContentView: View {
  @State private var notificationsEnabled = true
  var body: some View {
    VStack(spacing: 24) {
      headerView
      Toggle("Enable Notifications", isOn: $notificationsEnable
        .padding(.horizontal)
    }
    .padding()
  }
  // ... headerView
}
```

The four mode buttons sit together at the bottom left of the preview canvas, with the zoom control in the center and three more controls at the bottom right that we explore in the next recipe:

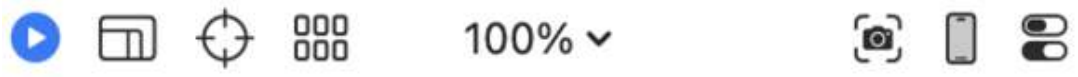


Figure 3.1: The preview canvas controls, with the four mode buttons at the left

3. Below `ContentView`, add a `DetailView` struct with its own toggle and a `@Binding` to a Boolean that controls whether notifications are enabled. Later, we will connect this binding to the `@State` property in `ContentView` so both screens share the same value:

```
struct DetailView: View {
  @Binding var isEnabled: Bool
  var body: some View {
    VStack(spacing: 16) {
      Image(systemName: isEnabled ? "bell.badge.fill" : "bell.sl
        .font(.system(size: 60))
        .foregroundStyle(isEnabled ? .green : .red)
      Text(isEnabled ? "Notifications On" : "Notifications Off")
        .font(.title3)
      Toggle("Enabled", isOn: $isEnabled)
        .labelsHidden()
    }
    .navigationTitle("Settings")
    .navigationBarTitleDisplayMode(.inline)
  }
}
```

4. Wrap the `ContentView` `body` in a `NavigationStack` and add a `NavigationLink` that navigates to `DetailView`. The `Spacer` pushes the controls toward the top of the screen:

```
struct ContentView: View {
  @State private var notificationsEnabled = true
  var body: some View {
    NavigationStack {
      VStack(spacing: 24) {
        headerView
        Toggle("Enable Notifications", isOn: $notificationsEnabl
          .padding(.horizontal)
        NavigationLink("Notification Settings") {
          DetailView(isEnabled: $notificationsEnabled)
        }
        .buttonStyle(.borderedProminent)
```

```
            Spacer()
        }
        .padding()
        .navigationTitle("Preview Explorer")
    }
  }
  // ... headerView
}
```

Make sure the preview canvas is in live mode to test our changes. Tap the toggle, then tap **Notification Settings** to navigate to the detail view. The preview should look like the following:

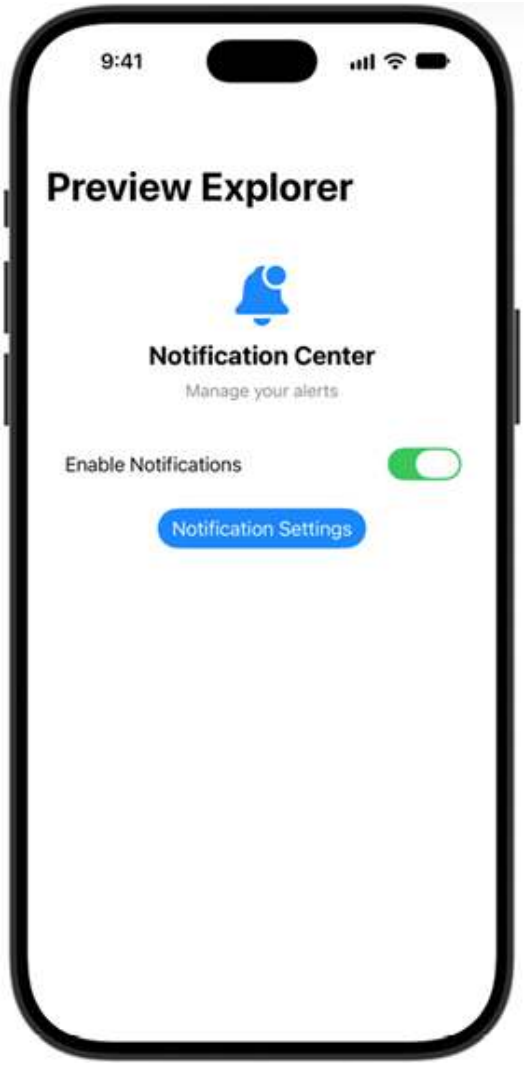


Figure 3.2: The PreviewExplorer app with NavigationStack and toggle

5. Give the default preview a name so we can tell it apart from the one we will add next. Replace the `#Preview` block at the bottom of the file:

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

6. Below the first preview, add a second one for `DetailView`. Since `DetailView` requires a binding, pass `.constant(true)` as a quick placeholder:

```
#Preview("Detail") {
  DetailView(isEnabled: .constant(true))
}
```

   Try tapping the toggle in this preview. It does not respond. A `.constant()` binding is read-only, so the view renders but cannot change its state. In the *Previews with Mock Data* section, we will learn more robust techniques for providing data to previews.

7. Replace the **Detail** preview with one that uses `@Previewable @State` for a live, interactive binding. Wrap `DetailView` in a `NavigationStack` so its `.navigationTitle` is visible:

```
#Preview("Detail — Interactive") {
  @Previewable @State var enabled = true
  NavigationStack {
    DetailView(isEnabled: $enabled)
  }
}
```

   Now, the toggle responds to taps, and the navigation bar shows the title **Settings**. The preview should look like the following:



Figure 3.3: DetailView preview with interactive binding and NavigationStack

8. To see what `@Previewable` does for us, temporarily delete the attribute from the preview we just wrote. Xcode flags the line with the warning '**@State' used inline will not work unless tagged with '@Previewable'** . Click the warning to reveal the fix-it, then click **Apply** next to **Add '@Previewable'** and the preview builds again:



Figure 3.4: The @Previewable warning with its Fix button

9. Switch to **selectable** mode by clicking the Selectable button, the third of the four mode buttons at the bottom left of the preview canvas. Click on any element in the preview, for example the **Notification Center** text, and Xcode highlights the SwiftUI code responsible for that element. This is a fast way to jump between the preview canvas and source code.

   Selection has one limitation worth knowing. When a preview embeds a `NavigationStack`, clicking any element selects the whole stack: Xcode highlights the entire `NavigationStack` expression in the code editor, not the view you clicked. To select individual views, use a preview without a `NavigationStack`, or temporarily comment out the wrapper while you wor k.

How it works...

Each `#Preview` block we added became its own tab in the preview canvas, and the string parameter gave it a label in the tab bar. This closure-based syntax replaced the older `PreviewProvider` protocol and is now the standard way to define previews. We can switch between the two previews we created, one for `ContentView` and one for `DetailView`, without scrolling or navigating to a different file.



New in Xcode 27. When a preview is not enough and you do run the app, Xcode 27 runs it on a simulated device in Device Hub, the window that replaces the standalone Simulator app. By default, Device Hub opens a compact window that shows the app on a device screen. Expand it to see the sidebar, which lists your simulators and paired physical devices, and the inspector, which shows the

> selected device's details, settings, and installed apps. Throughout this book, whenever a step needs the running app, we run it in Device Hub.

The key new tool in this recipe is `@Previewable`, introduced in iOS 18. When we wrote `@Previewable @State var enabled = true` inside the `#Preview` closure, the compiler generated a wrapper `View` struct behind the scenes. That wrapper holds the `@State` property, and the remaining code in the closure becomes its `body`. The result is a fully functional SwiftUI state cycle: the toggle writes to `enabled`, SwiftUI detects the change, and the preview re-renders, all without us writing a dedicated wrapper view. Before `@Previewable`, the only options were `.constant()` bindings (non-interactive) or hand-written wrapper structs (boilerplate). `@Previewable` works with any SwiftUI property wrapper, including `@FocusState` and `@Environment`. We will explore `@State` and `@Binding` in detail in the *State and Binding Basics* section of *Chapter 5*.

We extracted the header into a plain computed property. No special attribute is needed because `headerView` returns a single container view, the `VStack`. Reserve the `@ViewBuilder` attribute for computed properties that return multiple sibling views or choose between branches with `if` or `switch`; it is the attribute SwiftUI applies internally to the `body` property and to the closures of container views like `VStack`. Extracting groups of views into computed properties is a simple way to keep the main `body` readable as a view grows, and it works with previews just as well as inline code.

Wrapping `DetailView` in a `NavigationStack` inside its preview is a pattern we will use throughout the book. Any view that relies on navigation modifiers like `.navigationTitle`, `.navigationBarTitleDisplayMode`, or `.toolbar` needs a parent `NavigationStack` to render those modifiers. We first introduced `NavigationStack` in the *Buttons and Navigation* section of

Chapter 1. The same idea applies to other container-dependent views: if a view is designed to live inside a `List`, `TabView`, or `Form`, embedding it in that container in the preview gives us an accurate picture of the final UI.

See also

- `Preview` documentation: https://x.swiftuicookbook.com/qdfv5
- `Previewable` documentation: https://x.swiftuicookbook.com/ups7k
- `ViewBuilder` documentation: https://x.swiftuicookbook.com/jl67t
- What's new in Xcode 27 (WWDC 2026): https://x.swiftuicookbook.com/wh_672

Trait Variations and Device Simulations

When we build an app, we need to verify that it looks right under a range of conditions: light and dark color schemes, small and large text sizes, portrait and landscape orientation, and different device sizes. Doing this by running the app repeatedly on different simulators is slow. Xcode gives us two faster options: code-based preview configurations that we commit alongside our views, and preview canvas controls for quick, one-off checks.

In this recipe, we will build a card-based view that reacts to color scheme, text size, and orientation. That gives us a practical surface for comparing four approaches: view modifiers like `.preferredColorScheme()` and `.environment()` for targeted overrides, the `traits` parameter for device-level settings like landscape orientation, a custom `PreviewModifier` that bundles

multiple overrides into a single reusable trait, and the `arguments:` parameter that generates a whole family of previews from one `#Preview` block. We also explore the preview canvas controls, which are often faster than code when we just want a quick look at dark mode or a different text size, and we finish by resizing the device preview it self.

Getting ready

Create a new SwiftUI iOS app named **TraitExp lorer** .

How to do it...

We will start by building a card view that reads environment values to adapt its appearance, then add multiple preview configurations to test it, generate a family of previews with `arguments:` , and finally explore the preview canvas controls. These are the steps:

1. Create a `FeatureCard` view above `ContentView` that reads the current color scheme and dynamic type size from the environment. The card changes its icon color in dark mode and hides the subtitle when the user has an accessibility text size enabled:

```
struct FeatureCard: View {
  @Environment(\.colorScheme) var colorScheme
  @Environment(\.dynamicTypeSize) var dynamicTypeSize
  let title: String
  let subtitle: String
  let iconName: String
  var body: some View {
    HStack(spacing: 16) {
      Image(systemName: iconName)
        .font(.title)
        .foregroundStyle(colorScheme == .dark ? .yellow : .blue)
        .frame(width: 50, height: 50)
        .background(
          Circle()
            .fill(colorScheme == .dark
```

```
              ? Color.yellow.opacity(0.2)
              : Color.blue.opacity(0.1))
        )
      VStack(alignment: .leading, spacing: 4) {
        Text(title)
          .font(.headline)
        if !dynamicTypeSize.isAccessibilitySize {
          Text(subtitle)
            .font(.subheadline)
            .foregroundStyle(.secondary)
        }
      }
      Spacer()
    }
    .padding()
    .background(Color(.secondarySystemBackground))
    .clipShape(RoundedRectangle(cornerRadius: 12))
  }
}
```

2. Replace the `ContentView` struct with the following, where the `body` is a `NavigationStack` containing a `ScrollView` that displays four `FeatureCard` instances. Each card describes one of the traits we are going to explore:

```
struct ContentView: View {
  let features: [(String, String, String)] = [
    ("Color Scheme", "Adapts to light and dark", "paintbrush.fil
    ("Dynamic Type", "Responds to text size", "textformat.size")
    ("Size Class", "Compact vs Regular", "rectangle.split.2x1"),
    ("Layout Direction", "LTR and RTL support", "text.alignleft"
  ]
  var body: some View {
    NavigationStack {
      ScrollView {
        VStack(spacing: 12) {
          ForEach(features, id: \.0) { feature in
            FeatureCard(title: feature.0,
                        subtitle: feature.1,
                        iconName: feature.2)
          }
        }
```

```
          .padding()
        }
        .navigationTitle("Trait Explorer")
      }
    }
  }
```

3. Name the default preview and add a second one that forces dark mode using `.preferredColorScheme(.dark)`. Notice how the icon colors switch from blue to yellow and the card backgrounds adapt automatically:

```
#Preview("Default") {
  ContentView()
}
#Preview("Dark Mode") {
  ContentView()
    .preferredColorScheme(.dark)
}
```

4. Add a preview that overrides the dynamic type size to `.xxxLarge` using the `.environment()` modifier. The cards still show their subtitles at this size because `.xxxLarge` is not an accessibility size:

```
#Preview("Large Dynamic Type") {
  ContentView()
    .environment(\.dynamicTypeSize, .xxxLarge)
}
```

The following image shows the Default, Dark Mode, and Large Dynamic Type previews side by side. Notice how the icon colors switch to yellow in dark mode and the cards adapt their background automatically:

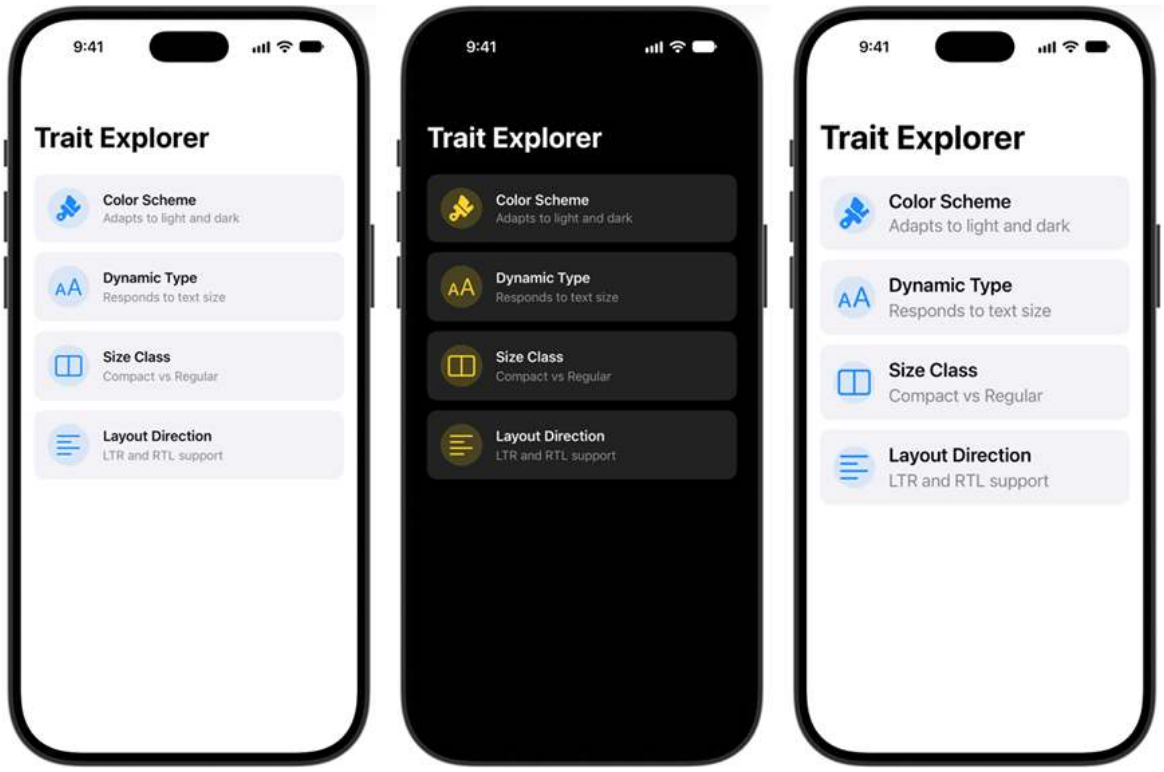


Figure 3.5: Default, Dark Mode, and Large Dynamic Type previews side by side

5. Add a landscape preview using the `traits` parameter of the `#Preview` macro. Unlike the environment modifiers we used in the previous steps, orientation is set through `traits` because it affects the device frame, not the view hierarchy:

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

6. In a real project, we often need to test the same combination of overrides across many previews. Instead of repeating the same modifiers each time, we can create a `PreviewModifier` that bundles them into a single reusable trait. Add an `AccessibilityReady` modifier below `ContentView` that combines dark mode, accessibility-level text, and bold weight:

```
struct AccessibilityReady: PreviewModifier {
  func body(content: Content, context: Void) -> some View {
    content
      .preferredColorScheme(.dark)
      .environment(\.dynamicTypeSize, .accessibility1)
      .environment(\.legibilityWeight, .bold)
  }
}
extension PreviewTrait where T == Preview.ViewTraits {
  @MainActor static var accessibilityReady: Self =
    .modifier(AccessibilityReady())
}
```

7. Use the new trait in a preview. The three overrides are now a single `.accessibilityReady` trait that any preview in the project can reuse:

```
#Preview("Accessibility Ready", traits: .accessibilityReady) {
  ContentView()
}
```

In this preview, the subtitles disappear because `.accessibility1` is an accessibility size, the icons turn yellow because we are in dark mode, and the text is bolder. The preview should look like the following:

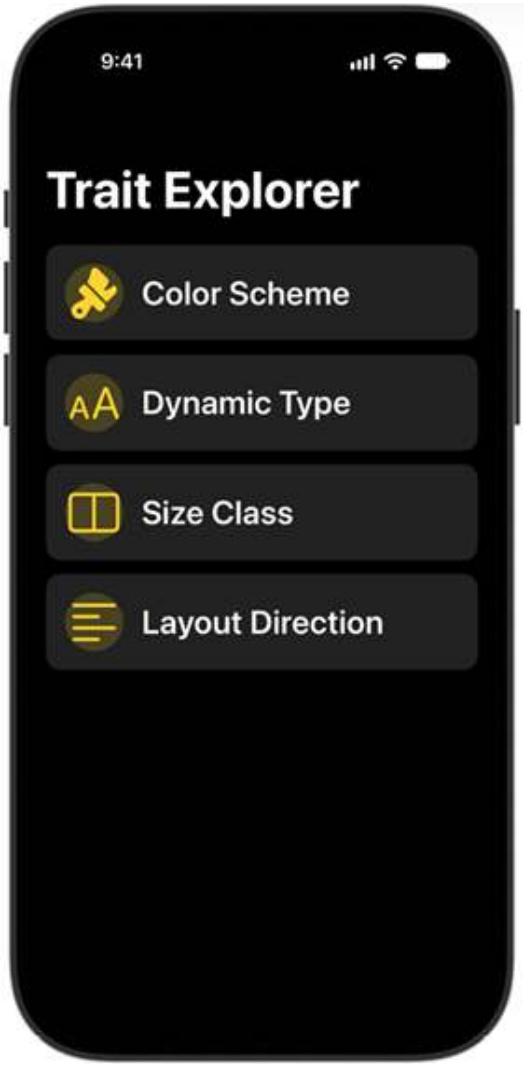


Figure 3.6: Accessibility Ready preview with dark mode, large type, and bold weight

8. One `#Preview` block can also fan out into several previews at once. Pass an array to the `arguments:` parameter and give the closure a parameter; the preview canvas renders one preview per element. Add a parameterized preview that renders `ContentView` at three dynamic type sizes, from a standard size up to an accessibility size:

```
#Preview("Dynamic Type", arguments: [
  DynamicTypeSize.large, .xxxLarge, .accessibility3
]) { size in
  ContentView()
```

```
        .environment(\.dynamicTypeSize, size)
    }
```

The parameterized preview appears as a single **Dynamic Type** tab, and the preview canvas renders one variant per array element, each labeled with its argument value. Note that while a parameterized preview is selected, resizable mode is not available:

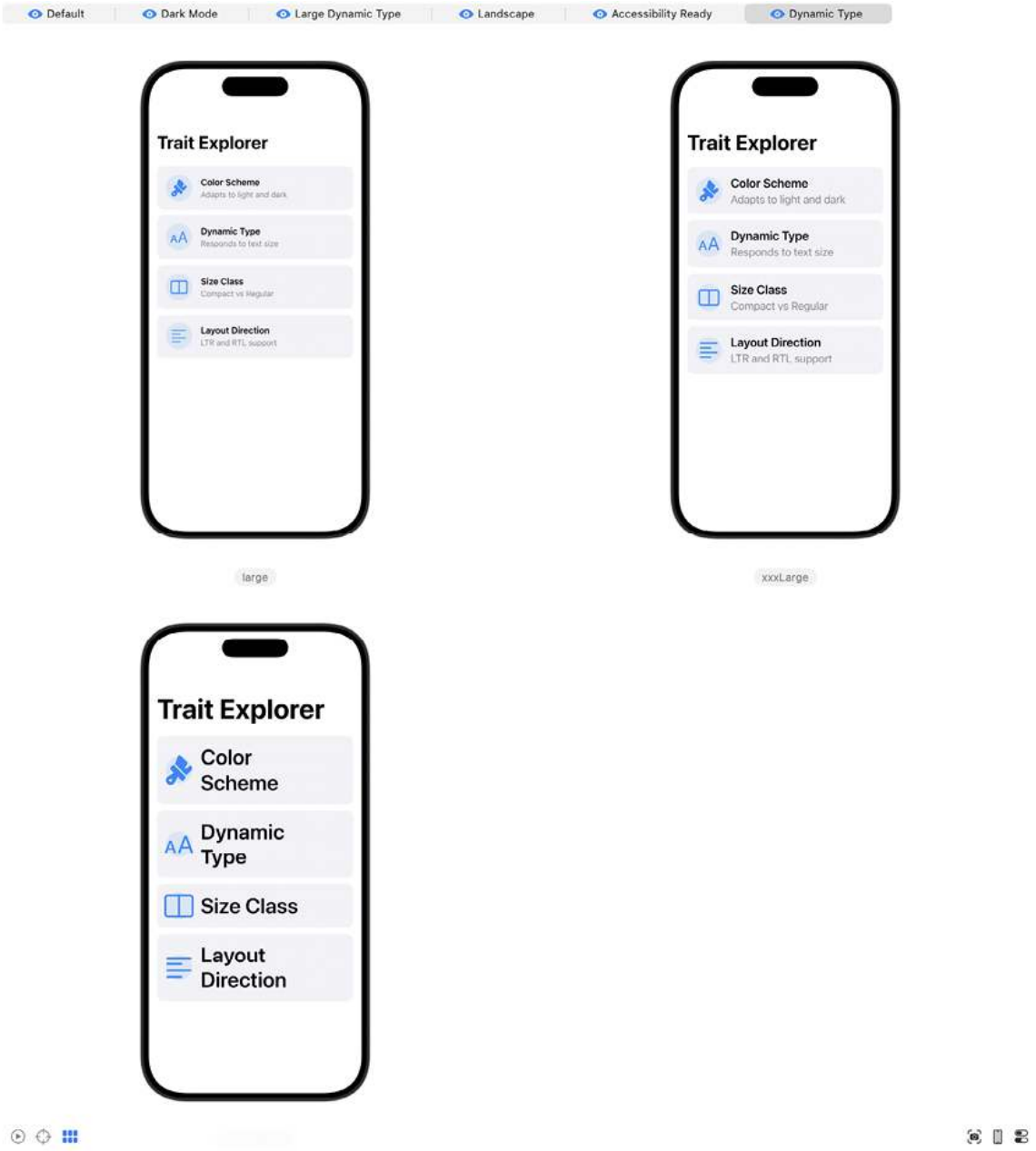


Figure 3.7: The parameterized Dynamic Type preview rendering three sizes



New in Xcode 27. Parameterized previews arrive with Xcode 27: the `arguments:` parameter accepts an array of any type, and the `#Preview` macro expands it into one preview per element. One block now covers a whole matrix of states that used to require a separate `#Preview` each. The macro is available from iOS 26, so you can use it in projects that still support the previous release.

9. Besides code-based previews, the preview canvas offers quick, interactive controls for the same traits. At the bottom right of the preview canvas sit three buttons: one exports a screenshot of the preview (the camera icon), the **Preview Device** button (the phone icon) picks the device, and the **Device Settings** button (the two-toggles icon) opens a pop-up with toggles for color scheme, contrast, control borders, orientation, and a dynamic type slider. The **Contrast** and **Control Borders** toggles arrived with Xcode 27 and preview the **Increase Contrast** and **Button Shapes** accessibility settings, which a view reads through the `colorSchemeContrast` and `accessibilityShowButtonShapes` environment values. Click **Device Settings** , toggle **Color Scheme** on to switch the preview to dark mode, and drag the dynamic type slider to see the cards react in real time:

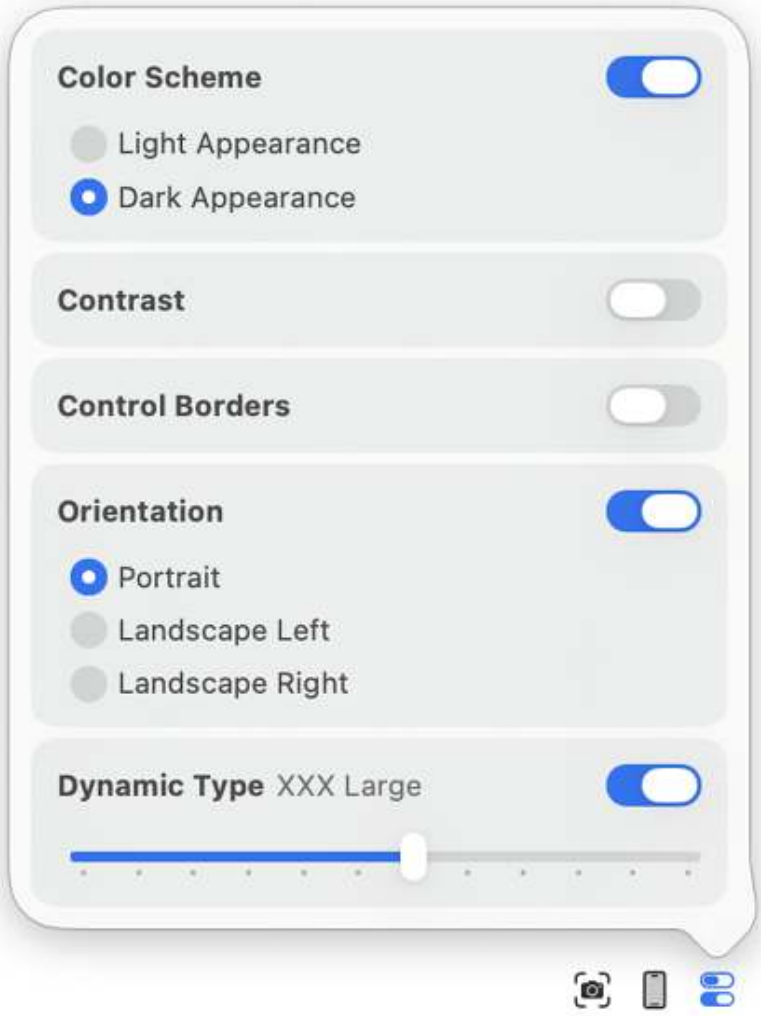


Figure 3.8: Device Settings pop-up with color scheme and dynamic type controls

10. First, select the **Default** preview tab so the variants start from a preview without traits; a preview whose traits already force dark mode or landscape would override the variants. Then, switch to **variants** mode by clicking the Variants button, the fourth mode button (the grid icon) at the bottom left of the preview canvas. A pop-up appears with five options: **Color Scheme Variants** , **Contrast Variants** , **Control Borders Variants** , **Orientation Variants** , and **Dynamic Type Variants** . Select **Color Scheme Variants** to see the

app in both light and dark mode side by side. Click on any single variant to expand it, and click **Back** to return to the side-by-side view:

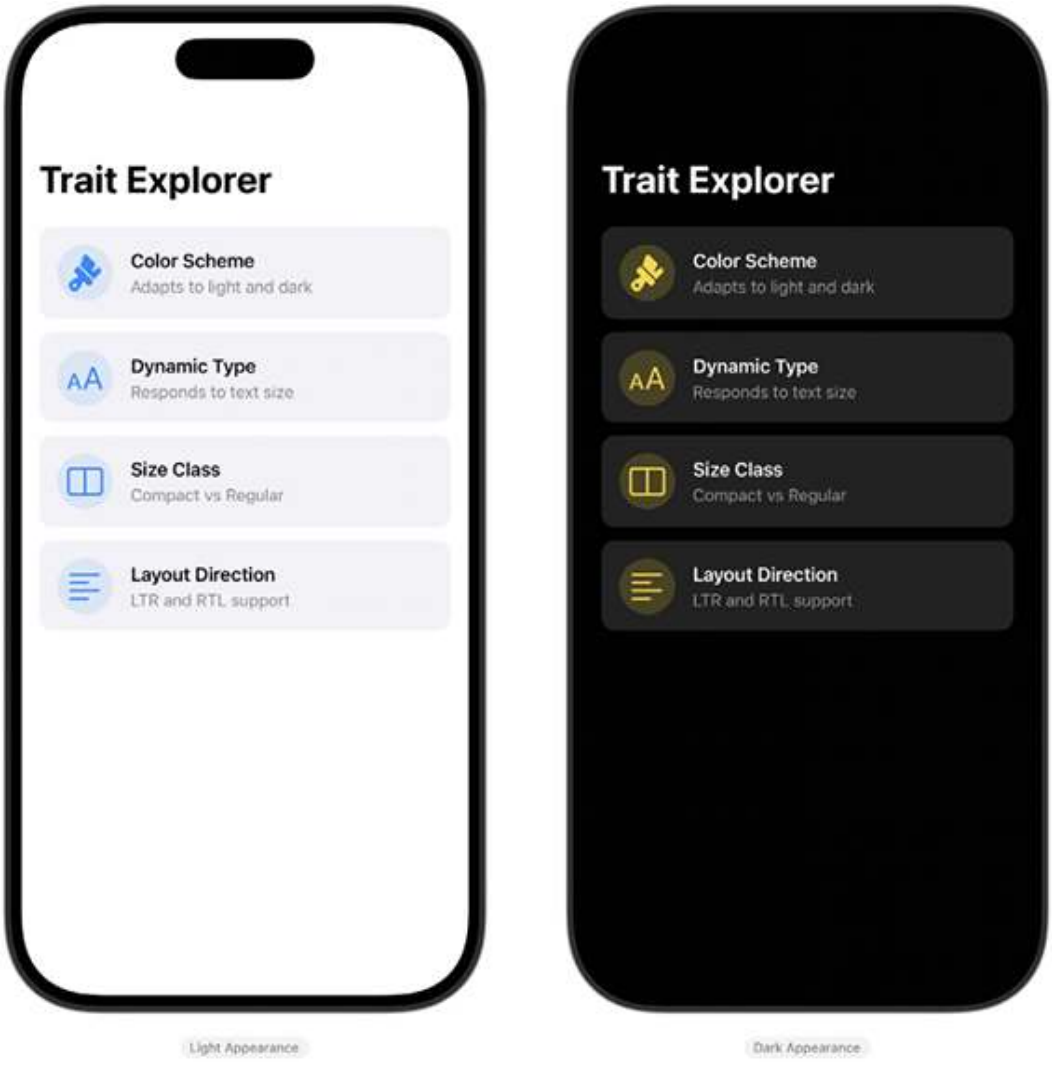


Figure 3.9: Color Scheme Variants showing light and dark mode side by side

11. Now, select **Dynamic Type Variants** from the same Variants menu. The preview canvas shows the app at several text sizes simultaneously, from the smallest to the largest accessibility size. Notice how our `FeatureCard` hides the subtitle at accessibility sizes:

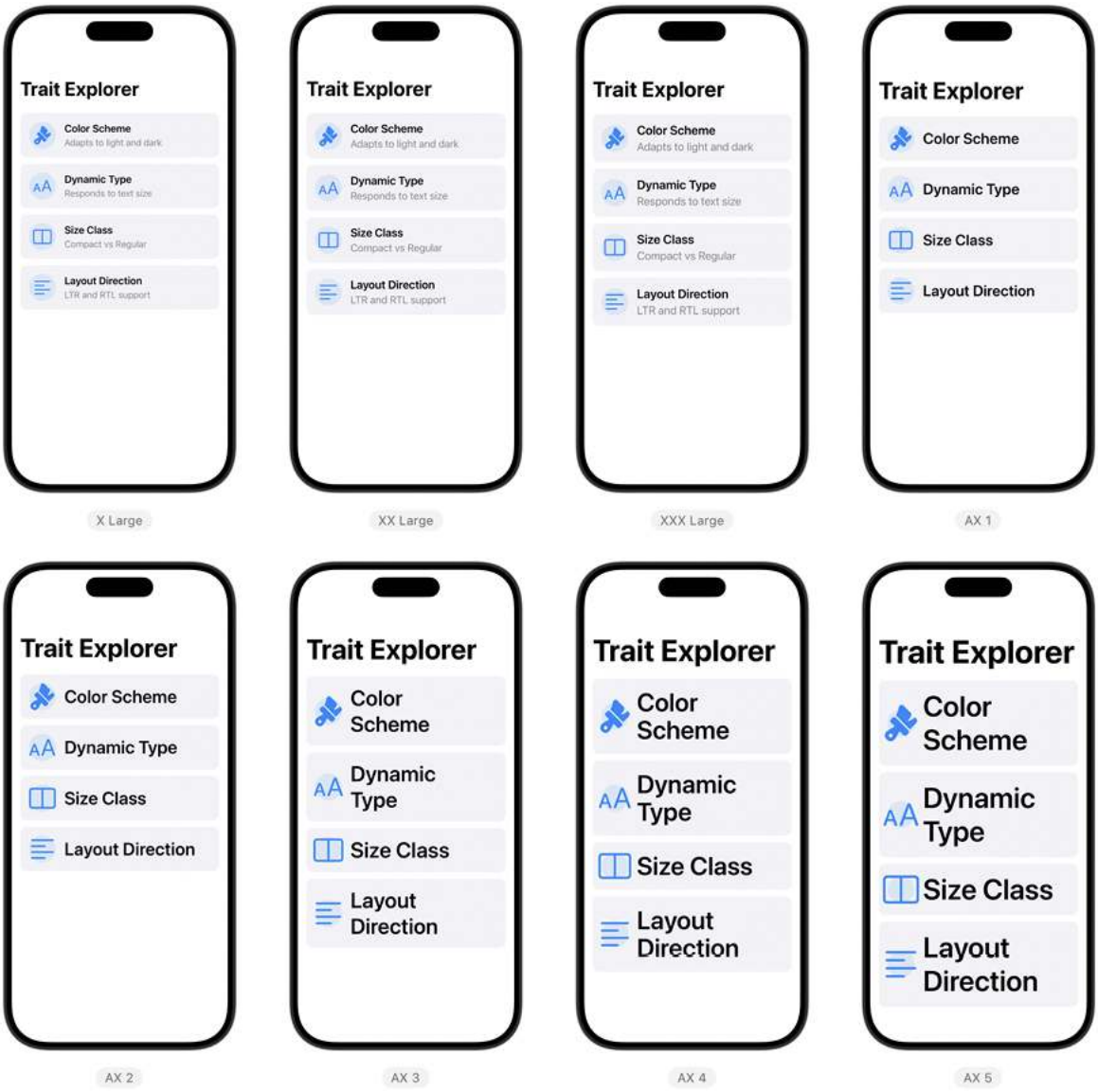


Figure 3.10: Dynamic Type Variants showing the app at different text sizes

12. Use the **Preview Device** button (the phone icon) at the bottom right of the preview canvas to switch between different iPhone and iPad models. The default setting is **Automatic** , which uses the currently selected simulator. You can also select a connected physical device to preview directly on your iPhone or iPad:

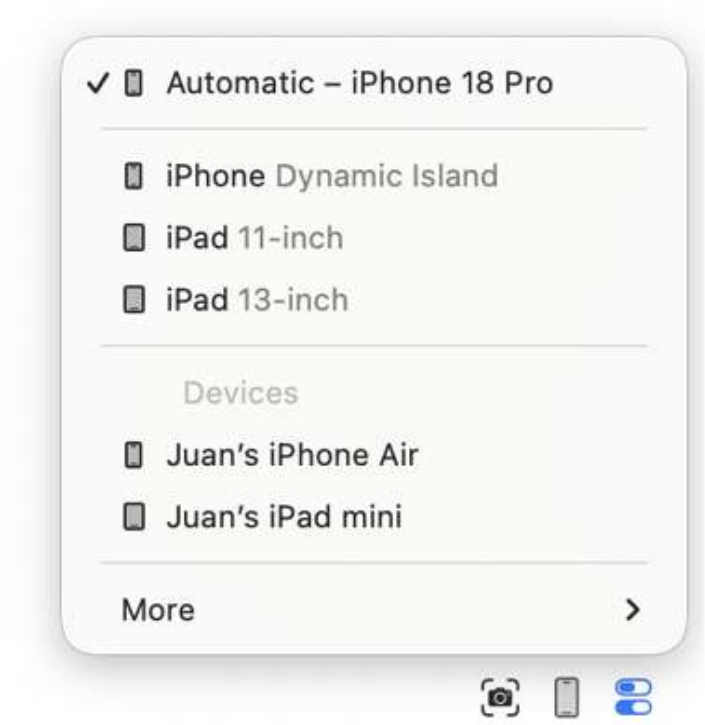


Figure 3.11: Preview Device selector with available simulators and device families

13. Finally, switch to **resizable** mode by clicking the Resizable button, the second of the four mode buttons at the bottom left of the preview canvas. The preview loses the device frame and gains a resize handle on each edge, and the bottom of the preview canvas shows the current dimensions next to the zoom level. Drag any handle to test the view at any size; the readout updates as you drag, the preview dims until you let go and then re-renders at the new size, and it stays as interactive as live mode:

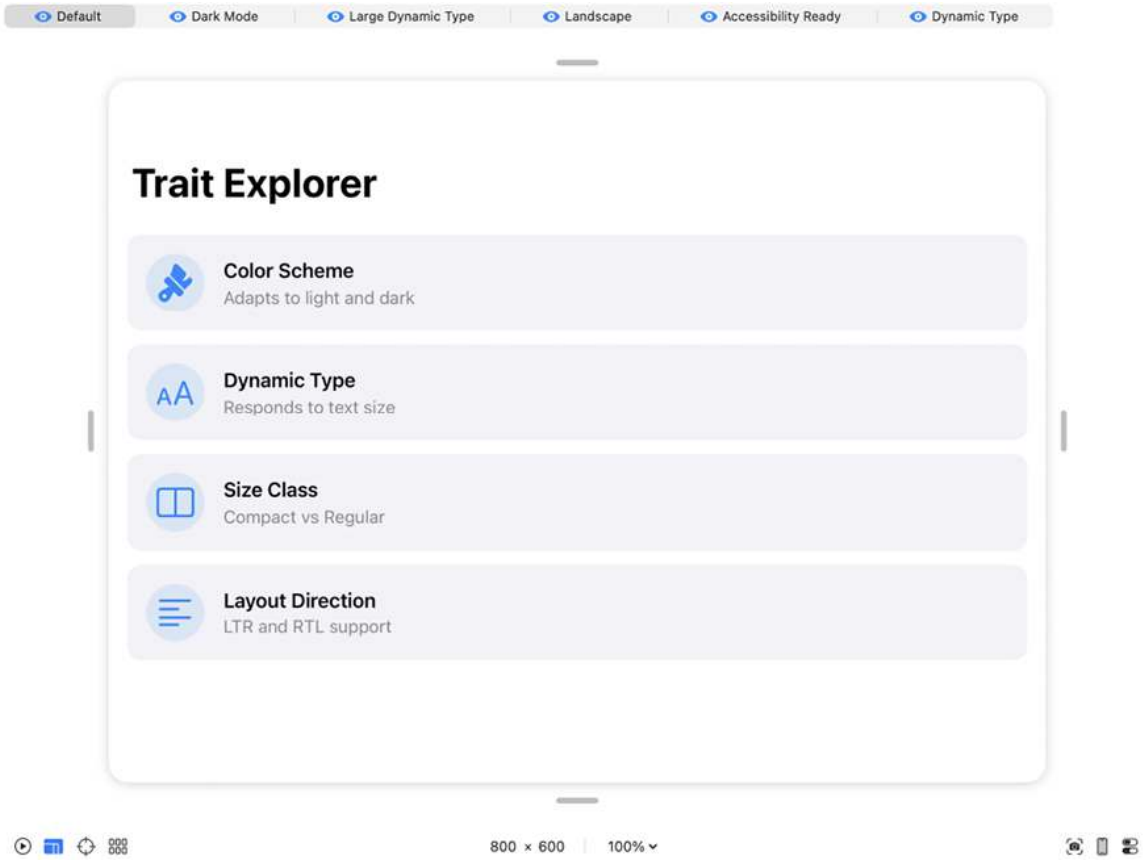


Figure 3.12: Resizing the device preview with the resize handles



New in Xcode 27. The Xcode 27 preview canvas adds resizable mode: like live mode, but with resize handles and a live size readout, so you can test any dimensions without committing to a fixed device frame.

14. As a challenge, reorganize **TraitExplorer** so each group of built-in SwiftUI views in `ContentView` gets its own computed property, and move `FeatureCard` and `AccessibilityReady` int o their own files.

How it works...

In practice, these are two complementary tools. Code-based previews using `.preferredColorScheme()`, `.environment()`, and `traits` are worth committing because they capture the states every team member should verify. The preview canvas controls (Device Settings, Variants, Preview Device, and resizable mode) are faster for one-off exploration but leave no trace in the code.

In `FeatureCard`, we read two environment values to make the card adaptive: `colorScheme` for picking icon and background colors, and `dynamicTypeSize.isAccessibilitySize` for hiding the subtitle at accessibility text sizes. The `@Environment` property wrapper gives a view read access to values that SwiftUI manages for us, and when we override them in a preview with `.environment(\.dynamicTypeSize, .xxxLarge)`, SwiftUI injects the new value into the entire view hierarchy below that modifier. We will explore `@State` and `@Binding` in the *State and Binding Basics* section of *Chapter 5*, and custom environment keys in the *Reading and Defining Environment Values* section of *Chapter 6*.

Orientation works differently from color scheme and dynamic type. It is not an environment value that we inject with `.environment()`, but a device-level trait that we pass through the `traits` parameter of the `#Preview` macro. The `traits` parameter also supports `.sizeThatFitsLayout` for component-sized previews and `.fixedLayout(width:height:)` for custom dimensions.

Parameterized previews add one more axis of variation. When we passed `arguments:` an array of dynamic type sizes, the `#Preview` macro expanded the array into one preview per element and handed each element to the closure parameter. The distinction from the variants mode is worth keeping in mind: variant modes vary a device setting, while parameterized previews vary our own input data, so the two compose naturally. The `arguments:` array accepts any type, including our own

model values, which makes this the code-based way to review one view across a set of realistic inputs.

Our `AccessibilityReady` modifier bundled three overrides into a single reusable type by conforming to the `PreviewModifier` protocol. Its `body(content:context:)` method receives the preview content and returns a modified version, and by extending `PreviewTrait` with a static property we created a clean `.accessibilityReady` shorthand that any `#Preview` in the project can use. This is especially valuable when a project has a standard set of test scenarios that every screen should be previewed against. `PreviewModifier` also supports a `makeSharedContext()` method for creating expensive resources like database containers once and caching them across previews. We will use that capability in the *Previews wit h Mock Data* section.

See also

- `PreviewModifier` documentation: https://x.swiftuicookbook.com/7nvzi
- `PreviewTrait` documentation: https://x.swiftuicookbook.com/5mygt
- `EnvironmentValues` documentation: https://x.swiftuicookbook.com/trzq2
- *Previews in Xcode* article: https://x.swift uicookbook.com/1hhvm

Previews with Mock Data

In a real project, the data we display usually comes from a remote server, a database, or some other asynchronous source. Fetching real data during previews makes them slow, unreliable, and dependent on a network

connection. The better approach is to feed our previews with mock data that exists only in development builds and never ships to the App Store.

In this recipe, we will build a recipe list app because it gives us preview scenarios that show up constantly in production code: a full list, an empty state, and a single interactive component. That makes it a good place to compare three mock data techniques: static samples defined on the model type, `@Previewable` `@State` for interactive bindings, and `PreviewModifier` with `makeSharedContext()` for reusable , cached environments.

Getting ready

Create a new SwiftUI iOS app named **MockDataPreviews** .

How to do it...

We will define a data model with built-in sample data, build two views that display it, wire the app entry point to the sample data, and then configure previews that exercise different data scenarios. These are the steps:

1. In `ContentView.swift` , create a `Recipe` struct with a nested `Difficulty` enum above the `ContentView` struct. Each difficulty case provides a color and an SF Symbol icon. Add a `samples` array and a `sample` shorthand as static properties so mock data lives directly on the type:

```
import SwiftUI
struct Recipe: Identifiable {
  let id = UUID()
  var name: String
  var cookTime: Int
  var difficulty: Difficulty
  var isFavorite: Bool = false
  enum Difficulty: String, CaseIterable {
    case easy = "Easy"
    case medium = "Medium"
```

```
      case hard = "Hard"
      var color: Color {
        switch self {
        case .easy: .green
        case .medium: .orange
        case .hard: .red
        }
      }
      var icon: String {
        switch self {
        case .easy: "flame"
        case .medium: "flame.fill"
        case .hard: "flame.circle.fill"
        }
      }
    }
  }
  extension Recipe {
    static let samples: [Recipe] = [
      Recipe(name: "Spaghetti Carbonara", cookTime: 25,
             difficulty: .medium, isFavorite: true),
      Recipe(name: "Caesar Salad", cookTime: 10,
             difficulty: .easy),
      Recipe(name: "Beef Wellington", cookTime: 120,
             difficulty: .hard),
      Recipe(name: "Chocolate Souffle", cookTime: 45,
             difficulty: .hard, isFavorite: true),
      Recipe(name: "Grilled Cheese", cookTime: 8,
             difficulty: .easy)
    ]
    static let sample = samples[0]
  }
  // ... ContentView struct
```

2. Below the `Recipe` extension and above `ContentView`, create a `RecipeCardView` that displays one recipe. It takes the recipe and a `@Binding` to a Boolean for the favorite toggle:

```
struct RecipeCardView: View {
  let recipe: Recipe
  @Binding var isFavorite: Bool
  var body: some View {
    HStack(spacing: 12) {
```

```
      Image(systemName: recipe.difficulty.icon)
        .font(.title2)
        .foregroundStyle(recipe.difficulty.color)
        .frame(width: 44, height: 44)
        .background(recipe.difficulty.color.opacity(0.15))
        .clipShape(RoundedRectangle(cornerRadius: 10))
      VStack(alignment: .leading, spacing: 4) {
        Text(recipe.name)
          .font(.headline)
        HStack {
          Label("\(recipe.cookTime) min", systemImage: "clock")
            .font(.caption)
            .foregroundStyle(.secondary)
          Text(recipe.difficulty.rawValue)
            .font(.caption)
            .foregroundStyle(recipe.difficulty.color)
        }
      }
      Spacer()
      Button {
        isFavorite.toggle()
      } label: {
        Image(systemName: isFavorite ? "heart.fill" : "heart")
          .foregroundStyle(isFavorite ? .red : .gray)
      }
    }
    .padding()
    .background(Color(.secondarySystemBackground))
    .clipShape(RoundedRectangle(cornerRadius: 12))
  }
}
```

3. Replace the `ContentView` struct with the following, where a `@State` array of recipes drives a `NavigationStack` that shows a scrollable list of recipe cards, or a `ContentUnavailableView` when the array is empty:

```
struct ContentView: View {
  @State var recipes: [Recipe]
  var body: some View {
    NavigationStack {
      Group {
        if recipes.isEmpty {
```

```
            ContentUnavailableView(
              "No Recipes",
              systemImage: "fork.knife",
              description: Text("Add your first recipe to get star
          } else {
            ScrollView {
              LazyVStack(spacing: 12) {
                ForEach(Array(recipes.enumerated()),
                        id: \.element.id) { index, recipe in
                  RecipeCardView(recipe: recipe,
                                 isFavorite: $recipes[index].isFav
                }
              }
              .padding()
            }
          }
        }
        .navigationTitle("My Recipes")
      }
    }
}
```



In a production app, the `Recipe` model and the `RecipeCardView` view would live in their own files, `Recipe.swift` and `RecipeCardView.swift`, each starting with `import SwiftUI`. We keep them in `ContentView.swift` so the recipe stays focused on previews.

4. Our `ContentView` now requires a `recipes` argument, so the app entry point no longer compiles. Open `MockDataPreviewsApp.swift` and pass the sample data:

```
@main
struct MockDataPreviewsApp: App {
  var body: some Scene {
    WindowGroup {
      ContentView(recipes: Recipe.samples)
```

```
    }
  }
}
```

In a real app, this argument would come from a database or a network layer; `Recipe.samples` keeps the app runnable while we focus on previews.

5. Add the first preview, passing `Recipe.samples` directly to `ContentView`. This is the simplest way to provide mock data:

```
#Preview("Recipe List") {
  ContentView(recipes: Recipe.samples)
}
```

6. Add an empty-state preview to verify the `ContentUnavailableView` appears when there are no recipes:

```
#Preview("Empty State") {
  ContentView(recipes: [])
}
```

7. Add a preview for a single `RecipeCardView` using `@Previewable` `@State` so the favorite heart button is interactive. The `.sizeThatFitsLayout` trait sizes the preview to fit the card instead of showing a full device frame:

```
#Preview("Single Card", traits: .sizeThatFitsLayout) {
  @Previewable @State var isFavorite = false
  RecipeCardView(recipe: .sample, isFavorite: $isFavorite)
    .padding()
}
```

> The `.sizeThatFitsLayout` trait only takes effect when the preview canvas is in **selectable** mode (the third of the



four mode buttons at the bottom left of the preview canvas). In the default **live** mode, the preview still renders inside a full device frame. Switch to selectable mode to see the card without the device chrome, but keep in mind that selectable mode is not interactive.

Tap the heart in this preview and it toggles between filled and outlined, confirming that `@Previewable` creates a live binding.

8. Passing `Recipe.samples` directly works well for a single file, but in a larger project we would repeat the same setup across many previews. A `PreviewModifier` solves this by creating the mock data once and caching it. Add a `SampleRecipes` modifier below `ContentView` that uses `makeSharedContext()` to prepare the data:

```
struct SampleRecipes: PreviewModifier {
  static func makeSharedContext() throws -> [Recipe] {
    Recipe.samples
  }
  func body(content: Content, context: [Recipe]) -> some View {
    ContentView(recipes: context)
  }
}
extension PreviewTrait where T == Preview.ViewTraits {
  @MainActor static var samples: Self =
    .modifier(SampleRecipes())
}
```

9. Replace the first preview so it uses the new `.samples` trait instead of passing data directly. The result is the same, but now any preview in the project can reuse the trait with a single line:

```
#Preview("Recipe List", traits: .samples) {
  ContentView(recipes: [])
}
```

Note that we pass an empty array on purpose. The closure's `ContentView(recipes: [])` becomes the `content` that `SampleRecipes` receives, and its `body(content:context:)` ignores that content and returns a fresh `ContentView(recipes: context)` built from the cached samples. The trait, not the closure, decides what data this preview shows.

If everything went well, the three previews should look like the following:

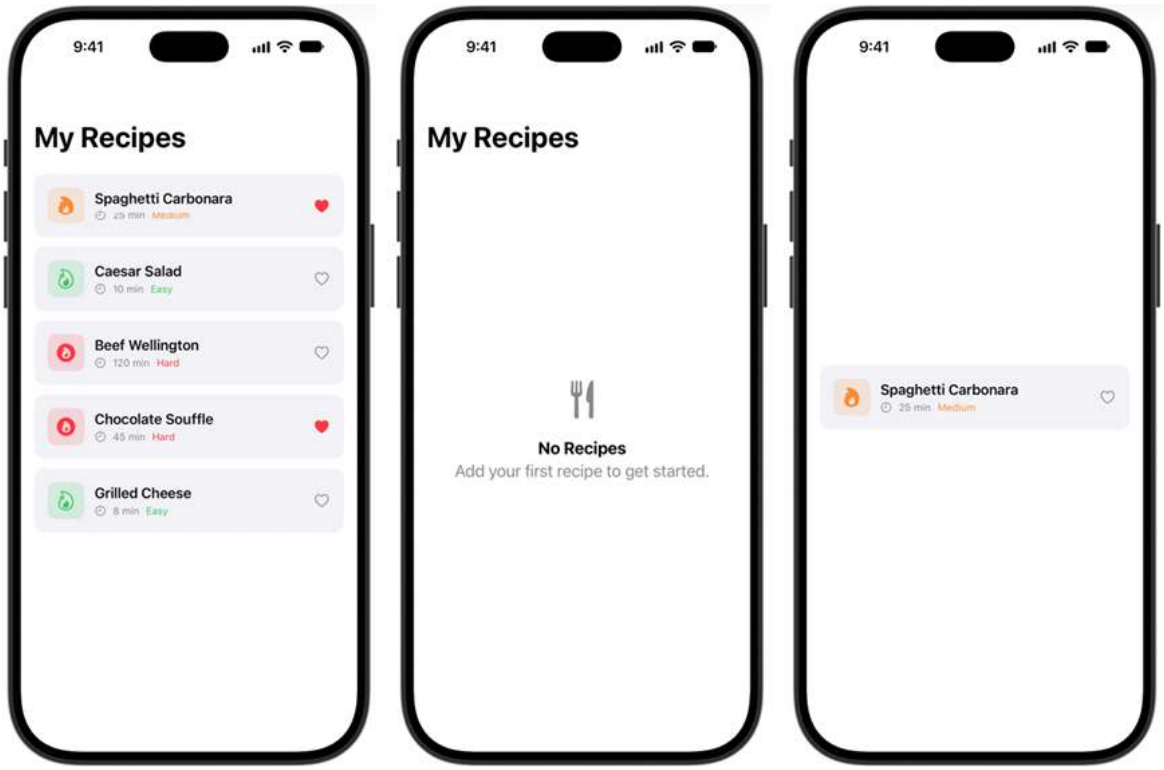


Figure 3.13: The recipe list, empty s tate, and single card previews

How it works...

The simplest mock data technique is defining static properties directly on the model type. We added `Recipe.samples` and `Recipe.sample` as static lets, so any preview can reference them without importing a separate file or decoding JSON. This pattern is fast to write and easy to maintain because

the sample data lives next to the type definition. For projects that need mock images or large JSON payloads, Xcode provides a Preview Content folder whose contents are automatically excluded from release builds. You can verify this under the project's Build Settings, in the Development Assets field.

The `@Previewable` macro, which we first used in the *Live Previews with #Preview* section, made the favorite heart button interactive in the Single Card preview. Without it, we would need `.constant(false)` (non-interactive) or a dedicated wrapper view. Combined with the `.sizeThatFitsLayout` trait, `@Previewable` gives us a compact, interactive preview for a single component without the overhead of a full device frame.

The `PreviewModifier` protocol takes mock data one step further by making it reusable and cached. The `makeSharedContext()` method is static, which means the framework calls it once and shares the result across every preview that uses the same modifier type. In our recipe, the context is a simple array, but in a production app it could be an in-memory SwiftData `ModelContainer` (see the *SwiftData* section of *Chapter 7*) or a pre-populated network cache. The `body(content:context:)` method receives both the preview content and the cached context, and returns a modified view. By extending `PreviewTrait` with a static `.samples` property, we created a shorthand that reads like a built-in trait: `#Preview("Recipe List", traits: .samples)`.

The empty-state preview is a good reminder that previews earn their keep when they cover the states you are least likely to exercise manually. In a real project, I would add a few more checks: single-item data, long recipe names, and an error or loading state. Each of these costs only a few lines but can catch layout problems that are ha rd to spot with sample data alone.

See also

- `PreviewModifier` documentation: https://x.swiftuicookbook.com/7nvzi
- `ContentUnavailableView` documentation: https://x.swiftuicookbook.com/4139k
- `PreviewTrait` documentation: https://x.swiftuicookbook.com/5mygt

UIKit Previews

Most production iOS apps still have a significant amount of UIKit code, and rewriting all of it in SwiftUI just to get previews is rarely practical. The good news is that the `#Preview` macro works with UIKit out of the box. Xcode wraps UIKit views and view controllers in SwiftUI representables behind the scenes, so they render in the preview canvas just like native SwiftUI views.

In this recipe, we will add preview macros to an existing UIKit app so we can test both ends of the workflow: previewing the full navigation flow from a storyboard, and previewing a single `UILabel` in isolation. This is useful if you are maintaining a UIKit codebase and want the same fast feedback loop we have been using throughout the chapter.

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. Download or clone the code for this chapter from GitHub as explained in the Technical requirements section, and open the Xcode project located in the **Starter** folder of this recipe.

How to do it...

We will use the provided UIKit app and add a couple of preview macros to its `ViewController.swift` file. These are the steps:

1. Run the app in Device Hub to get familiar with the navigation flow before we preview it. It is a simple app with a main view and a button that, when tapped, pushes a detail view on a navigation stack. Tapping the back button on the navigation bar returns to the main view.
2. Open the **Main** storyboard to see the UI for the whole app. Then, open the `ViewController.swift` file. There is not much code there because all the UI elements are defined in the storyboard.
3. Add the following code at the end of the `ViewController.swift` file. We instantiate the initial view controller from the **Main** storyboard, which gives us the full `UINavigationController` hierarchy rather than a single screen:

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

4. As soon as you add the `#Preview` macro, the preview canvas appears. Make sure the preview canvas is in live mode and interact with the app. You can tap the button to navigate to the detail view and tap back to return:



Figure 3.14: The preview canvas showing the live preview of a UIKit app



In some Xcode versions, storyboard-based previews can get stuck loading indefinitely. Storyboard previews are also slower to render than SwiftUI previews, and the first one after launching Xcode can take around twenty seconds, so give the spinner time before you assume it is stuck. If the spinner never finishes, quit and reopen Xcode, then open only `ViewController.swift` and let its previews finish rendering before you open the storyboard again.

5. Add a second preview at the end of the file that creates and configures a `UILabel`. This shows how to preview individual UIKit views without a full view controller:

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

6. Click the **UILabel** tab at the top of the preview canvas to switch to the label preview. Change the text or the font in the code and the preview updates in real time:



Figure 3.15: The preview canvas sh owing the live preview of a UILabel

How it works...

The `#Preview` macro accepts any `UIViewController` or `UIView` returned from its closure. Xcode handles the bridging to SwiftUI automatically, so we do not need to write `UIViewRepresentable` or `UIViewControllerRepresentable` wrappers just for previewing.

In our first preview, we instantiated the initial view controller from the **Main** storyboard using `UIStoryboard(name:bundle:).instantiateInitialViewController()`. Because the

storyboard's initial view controller is a `UINavigationController`, the preview renders the full navigation hierarchy, and we can interact with it in live mode just as we would in Device Hub.

Our second preview created a `UILabel` directly, configured its `text` and `font` properties, and returned it. This pattern works for any `UIView` subclass: instantiate it, configure it, and return it. The preview canvas renders the view at its intrinsic content size. This is useful for previewing individual components like buttons, labels, or custom views without setting up a full view controller.

The same preview features we used earlier in the chapter still apply on the UIKit side. Once a view or view controller is rendering in the preview canvas, you can name the preview, rotate it with `traits: .landscapeLeft`, or force dark mode with `.preferredColorScheme(.dark)` w ithout spinning up another simulator.

See also

- `UIViewRepresentable` documentation: https://x.swiftuicookbook.com/ba2oz
- `UIViewControllerRepresentable` documentation: https://x.swiftuicookbook.com/s3i25
- `Preview` documentation: https://x.swiftuicookbook.com/qdfv5

4

User Input and View Composition

I have always thought of user input as the part of app development that separates a prototype from a real product. A screen can look polished, but as soon as someone needs to type a name, pick a date, or confirm a deletion, the interaction quality matters more than the layout. I have seen it in apps I use every day: text fields that lag a beat behind the keyboard, date pickers that quietly revert when the sheet closes, and confirmation dialogs that vanish on a stray tap before the user has read a word. In UIKit, getting this right meant juggling delegate methods, target-action wiring, and manual state synchronization. SwiftUI's binding model simplifies all of that: each control reads from and writes to a source of truth, and the screen stays in sync without extra glue code.

In this chapter, we work through the places where interaction design shows up fastest in a real app. We build a registration form using `Form`, `@FocusState` for keyboard navigation, `SecureField` for passwords, and `.textContentType()` for autofill hints. We then use `.alert()` and `.confirmationDialog()` with `ButtonRole` to handle delete confirmations, rename prompts, and multi-option action sheets, before moving to their item-driven and error-driven forms. From

there, we organize secondary actions with `.contextMenu()`, `ControlGroup`, nested `Menu`, and `DisclosureGroup`. We explore toolbars through `ToolbarTitleMenu`, `ToolbarItemGroup`, `.toolbarRole(.editor)`, and the Liquid Glass bottom bar, then make them adapt to tight space with `visibilityPriority`, `ToolbarOverflowMenu`, and `.toolbarMinimizationBehavior`. Finally, we decompose a growing screen into reusable pieces using a custom `ViewModifier`, `@ViewBuilder` and `@ContentBuilder` computed properties, and a generic container view.

By the end of this chapter, you should be comfortable building production forms with keyboard focus management, presenting alerts and dialogs for user decisions, organizing actions in menus and toolbars, and extracting reusable view components to keep your screens readable as they grow.

The recipes we'll cover in this chapter are as follows:

- Forms
- Alerts and Confirmation Dialogs
- Menus and Disclosure Groups
- Toolbars and the Liquid Glass Design
- Reusable Views and Modifiers

Technical requirements

The code in this chapter is based on Xcode 27.0 and iOS 27.0. You can download and install the latest version of Xcode from the App Store. All the code examples for this chapter can be found on GitHub at https://x.swiftuicookbook.com/ch04.

Forms

I have spent more hours than I care to admit building forms in Interface Builder, wrangling Auto Layout constraints just to line up a few text fields and toggles. Nearly every app has at least one screen where the user needs to enter information, whether that is a login page, a profile editor, a settings panel, or a checkout flow. In SwiftUI, `Form` lets us spend less time wrestling with row alignment and more time on the parts that actually shape the experience, such as focus behavior, validation, and the order in which the screen asks for information.

In this recipe, we will build a registration screen with the usual mix of personal details, contact information, and account credentials. That gives us a realistic place to solve two problems that show up quickly in production forms: moving through fields without fighting the keyboard, and keeping the submit action tied to the state of the data. Along the way, we will use `@FocusState` for field-to-field navigation, `Group` to scope shared form behavior, and `LabeledContent` for the read-only preview rows that still need to align cleanly with the editable controls.

Getting ready

Create a new SwiftUI iOS app named **RegistrationForm** .

How to do it...

Rather than introducing each control in isolation, we will place them inside a form that feels like a real registration screen. We will start

with the basic form structure, add sections one at a time, and finish with focus management and validation. These are the steps:

1. Open `ContentView.swift` and replace the `ContentView` struct with the following. It declares the `@State` properties for the personal information section, and its `body` is a `NavigationStack` containing a `Form` and its first `Section` :

```
struct ContentView: View {
  @State private var firstName = ""
  @State private var lastName = ""
  @State private var birthDate = Date()
  var body: some View {
    NavigationStack {
      Form {
        Section("Personal Information") {
          TextField("First Name", text: $firstName)
          TextField("Last Name", text: $lastName)
          DatePicker("Date of Birth",
                     selection: $birthDate,
                     in: ...Date.now,
                     displayedComponents: .date)
        }
      }
      .navigationTitle("Register")
    }
  }
}
```

   Make sure the preview canvas is in live mode; the **Register** form shows its **Personal Information** section with the two name fields and the date picker.

2. Add `@State` properties for the contact fields, then add a second section below **Personal Information** . The email field gets `.keyboardType(.emailAddress)` to show the email keyboard,

`.textInputAutocapitalization(.never)` to prevent the system from capitalizing the first letter, and `.autocorrectionDisabled()` to stop the keyboard from suggesting corrections to email addresses. The phone field uses `.keyboardType(.phonePad)` to show a numeric keypad:

```
// ... existing @State properties
@State private var email = ""
@State private var phone = ""
var body: some View {
  NavigationStack {
    Form {
      // ... Personal Information section
      Section("Contact") {
        TextField("Email", text: $email)
          .keyboardType(.emailAddress)
          .textContentType(.emailAddress)
          .textInputAutocapitalization(.never)
          .autocorrectionDisabled()
        TextField("Phone", text: $phone)
          .keyboardType(.phonePad)
          .textContentType(.telephoneNumber)
      }
    }
    .navigationTitle("Register")
  }
}
```

3. Add `@State` properties for the passwords, then add a **Security** section with two `SecureField` views. `SecureField` hides the user's input behind dots, which is the standard behavior for password entry. We use a section footer to show a validation message when the passwords do not match:

```
// ... existing @State properties
@State private var password = ""
```

```
@State private var confirmPassword = ""
var body: some View {
  NavigationStack {
    Form {
      // ... Personal Information and Contact sections
      Section {
        SecureField("Password", text: $password)
          .textContentType(.newPassword)
        SecureField("Confirm Password", text: $confirmPass
          .textContentType(.newPassword)
      } header: {
        Text("Security")
      } footer: {
        if !password.isEmpty && !confirmPassword.isEmpty
            && password != confirmPassword {
          Text("Passwords do not match")
            .foregroundStyle(.red)
        }
      }
    }
    .navigationTitle("Register")
  }
}
```

4. Add a `@State` property for the bio text, then add a new `Section` with the header **About You** containing a multiline `TextField`. The `axis: .vertical` parameter tells the field to grow vertically as the user types, and `.lineLimit(3...6)` constrains it to between 3 and 6 lines:

```
// ... existing @State properties
@State private var bio = ""
var body: some View {
  NavigationStack {
    Form {
      // ... previous sections
      Section("About You") {
        TextField("Short bio", text: $bio, axis: .vertical
```

```
            .lineLimit(3...6)
        }
      }
      .navigationTitle("Register")
    }
  }
```

5. Add a new `Section` with the header **Account Preview** , using `LabeledContent` to display read-only values. `LabeledContent` aligns its label and value the same way form controls do, so the row looks consistent with the text fields and toggles above it. Add a `fullName` computed property below the `body` :

```
var body: some View {
  NavigationStack {
    Form {
      // ... previous sections
      Section("Account Preview") {
        LabeledContent("Full Name", value: fullName)
        LabeledContent("Account Type", value: "Standard")
      }
    }
    .navigationTitle("Register")
  }
}
// ... after the body
private var fullName: String {
  let name = (firstName + " " +
              lastName).trimmingCharacters(in:.whitespaces
  return name.isEmpty ? "—" : name
}
```

6. Add `@State` properties for the two toggles, then add a **Preferences** section with `Toggle` views and a submit button section. The button's action is a placeholder for now; its

`.disabled(!isFormValid)` modifier is what matters here, keeping the button inactive until the required fields are filled in and the passwords match. `isFormValid` is a computed property added below the `body` :

```
// ... existing @State properties
@State private var agreeToTerms = false
@State private var receiveNewsletter = true
var body: some View {
  NavigationStack {
    Form {
      // ... previous sections
      Section("Preferences") {
        Toggle("I agree to the Terms", isOn: $agreeToTerms
        Toggle("Receive Newsletter", isOn: $receiveNewslet
      }
      Section {
        Button("Create Account") {
          // Submit the registration
        }
        .frame(maxWidth: .infinity)
        .disabled(!isFormValid)
      }
    }
    .navigationTitle("Register")
  }
}
// ... after the body
private var isFormValid: Bool {
  !firstName.isEmpty &&
  !lastName.isEmpty &&
  !email.isEmpty &&
  !password.isEmpty &&
  password == confirmPassword &&
  agreeToTerms
}
```

With the preview still in live mode, every section is now in place, and the **Create Account** button stays dimmed until the form is valid.

7. Now, let's teach the system a bit more about our name fields. Adding `.textContentType()` here gives iOS a better chance of offering the right autofill data instead of treating both fields as generic text input. Add these modifiers to the **Personal Information** section:

```
Section("Personal Information") {
  TextField("First Name", text: $firstName)
    .textContentType(.givenName)
  TextField("Last Name", text: $lastName)
    .textContentType(.familyName)
  // ... DatePicker
}
```

8. Next, let's set up keyboard focus navigation. Define a `Field` enum with `Int` raw values inside `ContentView`, before the `@State` properties. Assigning `= 0` to the first case sets its raw value, and Swift auto-increments each case after it, so `lastName` is `1`, `email` is `2`, and so on down to `bio`. The `next` and `previous` computed properties add or subtract `1` from the current raw value to step through the cases in order, returning `nil` once they run past either end. Add a `@FocusState` property to track which field is focused:

```
struct ContentView: View {
  enum Field: Int {
    case firstName = 0
    case lastName
    case email
```

```
        case phone
        case password
        case confirmPassword
        case bio
        var next: Field? {
          Field(rawValue: rawValue + 1)
        }
        var previous: Field? {
          Field(rawValue: rawValue - 1)
        }
      }
      // ... existing @State properties
      @FocusState private var focusedField: Field?
```

9. Add `.focused()` to each field in the form. For example, in the **Personal Information** section:

```
TextField("First Name", text: $firstName)
  .textContentType(.givenName)
  .focused($focusedField, equals: .firstName)
TextField("Last Name", text: $lastName)
  .textContentType(.familyName)
  .focused($focusedField, equals: .lastName)
```

   Apply the same pattern to the email (`.email`), phone (`.phone`), password (`.password`), and confirm password (`.confirmPassword`) fields, and to the bio field (`.bio`) in the **About You** section.

10. Wrap the **Personal Information** , **Contact** , and **Security** sections in a `Group` and apply `.submitLabel(.next)` to it. This tells the keyboard to show **Next** on the return key for all fields in the group. The **About You** section stays outside the group because its multiline text field needs the standard return key for inserting line breaks:

```
Form {
  Group {
    Section("Personal Information") {
      // ... fields with .focused()
    }
    Section("Contact") {
      // ... fields with .focused()
    }
    Section {
      // ... SecureFields with .focused()
    } header: {
      // ...
    } footer: {
      // ...
    }
  }
  .submitLabel(.next)
  Section("About You") {
    // ...
  }
  // ... remaining sections
}
```

11. Add `.onSubmit` to the `Form` and two helper methods that move focus to the next or previous field. Also add a keyboard toolbar with previous, next, and dismiss buttons. The toolbar gives the user a way to navigate fields without relying solely on the return key:

```
var body: some View {
  NavigationStack {
    Form {
      // ... sections
    }
    .navigationTitle("Register")
    .onSubmit {
      moveToNextField()
    }
```

```
      .toolbar {
        ToolbarItemGroup(placement: .keyboard) {
          Button(action: moveToPreviousField) {
            Image(systemName: "chevron.up")
          }
          .disabled(focusedField == .firstName)
          Button(action: moveToNextField) {
            Image(systemName: "chevron.down")
          }
          .disabled(focusedField == .bio)
          Spacer()
          Button(action: { focusedField = nil } ) {
            Image(systemName: "keyboard.chevron.compact.dowr
          }
        }
      }
    }
}
// ... after the body
private func moveToNextField() {
  guard let current = focusedField else { return }
  focusedField = current.next
}
private func moveToPreviousField() {
  guard let current = focusedField else { return }
  focusedField = current.previous
}
```

12. Make sure the preview canvas is in live mode and interact with the view, or run the app in Device Hub. Tap through the fields, use the chevron buttons to navigate, fill in the form, and verify that the **Create Account** button enables only when validation passes. The preview should look like the following:



Figure 4.1: The finished RegistrationForm app with all sections

When navigating between fields using the return key, you may notice the keyboard briefly dismisses and reappears. In SwiftUI, pressing the return key on a `TextField` automatically resigns the first responder and fires the `.onSubmit` action. Even though our handler immediately moves focus to the next field, the resignation happens first, causing a visible keyboard



flicker. This differs from UIKit, where pressing return on a `UITextField` does nothing by default until you implement the delegate. The toolbar chevron buttons do not have this issue because they change `focusedField` directly without triggering the submit-and-resign cycle. The fix is to access the underlying UIKit text field and prevent the default dismiss behavior when the return key is pressed. Refer to the article in the *See also* section for a working sol ution.

How it works...

In our registration screen, the grouped appearance is doing a lot of work for us before we style anything manually. `Form` gives the rows, section spacing, and platform-specific treatment that make text fields, toggles, and pickers feel at home on iOS. We wrapped it in a `NavigationStack` so the title has somewhere to render, which is the same structural pattern we introduced in the *Buttons and Navigation* section of *`Chapter 1`* and will extend later in the *NavigationStack and NavigationPath* section of *`Chapter 11`*.

We organized the form into sections using `Section`, each with a string header that appears as a gray label above the group. The **Security** section used the multi-closure syntax (`header:` , `footer:`) so we could add a conditional footer that shows a red warning when the passwords do not match. This pattern of using section footers for inline validation messages is common in production forms. `SecureField` works exactly like `TextField` except that it masks the input, making it the standard choice for passwords.

The `.textContentType()` modifier deserves attention because it does something the user never sees in our code but notices immediately on the device. When we set `.textContentType(.emailAddress)` or `.textContentType(.givenName)`, the system can offer autofill suggestions pulled from the user's contacts or saved passwords. We applied `.autocorrectionDisabled()` only to the email field, because email addresses are structured data that should never be autocorrected. The name fields, by contrast, benefit from the system's awareness that they contain names (via `.textContentType(.givenName)` and `.textContentType(.familyName)`), so we left autocorrection enabled for them. This is a deliberate, per-field decision: disable autocorrection where it would interfere with structured input, and leave it on everywhere else.

We used `Group` as a transparent container to apply `.submitLabel(.next)` to the **Personal Information**, **Contact**, and **Security** sections all at once. `Group` does not add any visual structure to the form; it simply lets us attach modifiers to multiple sections without repeating them on each field. The **About You** section sits outside the group because its multiline `TextField` (created with `axis: .vertical`) uses the return key to insert line breaks, not to advance focus. If we had applied `.submitLabel(.next)` at the form level, we would have needed to override it back to `.submitLabel(.return)` on the bio field. Scoping it to a `Group` is cleaner: we apply the modifier exactly where it belongs and avoid the override.

The focus system uses a `Field` enum with `Int` raw values. The `next` and `previous` computed properties create a new `Field` from `rawValue + 1` or `rawValue - 1`, returning `nil` at the boundaries. This approach scales well: adding a new field means adding a case to the enum in

the right position, and navigation works automatically. We declared `@FocusState private var focusedField: Field?`, where `nil` means no field has focus (and the keyboard is dismissed). The `.focused($focusedField, equals: .email)` modifier binds each field to its enum case. When `focusedField` changes, SwiftUI moves the keyboard focus to the matching field. We included the bio field in the `Field` enum as the last case, so the toolbar chevrons can move into and out of it and the dismiss button can close its keyboard. It stays outside the `.submitLabel(.next)` group, so its return key still inserts line breaks rather than advancing focus; joining the focus chain and adopting that submit label are independent decisions.

The keyboard toolbar uses `ToolbarItemGroup(placement: .keyboard)` to place three buttons above the keyboard: up and down chevrons for navigating between fields, and a keyboard-dismiss button. The chevron buttons call `moveToNextField()` and `moveToPreviousField()`, which read the current case and step to the adjacent one. Each chevron is disabled at the corresponding boundary (`.firstName` for up, `.bio` for down). The `.onSubmit` modifier on the form calls `moveToNextField()` as a secondary navigation path when the user taps the return key. Finally, `LabeledContent` provides a read-only row that aligns its label and value like any other form control, which we used for the **Account Preview** section.



In a production app, you would typically extract the form's state properties and validation logic into a separate data model using the `@Observable` macro from the Observation framework. This keeps the view focused on layout and lets you test business rules

> independently. We will cover the Observation framework in detail in the *Observation Framework* section of *C hapter 5* .

See also

- `Form` documentation: https://x.swiftuicookbook.com/y0u12
- `TextField` documentation: https://x.swiftuicookbook.com/2qgyk
- `SecureField` documentation: https://x.swiftuicookbook.com/7fuaf
- `FocusState` documentation: https://x.swiftuicookbook.com/nbtw1
- `LabeledContent` documentation: https://x.swiftuicookbook.com/5z5bw
- `DatePicker` documentation: https://x.swiftuicookbook.com/nmfpo
- *SwiftUI TextField: Handling Keyboard the Right Way* article: https://x.swiftuicookbook. com/o9w9q

Alerts and Confirmation Dialogs

Delete confirmations, rename prompts, and multi-option action menus: these are the moments when our app pauses and lets the user make a choice. Getting these wrong, whether by skipping a confirmation before a destructive action or by forcing a full-screen

modal for a simple yes/no question, erodes the trust between the app and the person using it.

In this recipe, we will build a single-screen app that presents several modal views: a basic informational `.alert()`, a destructive confirmation with `Button(role: .destructive)`, a multi-option `.confirmationDialog()` whose actions appear in a menu anchored to the button that opens it, and an alert with an embedded `TextField` for inline renaming. We then move beyond toggling a `Bool` to the item-driven and error-driven forms, where an optional value both triggers the alert or dialog and carries the data it shows. A status message reflects the result of each action, so we can see the effect without needing a d ata model.

Getting ready

Create a new SwiftUI iOS app named **Aler tShowcase** .

How to do it...

We will build a single screen with buttons that trigger different alerts and dialogs. Each one demonstrates a different pattern. These are the steps:

1. Open `ContentView.swift` and replace the `ContentView` struct with the following. It declares `@State` properties for controlling which alert is shown, the item name, and a status message, and its `body` is a `NavigationStack` containing the item name, the status message, and a **Show Welcome Alert** button that triggers a basic alert:

```
struct ContentView: View {
  @State private var showWelcome = false
  @State private var itemName = "My Document"
  @State private var message = "Tap a button to see an ale
  var body: some View {
    NavigationStack {
      VStack(spacing: 32) {
        Text(itemName)
          .font(.title)
          .fontWeight(.bold)
        Text(message)
          .font(.headline)
          .foregroundStyle(.secondary)
          .multilineTextAlignment(.center)
          .padding(.horizontal)
        VStack(spacing: 16) {
          Button("Show Welcome Alert") {
            showWelcome = true
          }
          .buttonStyle(.bordered)
        }
        Spacer()
      }
      .padding(.top, 48)
      .padding(.horizontal, 8)
      .navigationTitle("Alerts & Dialogs")
      .alert("Welcome!", isPresented: $showWelcome) {
        Button("OK") {
          message = "You dismissed the welcome alert."
        }
      } message: {
        Text("This is a simple informational alert with a
      }
    }
  }
}
```

In the preview canvas, tap **Show Welcome Alert** . An alert appears with a title, a message, and a single **OK** button.

Tapping **OK** dismisses the alert and updates the status message.

2. Add a `@State` property for the delete alert, then add a **Delete Item** button below the welcome button. This alert uses `Button(role: .destructive)` for the delete action and `Button(role: .cancel)` for the cancel action. The system styles destructive buttons in red and positions the cancel button automatically:

```
@State private var showWelcome = false
@State private var showDelete = false
@State private var itemName = "My Document"
@State private var message = "Tap a button to see an alert
var body: some View {
  NavigationStack {
    VStack(spacing: 32) {
      // ... item name and status message
      VStack(spacing: 16) {
        // ... Show Welcome Alert button
        Button("Delete Item", role: .destructive) {
          showDelete = true
        }
        .buttonStyle(.bordered)
      }
      // ... Spacer
    }
    // ... padding, navigationTitle, and .alert("Welcome!"
    .alert("Delete Item?", isPresented: $showDelete) {
      Button("Delete", role: .destructive) {
        message = "Item deleted."
      }
      Button("Cancel", role: .cancel) { }
    } message: {
      Text("This action cannot be undone. " +
           "The item will be permanently removed.")
    }
  }
}
```

3. Add a `@State` property for the options dialog, then add a **Show Options** button. This time we use `.confirmationDialog()` instead of `.alert()`, and we attach it directly to the button rather than to the surrounding container. The action buttons appear in a menu anchored to the **Show Options** button. We set `titleVisibility: .visible` so the title is shown:



With the Liquid Glass design, a confirmation dialog animates out from the control that presents it, so it needs to know which element opened it. Attaching `.confirmationDialog()` to the triggering button gives the menu a clear anchor to morph from; attaching it to a parent container can leave the menu with an inconsistent anchor.

```
@State private var showWelcome = false
@State private var showDelete = false
@State private var showOptions = false
@State private var itemName = "My Document"
@State private var message = "Tap a button to see an alert or c
var body: some View {
  NavigationStack {
    VStack(spacing: 32) {
      // ... item name and status message
      VStack(spacing: 16) {
        // ... existing buttons
        Button("Show Options") {
          showOptions = true
        }
        .buttonStyle(.borderedProminent)
        .confirmationDialog("What would you like to do?",
                            isPresented: $showOptions,
                            titleVisibility: .visible) {
          Button("Duplicate") {
```

```
            message = "Item duplicated."
          }
          Button("Share") {
            message = "Item shared."
          }
          Button("Archive") {
            message = "Item archived."
          }
          Button("Delete", role: .destructive) {
            message = "Item deleted."
          }
        } message: {
          Text("Choose an action for \"\(itemName)\".")
        }
      }
      // ... Spacer
    }
    // ... padding, navigationTitle, and existing alerts
  }
}
```

4. Add `@State` properties for the rename alert and the text field, then add a **Rename Item** button. This alert includes a `TextField` inside the actions closure, which lets the user type a new name directly in the alert:

```
@State private var showWelcome = false
@State private var showDelete = false
@State private var showOptions = false
@State private var showRename = false
@State private var itemName = "My Document"
@State private var renameText = ""
@State private var message = "Tap a button to see an alert
var body: some View {
  NavigationStack {
    VStack(spacing: 32) {
      // ... item name and status message
      VStack(spacing: 16) {
        // ... existing buttons
```

```
            Button("Rename Item") {
              renameText = itemName
              showRename = true
            }
            .buttonStyle(.bordered)
          }
          // ... Spacer
        }
        // ... padding, navigationTitle, and existing .alert()
        .alert("Rename Item", isPresented: $showRename) {
          TextField("New name", text: $renameText)
          Button("Rename") {
            itemName = renameText
            message = "Item renamed to \"\(renameText)\"."
          }
          Button("Cancel", role: .cancel) {
            renameText = ""
          }
        } message: {
          Text("Enter a new name for this item.")
        }
      }
    }
```

Make sure the preview canvas is in live mode; the **Delete Item** , **Show Options** , and **Rename Item** buttons now each present their alert or dialog, and the status message updates after every choice.



When the rename alert opens in Device Hub or on a device, the software keyboard lifts the content behind the alert, so the item name can ride up under the navigation title. It is only a background detail while a modal is up, and it does not affect the alert itself. If it bothers you in

a real app, put the screen's content in a `ScrollView` so keyboard avoidance scrolls it instead of pushing it.

5. Now, we add the same delete confirmation in its item-driven form. Add a property for the name of the item being deleted, a **Delete (item-driven)** button that sets it, and an `.alert()` bound to that optional value. Because the item-driven overload has no `Identifiable` requirement, the property is a plain `String?` ; no wrapper type is needed:



New in iOS 27. Alerts and confirmation dialogs gain item-driven and error-driven presentation, replacing a separate `Bool` with the value the modal needs. We bind to an optional value, or to an `Error` . The bound item has no `Identifiable` requirement, unlike `sheet(item:)` . The alert or dialog appears when that value is not `nil` , and the action and message closures receive the unwrapped item or error.

```
// ... existing @State properties
@State private var itemToDelete: String?
var body: some View {
  NavigationStack {
    VStack(spacing: 32) {
      // ... item name and status message
      VStack(spacing: 16) {
        // ... existing buttons
        Button("Delete (item-driven)") {
          itemToDelete = itemName
```

```
          }
          .buttonStyle(.bordered)
        }
        // ... Spacer
      }
      // ... padding, navigationTitle, and existing alert modifie
      .alert("Delete Item?", item: $itemToDelete) { name in
        Button("Delete \(name)", role: .destructive) {
          message = "Deleted \(name)."
        }
        Button("Cancel", role: .cancel) { }
      } message: { name in
        Text("This permanently removes \(name).")
      }
    }
  }
```

6. The same item binding works for `.confirmationDialog()`. Add another `String?` property for the item being archived and an **Archive (item dialog)** button, with an item-driven confirmation dialog attached directly to it (same reason as before, so the menu anchors to that button):

```
// ... existing @State properties
@State private var itemToArchive: String?
var body: some View {
  NavigationStack {
    VStack(spacing: 32) {
      // ... item name and status message
      VStack(spacing: 16) {
        // ... existing buttons
        Button("Archive (item dialog)") {
          itemToArchive = itemName
        }
        .buttonStyle(.bordered)
        .confirmationDialog("Archive?", item: $itemToArchi
                            titleVisibility: .visible) { r
          Button("Archive \(name)") {
```

```
            message = "Archived \(name)."
          }
        } message: { name in
          Text("Move \(name) to the archive?")
        }
      }
      // ... Spacer
    }
    // ... padding, navigationTitle, and existing alert mo
  }
}
```

7. Errors get their own presentation form. Define a `LoadError` type that conforms to `LocalizedError`, add a property for an optional error, a **Simulate Load Error** button, and an error-driven alert that reads its text from the bound error:

```
struct LoadError: LocalizedError {
  let message: String
  var errorDescription: String? { message }
}
struct ContentView: View {
  // ... existing @State properties
  @State private var loadError: LoadError?
  var body: some View {
    NavigationStack {
      VStack(spacing: 32) {
        // ... item name and status message
        VStack(spacing: 16) {
          // ... existing buttons
          Button("Simulate Load Error") {
            loadError = LoadError(message: "The document f
          }
          .buttonStyle(.bordered)
        }
        // ... Spacer
      }
      // ... padding, navigationTitle, and existing alert
      .alert(error: $loadError) { error in
```

```
            Button("OK") {
              message = "Dismissed error."
            }
          } message: { error in
            Text(error.message)
          }
        }
      }
    }
```

8. Use the preview canvas in live mode to test each interaction. Tap every button, observe how each alert or dialog appears, and check that the status message updates after each action. The preview should look like the following:

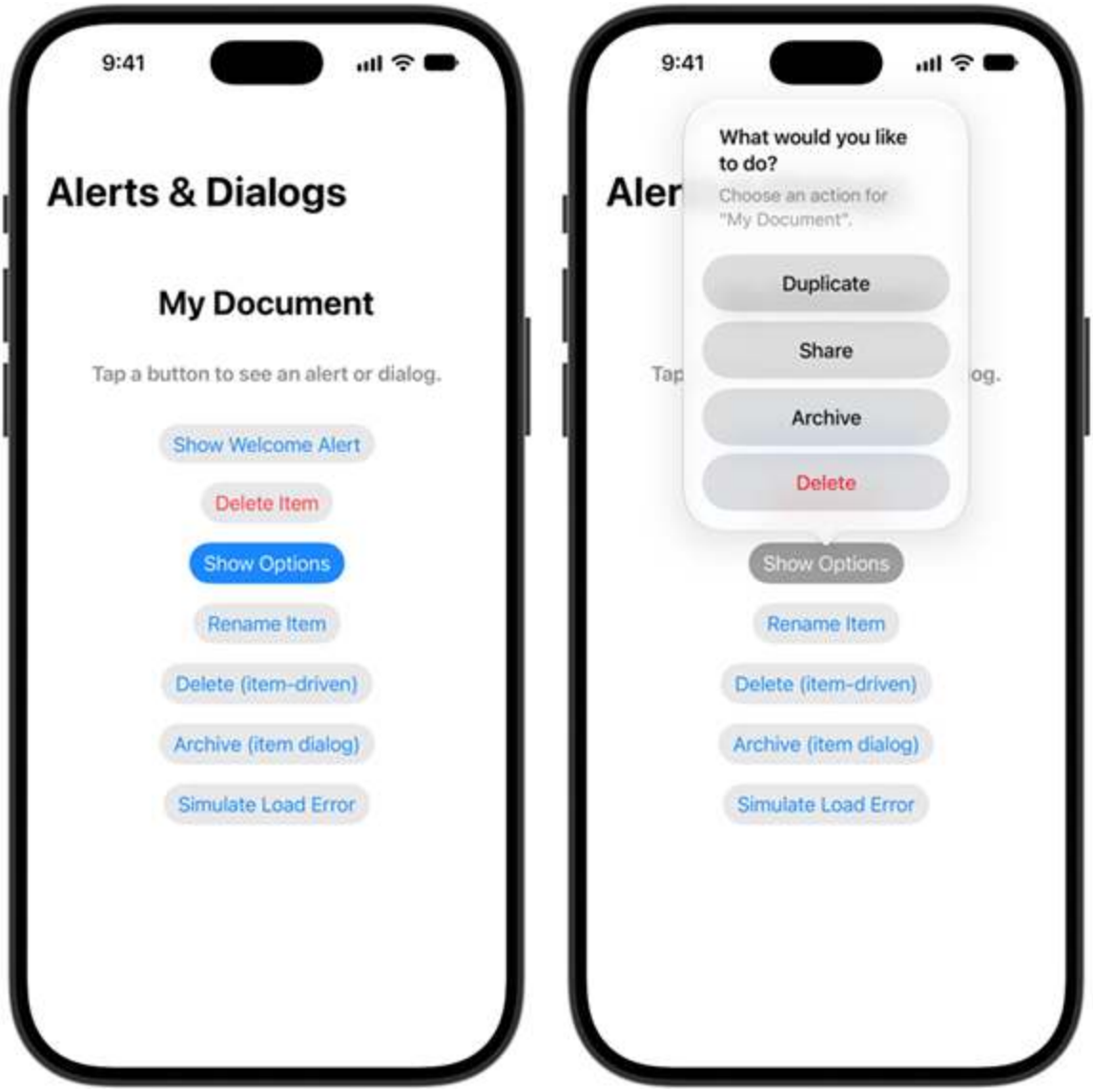


Figure 4.2: The AlertShowcase app showing the main screen and a con firmation dialog

How it works...

In this sample, the Boolean-driven alerts and dialog each flip a `Bool`, and that is enough for SwiftUI to decide which one to present; we move to the item-driven and error-driven forms, which bind to an optional value instead, later in this section. The `.alert()` modifier combines three pieces: the title, a binding that controls visibility, and

closures for the buttons and message. When the user taps one of the alert actions, SwiftUI resets the binding to `false` and dismisses the alert. We attached three separate `.alert()` modifiers to the same view, each driven by its own `@State` property (we will cover `@State` and `@Binding` in detail in the *State and Binding Basics* section of *Chapter 5*). SwiftUI only shows one at a time, based on which Boolean is `true`.

The delete alert and the options dialog show why roles matter more than button titles alone. When we mark the delete action as `.destructive`, the system gives it the visual weight that warns the user before they commit. When we mark another action as `.cancel`, SwiftUI places and styles it according to the platform's own conventions. The confirmation dialog uses the same idea on a broader set of actions, but its presentation is genuinely new. Liquid Glass, introduced in iOS 26, changed how these dialogs appear. Traditionally, a confirmation dialog (an action sheet) slid up from the bottom of the screen with its action buttons stacked along the bottom, regardless of which control the user tapped. With Liquid Glass, the dialog instead animates out from the control that presents it and appears as a menu anchored to that control. That is why we attach `.confirmationDialog()` directly to the **Show Options** button rather than to the surrounding container: the menu needs the triggering element to anchor to and morph from. Setting `titleVisibility: .visible` shows the dialog's title, **What would you like to do?**; a confirmation dialog hides its title by default (`.automatic`), so we make it visible here to label the menu of actions.

The rename alert demonstrates that `.alert()` can include interactive views like `TextField` inside the actions closure. We pre-fill `renameText`

with the current item name before showing the alert, so the user sees the existing name and can edit it. This pattern is useful for quick inline edits without navigating to a separate screen.

The last three modals move from `Bool`-driven to item-driven and error-driven presentation. Instead of a separate `Bool`, we bind to an optional value (`itemToDelete` and `itemToArchive` are plain `String?` here) or to an optional `LoadError?`: the modal appears when the value is not `nil`, SwiftUI hands the unwrapped value to the action and message closures, and it resets the binding to `nil` on dismissal. Unlike `.sheet(item:)`, the dialog and alert `item:` overloads put no `Identifiable` requirement on that value, so a bare `String` is enough when the name is all the modal needs; in a larger app you would usually bind a model value instead, which still needs no `Identifiable`. Because the closures receive the value, the message can name the specific item directly rather than reading from a separate property. The error-driven `.alert(error:)` applies the idea to failures: we hold an optional `LocalizedError`, and the alert reads its text from the bound error. Keeping the data and the presentation together this way removes the risk of a `Bool` and its data drif ting out of sync.

See also

- `alert(_:isPresented:actions:message:)` documentation: https://x.swiftuicookbook.com/u3f5y
- `confirmationDialog(_:isPresented:titleVisibility:actions:message:)` documentation: https://x.swiftuicookbook.com/qszcu
- `ButtonRole` documentation: https://x.swiftuicookbook.com/wrnno

- `alert(_:item:actions:message:)` documentation: https://x.swiftuicookbook.com/tyzbo
- `confirmationDialog(_:item:titleVisibility:actions:message:)` documentation: https://x.swiftuicookbook.com/ys286
- `alert(error:actions:message:)` documentation: https://x.swiftuicookbook.com/4x6ze
- Adopting Liquid Glass: https://x.swiftuicookbook.com/agbz4

Menus and Disclosure Groups

As an app's feature set grows, we need ways to organize actions without cluttering the main screen. The tradeoff is always between discoverability and clutter: put every action on the surface and the screen becomes noisy, hide them too deeply and the user never finds them.

In this recipe, we build a single screen that combines four patterns for tucking actions away while keeping them easy to reach: a `.contextMenu()` with a `ControlGroup` for compact quick actions and a nested `Menu` for color changes, a `DisclosureGroup` for collapsible notification settings, and a toolbar `Menu` with `.menuActionDismissBehavior(.disabled)` so the user can toggle multiple options without the menu closi ng after each tap.

Getting ready

Create a new SwiftUI iOS app n amed **MenuExplorer** .

How to do it...

We will start with the context menu on a central element, then add a disclosure group for settings, and finish with a toolbar menu. These are the steps:

1. Open `ContentView.swift` and replace the `ContentView` struct with the following. It declares `@State` properties for the star color and a status message, and its `body` is a `NavigationStack` containing a large star icon with a `.contextMenu()`. Inside the context menu, the `ControlGroup` renders its buttons as a compact horizontal strip at the top, which is the standard iOS pattern for quick actions. Below that, a nested `Menu` groups the color choices under a second level, and a destructive reset button stays at the bottom in red:

```
struct ContentView: View {
  @State private var starColor = Color.yellow
  @State private var message = "Long-press the star or " +
                               "use the toolbar menu."
  var body: some View {
    NavigationStack {
      VStack(spacing: 32) {
        Image(systemName: "star.fill")
          .font(.system(size: 80))
          .foregroundStyle(starColor)
          .contextMenu {
            ControlGroup {
              Button("Copy", systemImage: "doc.on.doc") {
                message = "Copied."
              }
              Button("Share", systemImage: "square.and.arr
                message = "Shared."
              }
              Button("Save", systemImage: "square.and.arro
                message = "Saved."
```

```
            }
          }
          Divider()
          Menu("Change Color") {
            Button("Yellow") {
              starColor = .yellow
              message = "Color changed to yellow."
            }
            Button("Blue") {
              starColor = .blue
              message = "Color changed to blue."
            }
            Button("Green") {
              starColor = .green
              message = "Color changed to green."
            }
            Button("Red") {
              starColor = .red
              message = "Color changed to red."
            }
            Button("Orange") {
              starColor = .orange
              message = "Color changed to orange."
            }
          }
          Divider()
          Button("Reset", role: .destructive) {
            starColor = .yellow
            message = "Color reset to yellow."
          }
        }
        Text(message)
          .font(.headline)
          .foregroundStyle(.secondary)
          .multilineTextAlignment(.center)
          .padding(.horizontal)
        Spacer()
      }
      .padding(.top, 48)
      .navigationTitle("Menu Explorer")
    }
  }
}
```

In the preview canvas, long-press the star. The context menu appears with a compact row of action buttons at the top (the `ControlGroup`), a **Change Color** submenu, and a red **Reset** button at the bottom.

2. Add `@State` properties for the notification toggles, then add a `DisclosureGroup` after the status message and before the `Spacer`. We wrap the toggles in a `Group` and apply `.padding(.trailing, 2)` to prevent them from being clipped by the disclosure group's layout:

```
@State private var starColor = Color.yellow
@State private var notificationsEnabled = true
@State private var soundEnabled = true
@State private var hapticEnabled = false
@State private var message = "Long-press the star or " +
                             "use the toolbar menu."
var body: some View {
  NavigationStack {
    VStack(spacing: 32) {
      // ... star and message
      DisclosureGroup("Notification Preferences") {
        Group {
          Toggle("Notifications", isOn: $notificationsEnab
          Toggle("Sound", isOn: $soundEnabled)
          Toggle("Haptic Feedback", isOn: $hapticEnabled)
        }
        .padding(.trailing, 2)
      }
      .padding(.horizontal)
      Spacer()
    }
    .padding(.top, 48)
    .navigationTitle("Menu Explorer")
  }
}
```

Tap the **Notification Preferences** header to expand and collapse the toggles.

3. Add `@State` properties for the view options, then add a `Menu` inside a toolbar right after the `.navigationTitle()`. The `.menuActionDismissBehavior(.disabled)` modifier keeps the menu open after each toggle, so the user can change multiple options without the menu closing:

```
@State private var starColor = Color.yellow
@State private var showGrid = false
@State private var showRulers = false
@State private var snapToGrid = false
// ... the three notification properties and message
var body: some View {
  NavigationStack {
    VStack(spacing: 32) {
      // ... star, message, and disclosure group
    }
    .padding(.top, 48)
    .navigationTitle("Menu Explorer")
    .toolbar {
      ToolbarItem(placement: .primaryAction) {
        Menu {
          Toggle("Show Grid", isOn: $showGrid)
          Toggle("Show Rulers", isOn: $showRulers)
          Toggle("Snap to Grid", isOn: $snapToGrid)
        } label: {
          Label("View Options", systemImage: "slider.horiz
        }
        .menuActionDismissBehavior(.disabled)
      }
    }
  }
}
```

Tap the toolbar icon to open the menu. Toggle any option and notice the menu stays open. Without `.menuActionDismissBehavior(.disabled)`, the menu would close after each tap.

4. With the preview canvas in live mode, explore all three interaction patterns: the context menu on the star, the disclosure group for notification settings, and the toolbar menu for view options. The preview should look like the following:

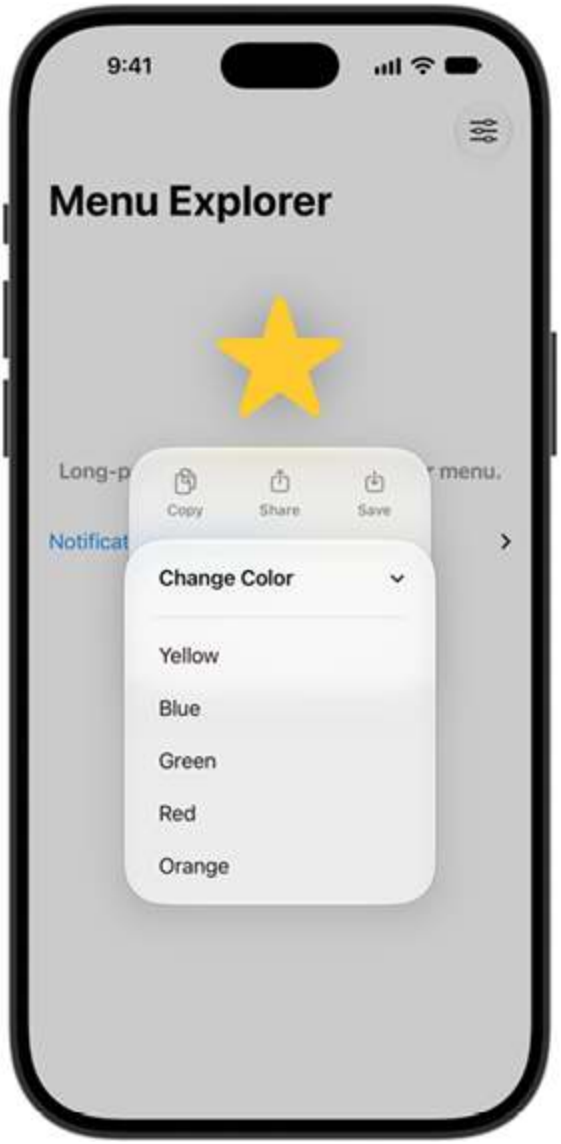


Figure 4.3: The MenuExplorer app showing the context menu with a ControlGro up and color submenu

How it works...

When we long-pressed the star, the actions appeared right next to the thing they belong to. That is the real strength of `.contextMenu()`: it keeps secondary actions attached to a specific view instead of interrupting the whole screen with a modal presentation, the way `.confirmationDialog()` does (see the *Alerts and Confirmation Dialogs* section). Inside that context menu, we combined three building blocks. `ControlGroup` gives us the compact row of quick actions at the top, `Menu("Change Color")` nests the color choices under a second level, and `Divider()` separates those chunks visually. The destructive reset action stays at the bottom in red, which makes the menu easier to scan.

The **Notification Preferences** section in our app collapses and expands with a single tap, thanks to `DisclosureGroup`. The system manages the expanded/collapsed state internally when no binding is provided. We wrapped the toggles in a `Group` and applied `.padding(.trailing, 2)` because the toggle icons can be clipped by the disclosure group's default layout. It is a subtle adjustment, but it keeps the toggles fully visible.

On the toolbar, the view-options control looks like a single icon, but it carries a full menu behind it. We used a `Label` so the toolbar can show the symbol while still keeping the text for accessibility. Inside the menu, the three `Toggle` rows become much more usable once `.menuActionDismissBehavior(.disabled)` is in place, because the user can

flip several options in one pass instead of reopening the menu after every tap. This is the kind of small behavior tweak that makes a settings menu feel deliberate rather than annoying.

See also

- `Menu` documentation: https://x.swiftuicookbook.com/00475
- `DisclosureGroup` documentation: https://x.swiftuicookbook.com/f0e7a
- `ControlGroup` documentation: https://x.swiftuicookbook.com/dcpnd
- `contextMenu` documentation: https://x.swiftuicookbook.com/jbgh1

Toolbars and the Liquid Glass Design

Toolbars matter most when a screen has several action buttons competing for limited space. iOS 26 changes how toolbars look and behave, with Liquid Glass surfaces that group related items visually, so this is a good time to revisit how we configure them.

In this recipe, we build a document browser and editor so we can see how toolbar choices change between a list screen and a focused editing screen. The list uses a `ToolbarTitleMenu` for sorting options, while the editor uses `ToolbarItemGroup` in the bottom bar for font controls and `.toolbarRole(.editor)` for the standard editor chrome. We give the editor enough actions to show the toolbar adapting to tight

space: it keeps the important buttons visible, moves the rest into an overflow menu, pins one button to the trailing edge, and minimizes the bar as the editor scrolls. That setup gives us a practical way to explore item grouping, placement, the glass styling, and the adaptivity controls without turning the recipe into a tour of isolated APIs.

Getting ready

Create a new SwiftUI iOS app named **ToolbarShowcase** .

How to do it...

We will build a list of documents with a title menu for sorting, then a detail editor with font controls in the bottom bar and secondary actions in the overflow menu. These are the steps:

1. Open `ContentView.swift` . Start by defining a `Note` model with just enough information to make the sample behave like a real document list: an identifier for navigation, text fields for the title and content, a font size the detail screen can adjust, a creation date for sorting, and a few Boolean flags the editor can toggle (favorite, pin, lock, flag, and tag). Then, replace the `ContentView` declaration with a `NavigationStack` , a `List` of notes, and navigation modifiers. We use value-based `NavigationLink` with `UUID` so each note navigates to a unique destination. For now, the destination is a `Text` placeholder showing the note's title; we replace it with the note editor in step 3:

```
struct Note: Identifiable {
  let id = UUID()
  var title: String
  var content: String = ""
  var fontSize: CGFloat = 16
  var dateCreated: Date = .now
  var isFavorite = false
  var isPinned = false
  var isLocked = false
  var isFlagged = false
  var isTagged = false
}
struct ContentView: View {
  @State private var notes = [
    Note(title: "Meeting Notes",
         dateCreated: .now.addingTimeInterval(-86400 * 3))
    Note(title: "Project Plan",
         dateCreated: .now.addingTimeInterval(-86400 * 4))
    Note(title: "Travel Itinerary",
         dateCreated: .now.addingTimeInterval(-86400 * 2))
    Note(title: "Shopping List",
         dateCreated: .now.addingTimeInterval(-86400)),
    Note(title: "Recipe Collection")
  ]
  var body: some View {
    NavigationStack {
      List(notes) { note in
        NavigationLink(value: note.id) {
          Text(note.title)
        }
      }
      .navigationTitle("Documents")
      .navigationBarTitleDisplayMode(.inline)
      .navigationDestination(for: UUID.self) { id in
        if let index = notes.firstIndex(where: { $0.id ==
          Text(notes[index].title)
            .navigationTitle(notes[index].title)
        }
      }
    }
  }
}
```



Value-based navigation with `NavigationLink(value:)` and `.navigationDestination(for:)` is covered in depth in the *NavigationStack and NavigationPath* section of *Chapter 11*.

2. Add a `.toolbar` to the list with a `ToolbarTitleMenu` for sorting options and a `ToolbarItem` for adding new notes. `ToolbarTitleMenu` attaches a menu to the navigation title, so the user taps the title to see the different sorting options. We use `.navigationBarTitleDisplayMode(.inline)` because `ToolbarTitleMenu` requires an inline title:

```
var body: some View {
  NavigationStack {
    List(notes) { note in
      // ... note row
    }
    // ... navigationTitle and navigationDestination
    .toolbar {
      ToolbarTitleMenu {
        Button("Sort by Name", systemImage: "textformat.ab
          notes.sort { $0.title < $1.title }
        }
        Button("Sort by Date", systemImage: "calendar") {
          notes.sort { $0.dateCreated < $1.dateCreated }
        }
        Divider()
        Button("Reverse Order", systemImage: "arrow.up.arr
          notes.reverse()
        }
      }
      ToolbarItem(placement: .primaryAction) {
```

```
          Button("New", systemImage: "plus") {
            let count = notes.count + 1
            notes.append(Note(title: "Untitled \(count)"))
          }
        }
      }
    }
  }
```

In the preview canvas, tap the **Documents** title. A menu appears with sorting options. Tap **Sort by Name** to reorder the list alphabetically.

3. Create a `DetailView` that takes a `@Binding` to a `Note` and reads the `dismiss` action from the environment, which a toolbar button will use later to close the editor. Add a `TextEditor` for editing the note content, with the font size driven by the note's `fontSize` property. Apply `.toolbarRole(.editor)`, which tells SwiftUI that this screen is an editor and lets the system adjust the toolbar appearance accordingly:

```
struct DetailView: View {
  @Binding var note: Note
  @Environment(\.dismiss) private var dismiss
  var body: some View {
    TextEditor(text: $note.content)
      .font(.system(size: note.fontSize))
      .padding()
      .navigationTitle(note.title)
      .navigationBarTitleDisplayMode(.inline)
      .toolbarRole(.editor)
  }
}
```

Then, in `ContentView`, replace the `Text` placeholder in the `.navigationDestination()` closure, along with its `.navigationTitle()` modifier, with the new `DetailView`. We pass a binding to the selected note so the editor changes the note in place, and `DetailView` sets its own title:

```
.navigationDestination(for: UUID.self) { id in
  if let index = notes.firstIndex(where: { $0.id == id }) {
    DetailView(note: $notes[index])
  }
}
```

4. Add the detail view's `.toolbar`, starting with the primary actions: a `ShareLink` for sharing the note and a row of toggle buttons for marking it (favorite, pin, lock, flag, and tag). We give the editor this many actions on purpose, because the toolbar's adaptive behavior only shows up once the buttons no longer fit in the horizontal space available to it. Attach `.visibilityPriority()` to each item to control what survives when space runs short: `.high` items stay visible, while `.low` items are the first to collapse into the overflow. Here `ShareLink` and `Flag` are `.high` and the others are `.low`:



New in iOS 27. Our editor can now decide which toolbar buttons stay visible as space shrinks. We use visibility priorities to protect the `ShareLink` and `Flag` buttons, move the counts into `ToolbarOverflowMenu`, pin the checkmark button to the trailing edge, and minimize the navigation bar as the document scrolls.

```
var body: some View {
  TextEditor(text: $note.content)
    // ... font, padding, navigationTitle, and toolbarRole
    .toolbar {
      ToolbarItem(placement: .primaryAction) {
        ShareLink(item: note.content)
      }
      .visibilityPriority(.high)
      ToolbarItem(placement: .primaryAction) {
        Button("Favorite",
               systemImage: note.isFavorite ? "star.fill" : "st
          note.isFavorite.toggle()
        }
      }
      .visibilityPriority(.low)
      ToolbarItem(placement: .primaryAction) {
        Button("Pin",
               systemImage: note.isPinned ? "pin.fill" : "pin")
          note.isPinned.toggle()
        }
      }
      .visibilityPriority(.low)
      ToolbarItem(placement: .primaryAction) {
        Button("Lock",
               systemImage: note.isLocked ? "lock.fill" : "lock
          note.isLocked.toggle()
        }
      }
      .visibilityPriority(.low)
      ToolbarItem(placement: .primaryAction) {
        Button("Flag",
               systemImage: note.isFlagged ? "flag.fill" : "fla
          note.isFlagged.toggle()
        }
      }
      .visibilityPriority(.high)
      ToolbarItem(placement: .primaryAction) {
        Button("Tag",
               systemImage: note.isTagged ? "tag.fill" : "tag")
          note.isTagged.toggle()

        }
```

```
            }
        }
        .visibilityPriority(.low)
    }
}
```

5. Create the word and character count items inside a `ToolbarOverflowMenu`, and add a checkmark button with the `.topBarPinnedTrailing` placement. Items inside a `ToolbarOverflowMenu` always live in the overflow menu, regardless of available space, which suits secondary information like these counts. The pinned checkmark button stays anchored at the trailing edge even as the other items reflow, and it calls `dismiss()` to close the editor:

```
var body: some View {
  TextEditor(text: $note.content)
    // ... font, padding, navigationTitle, and toolbarRole
    .toolbar {
      // ... primary action buttons
      ToolbarOverflowMenu {
        Button("Words: \(note.content.split(separator: " "
               systemImage: "text.word.spacing") { }
        Button("Characters: \(note.content.count)",
               systemImage: "character.cursor.ibeam") { }
      }
      ToolbarItem(placement: .topBarPinnedTrailing) {
        Button("Done", systemImage: "checkmark") { dismiss
      }
    }
}
```

6. Add two font size controls to the bottom bar with a `ToolbarItemGroup` that places them on a shared Liquid Glass

surface, then a `ToolbarSpacer(.flexible)` to push the group to the leading edge. After the `.toolbar`, keep the glass background visible with `.toolbarBackgroundVisibility(.visible, for: .bottomBar)`, and let the navigation bar shrink as the user scrolls with `.toolbarMinimizationBehavior(.onScrollDown)`:

```
var body: some View {
  TextEditor(text: $note.content)
    // ... font, padding, navigationTitle, and toolbarRole
    .toolbar {
      // ... primary action buttons, overflow menu and pin
      ToolbarItemGroup(placement: .bottomBar) {
        Button("Smaller", systemImage: "textformat.size.sm
          if note.fontSize > 12 { note.fontSize -= 2 }
        }
        Button("Larger", systemImage: "textformat.size.lar
          if note.fontSize < 28 { note.fontSize += 2 }
        }
      }
      ToolbarSpacer(.flexible, placement: .bottomBar)
    }
    .toolbarBackgroundVisibility(.visible, for: .bottomBar
    .toolbarMinimizationBehavior(.onScrollDown)
}
```

The font size and the marking flags are stored in the `Note` model, so each note remembers its size and its favorite, pin, lock, flag, and tag states when the user navigates back and returns later.

7. Add a second preview so you can iterate on the detail editor without opening it from the list each time. `@Previewable @State` gives the preview a mutable `Note` to bind to:

```
#Preview("Detail View") {
  @Previewable @State var note = Note(title: "Meeting Note
    dateCreated: .now.addingTimeInterval(-86400 * 3))
  NavigationStack {
    DetailView(note: $note)
  }
}
```

Make sure the preview canvas is in live mode and select the **Detail View** preview; the editor opens on its own so you can type, toggle the marking buttons, and adjust the font size without navigating from the list.

8. Back in the preview canvas, explore both screens. On the list screen, tap the title to try the sorting options. On the detail screen, type some text, toggle the marking buttons, and adjust the font size. To watch the toolbar adapt, vary the width available to the editor: in portrait the lower-priority buttons collapse into the overflow, while a wider layout such as landscape fits more of them. Scroll the editor to see the navigation bar minimize. The preview should look like the following:



Figure 4.4: The ToolbarShowcase app showing the document list and the detail editor with toolbar controls



You don't have to run the app in Device Hub to see this, and previews are usually faster. The preview canvas can show portrait and landscape orientation variants, so you can interact with the landscape one directly. Xcode 27 also adds a Resizable mode, which shows the preview in an arbitrarily sized screen: drag

> the canvas wider and narrower and watch the buttons flow in and out of the overflow in re al time as the width changes.

How it works…

On the documents screen, the title itself becomes part of the sorting UI. Tapping it opens a title menu that keeps the list-level actions close to the content they affect, while the plus button stays available as a separate toolbar item with `.primaryAction` placement, which positions it in the trailing area of the navigation bar. That split is a good fit for this kind of screen: sorting changes how the whole list is organized, while adding a note is a primary action. One detail to watch: the title menu only appears when the title display mode is `.inline`; a large title does not support this interaction.

The detail screen is where the iOS 27 toolbar adaptivity comes into play. `.toolbarRole(.editor)` changes how the navigation bar renders: it centers the title and turns the back button into a compact chevron, the standard look for editor screens. The trailing area now holds more actions than a phone can show at once, so SwiftUI collapses the extras into an overflow menu, and `.visibilityPriority()` decides which ones go. The `.high` items, the share button and the flag toggle, stay visible, while the `.low` toggles are the first to move into the overflow. Because priority wins over position, a later button like `Flag` can outrank earlier ones such as `Favorite`. The word and character counts sit in a `ToolbarOverflowMenu`, which keeps them in the overflow regardless of available space, and the checkmark button uses the

`.topBarPinnedTrailing` placement to hold its spot at the trailing edge no matter how the other items reflow.

The bottom bar uses `ToolbarItemGroup` to place the font size buttons on a shared Liquid Glass surface; when several items share a group, iOS 26 renders them on the same glass background to signal that they belong together. `ToolbarSpacer(.flexible)` pushes the group to the leading edge, much like `Spacer()` in an `HStack`, and `.toolbarBackgroundVisibility(.visible, for: .bottomBar)` keeps the glass background visible even when the bar would otherwise be transparent. Finally, `.toolbarMinimizationBehavior(.onScrollDown)` shrinks the navigation bar as the user scrolls the editor, giving the text more room and expanding the bar again when they scroll back up.

The font size and the marking flags live directly in the `Note` model rather than as local state in the detail view. The binding (`$notes[index]`) writes changes back to the source array in `ContentView`, so each note keeps its own font size and its favorite, pin, lock, flag, and tag states when the u ser navigates away and returns.

See also

- `toolbar(content:)` documentation: https://x.swiftuicookbook.com/j6eqa
- `ToolbarItem` documentation: https://x.swiftuicookbook.com/vzmv5
- `ToolbarItemGroup` documentation: https://x.swiftuicookbook.com/6g0db

- `ToolbarTitleMenu` documentation: https://x.swiftuicookbook.com/q65mg
- `visibilityPriority(_:)` documentation: https://x.swiftuicookbook.com/qyrom
- `ToolbarOverflowMenu` documentation: https://x.swiftuicookbook.com/meq1j
- `toolbarMinimizationBehavior(_:for:)` documentation: https://x.swiftuicookbook.com/3zlab
- Build a SwiftUI app with the new design (WWDC 2025): https://x.swiftuicookbook.com/wliha

Reusable Views and Modifiers

As our SwiftUI views grow, the `body` property tends to get long and hard to follow. At that point, the question is no longer whether the screen works, but whether we can still see its structure quickly enough to keep changing it with confidence. In this recipe, we will take a profile dashboard and break it into the kinds of reusable pieces that make a screen easier to read, easier to extend, and much less fragile once the layout starts growing. By the end, the main view reads like an outline rather than a wall of nested code.

We will work through three specific techniques: a custom `ViewModifier` that packages repeated card styling into a single `.cardStyle()` call, `@ViewBuilder` computed properties that break the `body` into named fragments, and a generic `InfoCard` container view that accepts content through a closure, the same pattern SwiftUI uses in `VStack` and `List`. Every recipe in the chapters that follow will use these techniques, so it is worth getting comfortable with them here.

Along the way, we adopt `ContentBuilder`, the unified result builder iOS 27 introduces as a drop- in replacement for `@ViewBuilder`.

Getting ready

Create a new Sw iftUI iOS app named **ViewComposer**.

How to do it...

We will start with a flat profile screen and then extract reusable pieces one at a time, starting with the card style and ending with a custom container view. These are the steps:

1. Open `ContentView.swift`. Before the `ContentView` struct, define a `CardStyle` view modifier that wraps its content in padding, a background, rounded corners, and a shadow. Then, add a `View` extension that provides a `.cardStyle()` convenience method:

```
struct CardStyle: ViewModifier {
  var cornerRadius: CGFloat = 16
  func body(content: Content) -> some View {
    content
      .padding()
      .background(Color(.systemBackground))
      .clipShape(RoundedRectangle(cornerRadius: cornerRadi
      .shadow(color: .black.opacity(0.1), radius: 8, x: 0,
  }
}
extension View {
  func cardStyle(cornerRadius: CGFloat = 16) -> some View
    modifier(CardStyle(cornerRadius: cornerRadius))
  }
}
```

The `View` extension wraps the `.modifier(CardStyle())` call so we can write `.cardStyle()` on any view, just like a built-in SwiftUI modifier. The `cornerRadius` parameter has a default value, so most call sites can omit it.

2. Below the extension, define a `StatItem` view for displaying a numeric value with a label. This is a simple extracted subview that we will reuse later:

```
struct StatItem: View {
  let value: String
  let label: String
  var body: some View {
    VStack(spacing: 4) {
      Text(value)
        .font(.title2)
        .fontWeight(.bold)
      Text(label)
        .font(.caption)
        .foregroundStyle(.secondary)
    }
  }
}
```

3. Define a custom container view called `InfoCard` that accepts a title, an SF Symbol icon, an icon color, and a `@ViewBuilder` content closure. The container applies our `.cardStyle()` modifier automatically, so every card in the app gets consistent styling:

```
struct InfoCard<Content: View>: View {
  let title: String
  let icon: String
  let iconColor: Color
  let content: Content
  init(title: String, icon: String, iconColor: Color = .gr
       @ViewBuilder content: () -> Content) {
```

```
    self.title = title
    self.icon = icon
    self.iconColor = iconColor
    self.content = content()
  }
  var body: some View {
    VStack(alignment: .leading, spacing: 12) {
      HStack {
        Image(systemName: icon)
          .foregroundStyle(iconColor)
        Text(title)
          .font(.headline)
      }
      content
    }
    .cardStyle()
  }
}
```

4. Now, replace the `ContentView` struct with the following. Its `body` is a `NavigationStack` containing a `ScrollView`, with two `InfoCard` instances for stats and skills, plus placeholder computed properties for `profileHeader`, `skillBadges`, and `extraContent`. These placeholders let the code compile while we implement each one in the following steps:

```
struct ContentView: View {
  @State private var showExtras = true
  var body: some View {
    NavigationStack {
      ScrollView {
        VStack(spacing: 24) {
          profileHeader
          InfoCard(title: "Stats",
                   icon: "chart.bar.fill", iconColor: .gre
            HStack(spacing: 24) {
              StatItem(value: "42", label: "Projects")
              StatItem(value: "128", label: "Commits")
```

```
                StatItem(value: "15", label: "PRs")
              }
              .frame(maxWidth: .infinity)
            }
            InfoCard(title: "Skills",
                     icon: "star.fill", iconColor: .yellow)
              skillBadges
            }
            extraContent
          }
          .padding()
        }
        .navigationTitle("Profile")
        .background(Color(.systemGroupedBackground))
      }
    }
    private var profileHeader: some View {
      Text("Profile Header")
    }
    private var skillBadges: some View {
      Text("Skill Badges")
    }
    private var extraContent: some View {
      Text("Extra Content")
    }
}
```

Make sure the preview canvas is in live mode; the profile dashboard renders with its **Stats** card fully built, while the header, skills, and extra content still show placeholder text.

5. Replace the `profileHeader` placeholder with the real implementation. The `@ViewBuilder` attribute lets us return the view hierarchy directly without an explicit `return` or a wrapping container:

```
@ViewBuilder
private var profileHeader: some View {
```

```
    VStack(spacing: 8) {
      Image(systemName: "person.circle.fill")
        .font(.system(size: 72))
        .foregroundStyle(.blue)
      Text("Jane Developer")
        .font(.title2)
        .fontWeight(.bold)
      Text("iOS Engineer")
        .font(.subheadline)
        .foregroundStyle(.secondary)
    }
    .cardStyle()
  }
```

6. Replace the `skillBadges` placeholder. We wrap the two `HStack` rows in a `Group` so that `.font(.caption)` and `.fontWeight(.semibold)` apply to all the text views at once, without repeating the modifiers on each one:

```
  private var skillBadges: some View {
    Group {
      HStack(spacing: 8) {
        Text("SwiftUI")
        Text("UIKit")
        Text("Combine")
      }
      HStack(spacing: 8) {
        Text("Swift")
        Text("Xcode")
      }
    }
    .font(.caption)
    .fontWeight(.semibold)
  }
```

7. Replace the `extraContent` placeholder. This property conditionally shows a **Recent Activity** card. The `@ViewBuilder`

attribute is required here because the property uses an `if` statement to decide whether to show content:

```
@ViewBuilder
private var extraContent: some View {
  if showExtras {
    InfoCard(title: "Recent Activity", icon: "clock.fill")
      VStack(alignment: .leading, spacing: 8) {
        Text("Merged PR #42: Fix navigation bug")
        Text("Opened issue: Dark mode support")
        Text("Pushed 3 commits to main")
      }
      .font(.subheadline)
      .foregroundStyle(.secondary)
    }
  }
}
```

With the preview still in live mode, the placeholders are gone: the real profile header, skills, and **Recent Activity** cards now fill the dashboard.

8. iOS 27 unifies SwiftUI's result builders under `ContentBuilder`, which we apply exactly like `@ViewBuilder`. Add a `footerNote` property that uses it for a two-line footer, reference it at the end of the dashboard's `VStack`, and switch the existing `extraContent` from `@ViewBuilder` to `@ContentBuilder` to confirm the two are interchangeable:



New in iOS 27. Built with Xcode 27, SwiftUI unifies its result builders: `ContentBuilder` is the new name for `ViewBuilder` (the SDK declares it as a typealias), and the type-specific builders



behind toolbars and commands now route through the same one, so `@ContentBuilder` works anywhere `@ViewBuilder` did. The unification applies even when we target earlier OS versions, as long as we build with Xcode 27.

```
var body: some View {
  NavigationStack {
    ScrollView {
      VStack(spacing: 24) {
        // ... profile header, stats, and skills cards
        extraContent
        footerNote
      }
      // ... padding
    }
    // ... navigationTitle and background
  }
}
// profileHeader and skillBadges variables
@ContentBuilder
private var extraContent: some View {
  // ...
}
@ContentBuilder
private var footerNote: some View {
  Divider()
  Text("Built with reusable views and modifiers.")
    .font(.footnote)
    .foregroundStyle(.secondary)
}
```

9. Make sure the preview canvas is in live mode and interact with the view, or run the app in Device Hub. The profile dashboard shows the profile header, stats, skills, and recent activity cards,

followed by the footer line. The preview should look like the following:

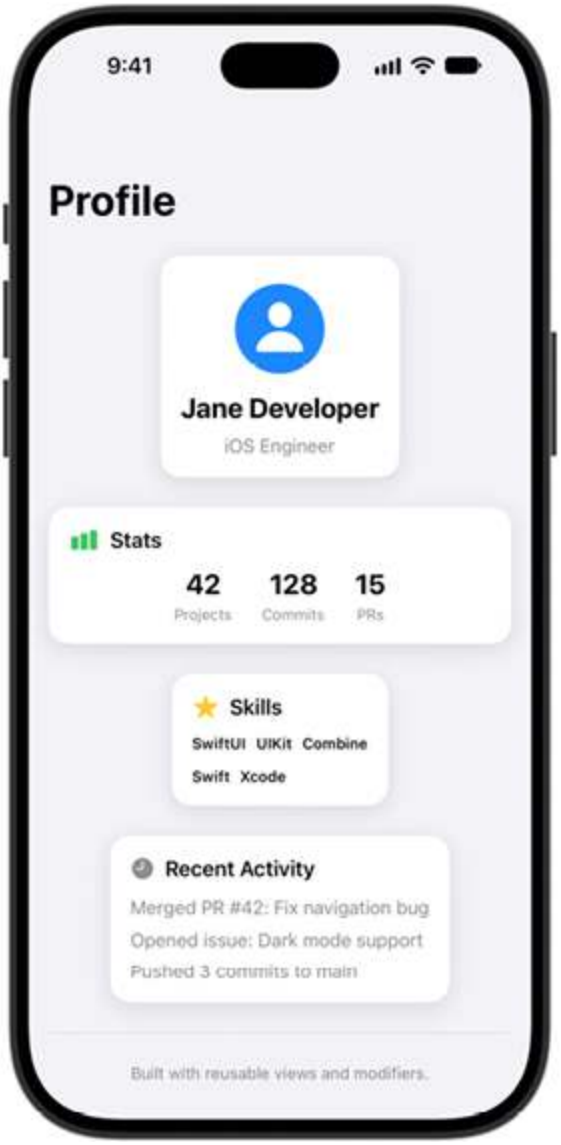


Figure 4.5: The ViewComposer app showing the profile da shboard with reusable card components

How it works...

In our dashboard, the repeated card treatment would become noisy fast if we had to restate the same padding, background, corner

radius, and shadow everywhere. `CardStyle` packages that visual treatment into one reusable unit, and the `View` extension turns it into the cleaner `.cardStyle()` call we used throughout the screen. This two-step pattern, define a `ViewModifier`, then add a `View` extension, is the usual SwiftUI way to turn repeated styling into something that reads like part of the framework.

In this recipe, `@ViewBuilder` is what lets `profileHeader` read like a view fragment instead of a helper function returning a single wrapped view. When we write `@ViewBuilder private var profileHeader: some View`, SwiftUI can assemble multiple child views and other control flow without forcing us into an extra container just to satisfy the type system. That is why `profileHeader` can return the avatar stack directly rather than falling back to the type erasure we discuss at the end of this section.

`extraContent` and `footerNote` use `@ContentBuilder`, the unified result builder iOS 27 introduces. Because `@ContentBuilder` and `@ViewBuilder` are now the same builder, it behaves the same: `extraContent` can branch on `if showExtras`, and `footerNote` can return a `Divider` and a `Text` together, both without a wrapping container. Switching `extraContent` from `@ViewBuilder` to `@ContentBuilder` changes nothing at runtime; it only shows that the two are interchangeable. The unification is a compile-time change that folds the previously type-specific builders behind toolbars and commands into one, and it applies even when we target earlier OS versions, as long as we build with Xcode 27.

The `Stats` and `Skills` cards share the same outer shell even though their inner content is different. `InfoCard` captures that shell once and

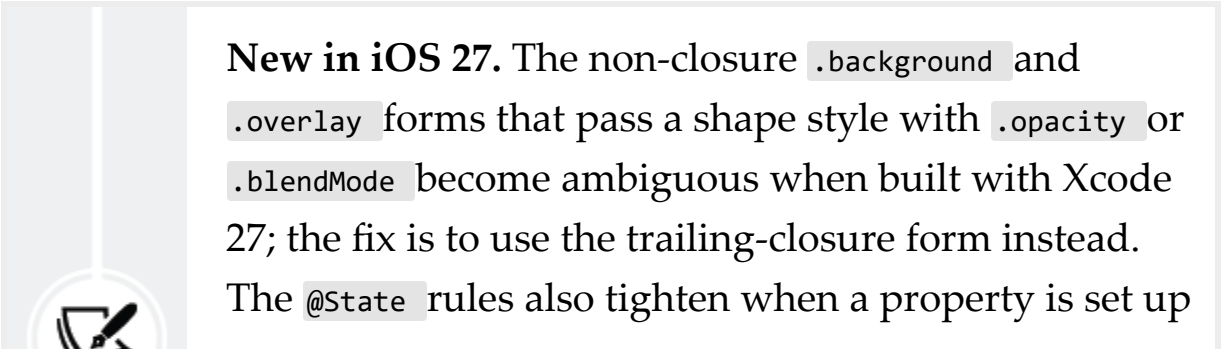
lets each call site supply its own content through a `@ViewBuilder` closure, which is exactly why the dashboard still reads cleanly as it grows. This is the same container pattern SwiftUI uses in `VStack`, `HStack`, and `List`.

In `skillBadges`, the two rows need the same font styling, but they do not need another layer of layout. That is exactly where `Group` helps: it lets us apply shared modifiers to several child views without changing spacing or alignment. We used the same idea earlier in the *Forms* section when we scoped `.submitLabel(.next)` to multiple form sections at once.

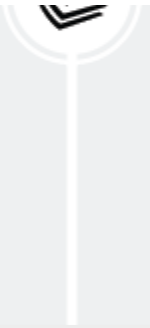
It can be tempting to reach for `AnyView` once a screen starts branching, because type erasure looks like an easy way to make mismatched view types compile. In a sample like this one, though, it would work against the composition style we just built. `@ViewBuilder` already lets us express conditional content without erasing types, so the resulting code stays closer to SwiftUI's natural model and gives the framework more information to optimize updates.

There's more...

Two Xcode 27 changes can cause build errors in existing code when we move a project to iOS 27.



New in iOS 27. The non-closure `.background` and `.overlay` forms that pass a shape style with `.opacity` or `.blendMode` become ambiguous when built with Xcode 27; the fix is to use the trailing-closure form instead. The `@State` rules also tighten when a property is set up



in a custom `init`. Initialize the view's other stored properties before assigning the `@State`. Do not assign in `init` to a `@State` that already has an inline default, and do not wrap another property wrapper around `@State`.

The closure form replaces the shape style with a trailing closure that returns it:

```
.background { Color.blue.opacity(0.3) }
```

The stricter `@State` rules come from the `State()` macro that Xcode 27 introduces, which we cover in the *State a nd Binding Basics* section of *Chapter 5*.

See also

- `ViewModifier` documentation: https://x.swiftuicookbook.com/yqm4n
- `ViewBuilder` documentation: https://x.swiftuicookbook.com/jl67t
- `ContentBuilder` documentation: https://x.swiftuicookbook.com/lx5vy
- `Group` documentation: https://x.swiftuicookbook.com/b8pbd
- `AnyView` documentation: https://x.swiftuicookbook.com/7mu8v

- *TN3211: Resolving SwiftUI source incompatibilities for State and ContentBuilder* article: https://x.swiftuicookbook.com/l36rn

Part 2

Data Flow in SwiftUI

With the essentials in place, this part of the book turns to the data behind the views. You will learn where each piece of state should live and how it flows between views, from local state and bindings to observable models, and how Combine fits alongside them. The same models and settings then reach the whole app through the environment, along with the accessibility support that makes the app usable by everyone. Next, you will persist data on the device with app storage, user defaults, SwiftData, and files, and fetch and decode remote data with async/await and Combine. The part closes with the tools that show what an app is actually doing: debugging views and state, measuring performance with Instruments, and automating checks with UI tests.

This part of the book includes the following chapters:

- `Chapter 5` , *State and Data Binding*
- `Chapter 6` , *App-Wide Settings and Accessibility*
- `Chapter 7` , *Data Persistence and Local Storage*
- `Chapter 8` , *Networking and Remote Data*
- `Chapter 9` , *Debugging, Testing, and Performance*

5

State and Data Binding

I have come to think of state as the real subject of SwiftUI; everything else is just how we display it. The first four chapters were about putting views on screen, and this chapter, which opens the data-flow part of the book, is about what makes those views come alive: where a value lives, and how it travels between the views that read and change it. State management is also the concept I most often see trip up developers moving from UIKit, where for years I copied values between a model and the interface by hand, setting a label's text, reloading a table, toggling a button's enabled flag, and hunting for the one spot I forgot. SwiftUI inverts that relationship: we declare where a value lives and which views depend on it, and the framework keeps the screen in sync for us. The work shifts from synchronizing the UI to deciding who owns each piece of state and how it flows.

We approach that in three passes, working from the language primitives up to the frameworks built on them. The first recipe uses `@State` and `@Binding` to build a theme designer where one view owns a value and child controls edit it through the `$` projection, reacting to edits with `.onChange()` and loading a starting palette with `.task()`. The Observation framework comes next, modeling a budget as an `@Observable` class held in `@State`, whose `State()` macro form in Xcode 27 builds that model lazily and just once; several screens read and edit it with `@Bindable`, and SwiftUI tracks changes per property so it re-renders only the views that used what changed. The last recipe looks at the pattern that came before it,

`ObservableObject` with `@Published`, observed through `@StateObject` and `@ObservedObject`, together with Combine, the framework underneath it, whose operators we use to debounce the search field in a movie browser. This is still the most common approach in existing SwiftUI projects, and very possibly the one you reach for today.

By the end of this chapter, you should be comfortable choosing the right tool for a given piece of state, whether that is a local value, a shared observable model, or a Combine pipeline, and confident moving data between views without the manual synchronization that older frameworks demanded.

In this chapter, we will cover the following recipes:

- State and Binding Basics
- Observation Framework
- Combine with Local Data

Technical requirements

The code in this chapter is based on Xcode 27.0 and iOS 27.0. You can download and install the latest version of Xcode from the App Store. All the code examples for this chapter can be found on GitHub at https://x.swiftuicookbook.com/ch05.

State and Binding Basics

State is where SwiftUI asks for the biggest change in mindset coming from UIKit. When working with UIKit, I treated the model and the view as two separate things and synchronized them with custom logic, and most of my bugs came from the two drifting apart: the one update I forgot to mirror somewhere. SwiftUI handles that synchronization for us, keeping the view in step with the model automatically. We declare where a value

lives, pass it to the views that need it, and the framework re-renders whatever depends on it the moment it changes. The hard part is no longer keeping the screen in sync with the data; it is deciding who owns each piece of state and how it flows to the views that read or change it.

In this recipe, we build a small theme designer: a screen with a live sample card at the top and a set of sliders below that restyle it in real time. It is a good vehicle for state because one theme value drives several views at once. One view only displays the theme, while two others edit it, so we see both directions of data flow on a single screen. Along the way we keep the theme in one place with `@State`, pass a read-only copy to the card, hand an editable reference to the control views with `@Binding` and the `$` projection, react to a palette selection with `.onChange()`, and load a set of starter palettes asynchronously with `.task` and a `@MainActor` function.

This is the first recipe in the data-flow part of the book, so we stay deliberately close to the language primitives. `@State` and `@Binding` are the foundation everything else rests on. Once an app outgrows a single owning view and needs to share a mutable model across many screens, we reach for the Observation framework, which we explore in the *Observation Framework* section .

Getting ready

Create a new SwiftUI iOS app named **ThemeDesigner** .

How to do it...

We'll build the recipe in four passes that mirror how state moves through the app: first the `Theme` model, then the read-only `ThemeCardView`, then the editing controls that mutate it via `@Binding`, and finally a `.task` that loads starter palettes asynchronously. Building this way lets us try the app after

almost every step and watch one more connection come alive. These are the steps:

1. Start with the data. Create a new file `Theme.swift` in a `Models` folder and define the value type the whole screen revolves around. A `Theme` is a plain `struct`, a value type, which is exactly what we want for a source of truth: when we copy it or hand it to a child view, that child gets its own snapshot rather than a shared reference. The `accentColor` computed property turns the stored `hue`, `saturation`, and `brightness` into a `Color`, and `default` gives us a starting theme to edit:

```
import SwiftUI
struct Theme: Identifiable {
  let id = UUID()
  var name: String
  var hue: Double
  var saturation: Double
  var brightness: Double
  var cornerRadius: Double
  var fontScale: Double
  var accentColor: Color {
    Color(hue: hue, saturation: saturation, brightness: brightne
  }
  static let `default` =
    Theme(name: "Custom", hue: 0.58, saturation: 0.70,
          brightness: 0.85, cornerRadius: 16, fontScale: 1.0)
}
```

2. Next, create `ThemeCardView.swift` in a `Views` folder. This view shows what a theme looks like applied to a real piece of UI, and it is deliberately read-only: it receives a `Theme` by value through a plain `let` property and never changes it. Because it only reads, it needs no binding. The button's action prints the current theme name, a small reminder that even a read-only view can act on the value it was handed:

```
import SwiftUI
struct ThemeCardView: View {
  let theme: Theme
  var body: some View {
    VStack(alignment: .leading) {
      Text(theme.name)
        .font(.system(size: 22 * theme.fontScale, weight: .bold)
      Text("Design once, apply everywhere.")
        .font(.system(size: 15 * theme.fontScale))
        .foregroundStyle(.secondary)
      Button("Get Started") {
        print("Get Started tapped (theme: \(theme.name))")
      }
      .buttonStyle(.borderedProminent)
      .tint(theme.accentColor)
    }
    .padding()
    .frame(maxWidth: .infinity, alignment: .leading)
    .background(theme.accentColor.opacity(0.15))
    .clipShape(RoundedRectangle(cornerRadius: theme.cornerRadius
    .overlay {
      RoundedRectangle(cornerRadius: theme.cornerRadius)
        .strokeBorder(theme.accentColor, lineWidth: 2)
    }
  }
}
#Preview {
  ThemeCardView(theme: .default)
    .padding()
}
```

The `#Preview` shows the card with the default theme. We will make this preview interactive once the control views exist.

3. Now, create the owner. Open `ContentView.swift` and replace the template with a `NavigationStack` containing a `Form`. The `@State private var theme` is the single source of truth for the screen: this view owns the value, and SwiftUI watches it for changes. We pass `theme` (not `$theme`) to `ThemeCardView`, because the card only needs to read it:

```
import SwiftUI
struct ContentView: View {
  @State private var theme = Theme.default
  var body: some View {
    NavigationStack {
      Form {
        Section("Preview") {
          ThemeCardView(theme: theme)
        }
      }
      .navigationTitle("Theme Designer")
    }
  }
}
#Preview {
  ContentView()
}
```

Open the live preview in the preview canvas. The card appears with the default theme. Nothing changes it yet, which is what we tackle next.

4. Create `ColorControlsView.swift`. This view edits the theme, so it needs more than a plain property. If we declare a regular `var theme` and try to wire a slider to it, Xcode stops us:

```
import SwiftUI
struct ColorControlsView: View {
  var theme: Theme
  var body: some View {
    Slider(value: $theme.hue)
  }
}
```

Xcode reports **Cannot find '$theme' in scope**. The `$` projection exists only for property wrappers, and a plain `var` is not one. We want a read-write reference back to the value `ContentView` owns, so we mark the property `@Binding`.

5. Mark the property `@Binding`, the read-write reference back to the value `ContentView` owns. Each `Slider` then binds to a sub-property with the `$` projection, for example `$theme.hue`, writing straight through to the source of truth:

```
import SwiftUI
struct ColorControlsView: View {
  @Binding var theme: Theme
  var body: some View {
    VStack {
      LabeledContent("Hue") {
        Slider(value: $theme.hue)
      }
      LabeledContent("Saturation") {
        Slider(value: $theme.saturation)
      }
      LabeledContent("Brightness") {
        Slider(value: $theme.brightness)
      }
    }
  }
}
#Preview {
  @Previewable @State var theme = Theme.default
  VStack {
    ThemeCardView(theme: theme)
      .padding()
    Form {
      ColorControlsView(theme: $theme)
    }
  }
}
```

The `#Preview` is where this gets interesting. We declare a `@Previewable @State` theme to own a value inside the preview canvas, render the card to read it, and pass `$theme` to the controls to write it. Dragging a slider in the preview restyles the card live, with no running app:

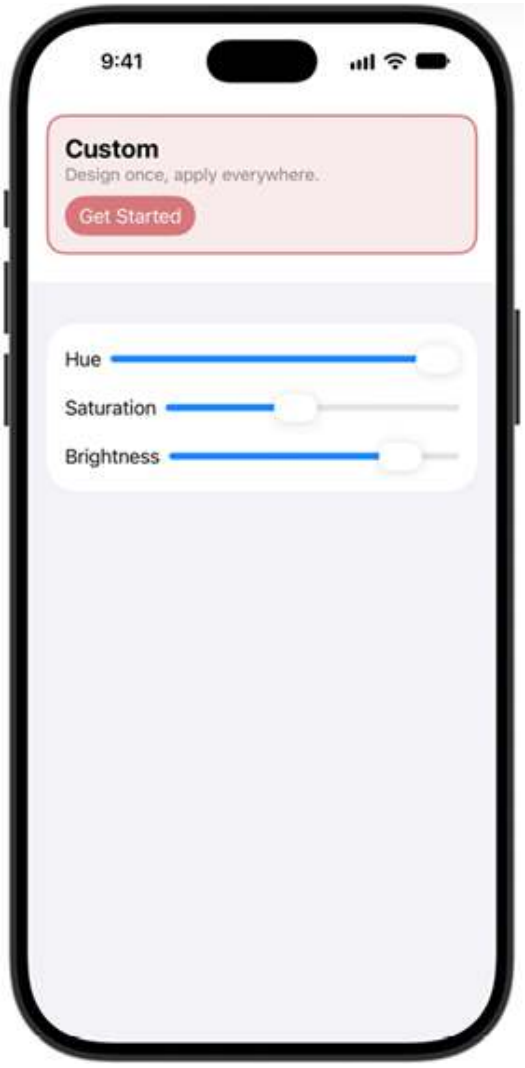


Figure 5.1: The color controls preview running interactively in the Xcode preview canvas

6. Wire the controls into the owner. Back in `ContentView`, add a **Colors** section. It is tempting to pass the theme straight through:

```
Section("Colors") {
  ColorControlsView(theme: theme)
}
```

This time the error is **Cannot convert value of type 'Theme' to expected argument type 'Binding<Theme>'** . `ColorControlsView` declared `@Binding var theme`, so it does not want a `Theme`; it wants a

binding to one. The `$` projection on our `@State` produces exactly that.

7. Pass the projected binding, so the section hands `$theme` to the control view rather than a plain `Theme`:

```
Form {
  // ... Preview section
  Section("Colors") {
    ColorControlsView(theme: $theme)
  }
}
```

   In the preview canvas, drag the color sliders. The card restyles as we move them: the slider writes through the binding to `ContentView`'s `@State`, and SwiftUI re-renders the card because it reads that same value.

8. We want a second group of controls for shape and text, and those sliders need ranges. The valid range for a corner radius is a rule of the editor, not a property of any single theme, so it belongs on the type rather than on each instance. Add two `static` constants to `Theme`:

```
struct Theme: Identifiable {
  // ... existing properties and default
  static let cornerRadiusRange: ClosedRange<Double> = 0...40
  static let fontScaleRange: ClosedRange<Double> = 0.8...1.6
}
```

9. Create `ShapeControlsView.swift`. It follows the same `@Binding` pattern as the color controls, but its sliders take explicit ranges from the constants we just added, plus a `step` so the values move in sensible increments:

```
import SwiftUI
struct ShapeControlsView: View {
```

```
  @Binding var theme: Theme
  var body: some View {
    VStack {
      LabeledContent("Corner Radius") {
        Slider(value: $theme.cornerRadius, in: Theme.cornerRadiu
      }
      LabeledContent("Font Scale") {
        Slider(value: $theme.fontScale, in: Theme.fontScaleRange
      }
    }
  }
}
#Preview {
  @Previewable @State var theme = Theme.default
  VStack {
    ThemeCardView(theme: theme)
      .padding()
    Form {
      ShapeControlsView(theme: $theme)
    }
  }
}
```

10. Add the matching section to `ContentView` so the shape controls feed the same theme:

```
Form {
  // ... Preview and Colors sections
  Section("Shape & Text") {
    ShapeControlsView(theme: $theme)
  }
}
```

    Back in the preview canvas, corner radius and font scale now reshape the card alongside the color sliders, all driven by the one `theme` value.

11. Real apps rarely start from a single hard-coded default; they load a few curated options first. Simulate that by adding a `loadPresets()` function to `Theme` in an extension. It is marked `@MainActor` and is `async`

, and the `Task.sleep` stands in for a network or database call that takes a moment to return:

```
extension Theme {
  @MainActor
  static func loadPresets() async -> [Theme] {
    try? await Task.sleep(for: .seconds(2))
    return [
      Theme(name: "Seafoam", hue: 0.45, saturation: 0.55,
            brightness: 0.80, cornerRadius: 20, fontScale: 1.1),
      Theme(name: "Sunset", hue: 0.03, saturation: 0.75,
            brightness: 0.95, cornerRadius: 8, fontScale: 1.0),
      Theme(name: "Grape", hue: 0.78, saturation: 0.65,
            brightness: 0.80, cornerRadius: 28, fontScale: 1.2),
      Theme(name: "Slate", hue: 0.58, saturation: 0.20,
            brightness: 0.55, cornerRadius: 4, fontScale: 0.9)
    ]
  }
}
```

12. Now, load those presets when the screen appears. In `ContentView`, add state for the loaded `presets`, the selected preset name, and an `isLoading` flag, then attach a `.task` that calls `loadPresets()`. Add a **Starter Palette** section that shows a `ProgressView` while loading and a `Picker` of palette names once the data arrives:

```
struct ContentView: View {
  @State private var theme = Theme.default
  @State private var presets: [Theme] = []
  @State private var selectedPresetName = Theme.default.name
  @State private var isLoading = true
  var body: some View {
    NavigationStack {
      Form {
        // ... Preview section
        Section("Starter Palette") {
          if isLoading {
            HStack {
              ProgressView()
              Text("Loading palettes…")
```

```
                    .foregroundStyle(.secondary)
                }
            } else {
                Picker("Start from", selection: $selectedPresetName)
                    Text("Custom").tag("Custom")
                    ForEach(presets) { preset in
                        Text(preset.name).tag(preset.name)
                    }
                }
            }
            // ... Colors and Shape & Text sections
        }
        .navigationTitle("Theme Designer")
        .task {
            presets = await Theme.loadPresets()
            isLoading = false
        }
    }
  }
}
```

In the preview canvas, we see the spinner for two seconds, then the picker appears. The `.task` runs when the view appears and is canceled automatically if it disappears.



`.task` and `async` / `await` are part of Swift Concurrency, a large topic in its own right. You do not need to master it to follow this chapter: here, `.task` is just a convenient place to load data when the view appears, and the rest of the recipe is about how that data drives the UI. We cover asynchronous work in depth in the *Async/await with Remote APIs* section of *Chapter 8*.

13. Selecting a palette should restyle the card, so we react to the selection with `.onChange(of:)`. When the name changes, we look up the

matching preset and apply it, or fall back to the default when the user picks **Custom** :

```
.navigationTitle("Theme Designer")
.task {
  presets = await Theme.loadPresets()
  isLoading = false
}
.onChange(of: selectedPresetName) {
  if selectedPresetName == "Custom" {
    theme = .default
  } else if let preset = presets.first(where: {
    $0.name == selectedPresetName
  }) {
    theme = preset
  }
}
```

Back in the preview canvas, pick a palette. The card jumps to that palette, and the sliders below let us fine-tune it from there.

14. With both control groups in place, we can finally make the card's own preview interactive, the same way we did for the controls. Update the `#Preview` in `ThemeCardView.swift` to own a theme and host both control groups:

```
#Preview {
  @Previewable @State var theme = Theme.default
  VStack {
    ThemeCardView(theme: theme)
      .padding()
    Form {
      ColorControlsView(theme: $theme)
      ShapeControlsView(theme: $theme)
    }
  }
}
```

15. Before moving on, let's watch SwiftUI's re-rendering directly. Add `let _ = Self._printChanges()` as the first line of `ThemeCardView`'s `body`, and the same line to `ContentView`'s `body`. It logs to the console every time the view re-evaluates, together with the reason:

```
var body: some View {
  let _ = Self._printChanges()
  // ... existing view content
}
```



The `print` statements and `Self._printChanges()` calls in this chapter are quick, disposable probes: we add them, read the output, then delete them. To watch the same thing without editing your code, set a breakpoint and give it a *Log Message* action (or use `os_log`), so the debugger logs on each hit and leaves the source untouched.

Run the app in Device Hub and drag the Hue slider. The console reports both views re-evaluating on each change, and why: `ContentView` because its `@State _theme` changed, and `ThemeCardView` because the value it was handed is now different. Running in Device Hub keeps this console output easy to read, while the preview canvas logs the same re-evaluations in a more verbose form. `Self._printChanges()` is a quick window into what SwiftUI is doing under the hood, and we cover it more fully in the *Debugging Views and State* section of [`Chapter 9`]. Remove the two lines once we have seen the output:



Figure 5.2: Self._printChanges() reporting each view re-evaluation and its cause

16. Run the app in Device Hub one last time and put it through its paces: wait for the palettes to load, pick one, then adjust hue, brightness, corner radius, and font scale and watch the card update with every change. The finished screen should look like the following:

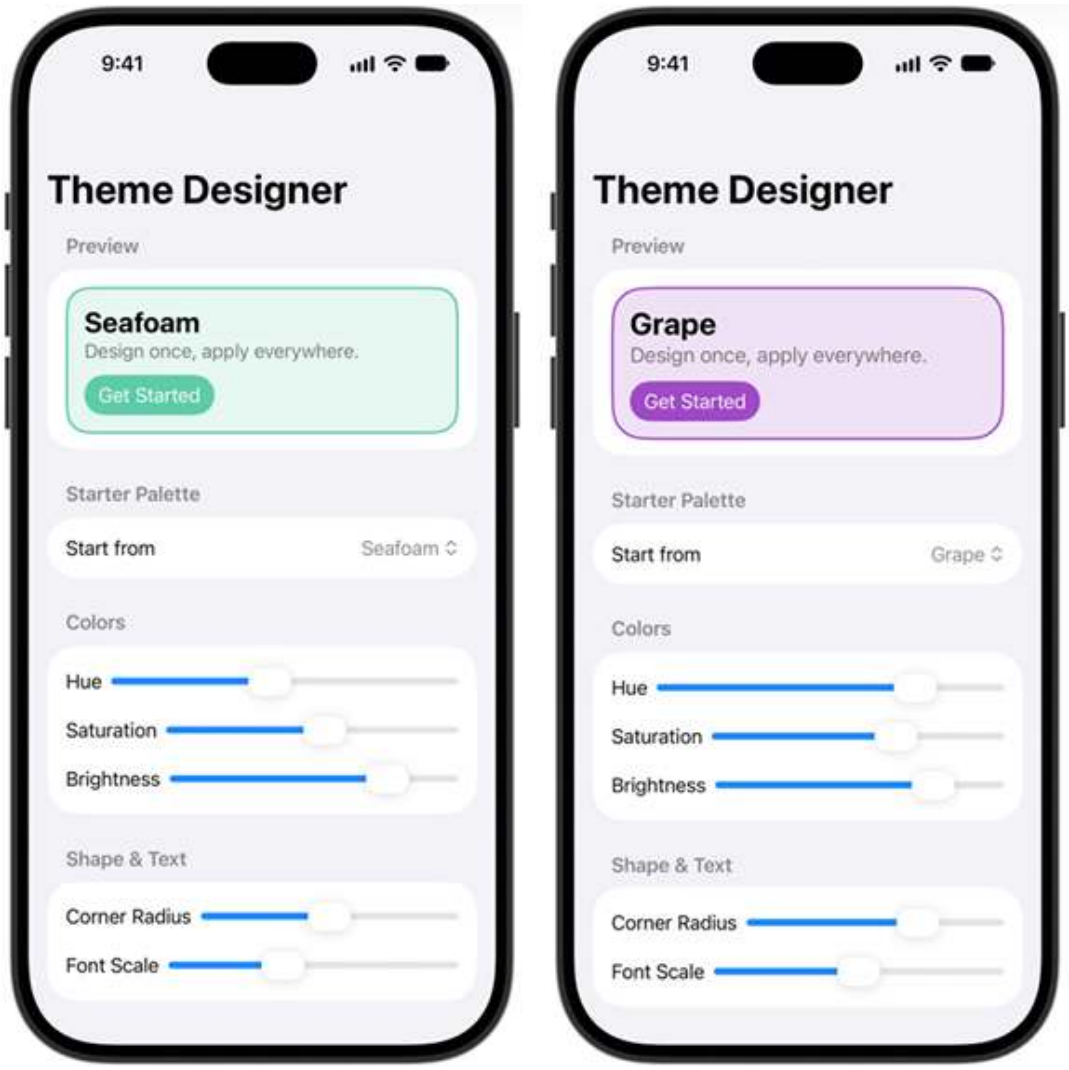


Figure 5.3: The same theme designer screen with two different palettes applied

How it works…

Every change we made flowed from one decision: `ContentView` owns the `theme` with `@State`. `@State` tells SwiftUI that this view holds a value that can change over time, and that the view, along with anything depending on the value, should be re-rendered when it does. We never wrote code to refresh the card after a slider moved; we only changed the `theme`, and SwiftUI recomputed the parts of the screen that read it. Under Xcode 27, `@State` is implemented by the `State()` macro. That leaves value types such

as `theme` behaving exactly as before, but it makes the model classes we store in state initialize lazily and only once, which the *Observation Framework* section explores, with a short before-and-after example.

The `Self._printChanges()` output we watched is worth unpacking, because it shows what that recomputation really is. When the `theme` changes, SwiftUI marks the views that read it for an update, reruns their `body`, and diffs the result against the previous one, touching only the parts that differ. Re-running a `body` is not the same as recreating the view: SwiftUI keeps a persistent tree of the views on screen, keyed by each view's identity, so a view's own `@State` survives across these re-evaluations instead of resetting.

The reason printed after each view name tells us what triggered the re-evaluation. A leading underscore marks a property's backing storage, and the `State()` macro gives that storage two of them, so `__theme changed` means the storage behind `theme` now holds a new value, a genuine value change driving the update. The other two tokens describe the view itself rather than one of its properties. `@self` means the view value as a whole compares unequal to its previous version, because one of its stored inputs differs, which is why `ThemeCardView` reports it when the card receives a changed `theme`. `@identity` means SwiftUI assigned the view a brand-new identity rather than re-evaluating an existing one. The two usually appear together on a view's first render, when both its value and its identity are new, while later updates report just the property that changed.

Identity is a deep topic that matters most for navigation and animation, which later chapters return to; here the point is that a state change reruns the affected `body` properties, and `_printChanges()` lets us see which ones and why. The Observation framework makes this re-rendering far more selective, and we will prove it the same way in the *Observation Framework* section.

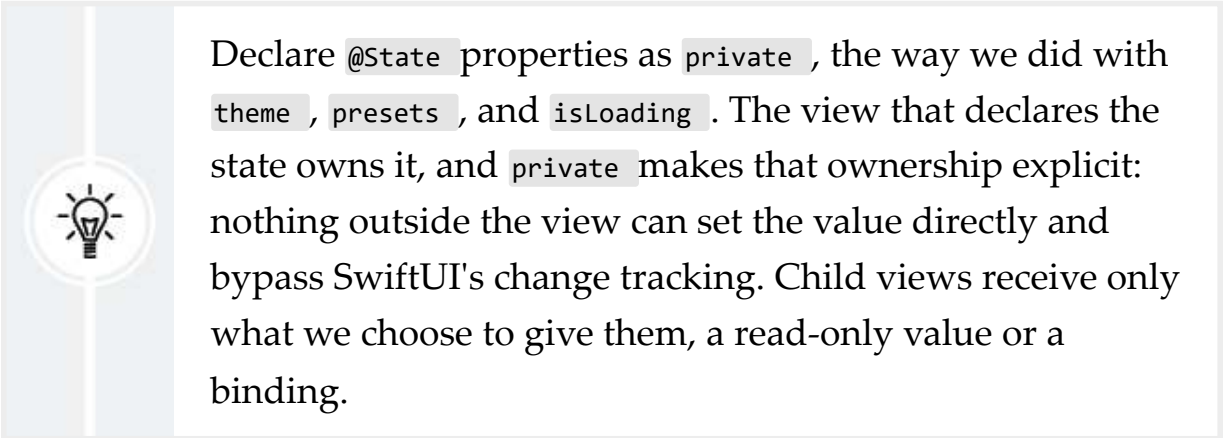
The two ways we handed the theme to child views are the heart of the recipe. `ThemeCardView` takes `theme` by value with a plain `let`: it receives a snapshot it can read but not change, which is exactly right for a view whose only job is to display. Because `Theme` is a value type, that snapshot is genuinely independent; even if the card mutated its copy, the change would stay local and never reach `ContentView`'s value. `ColorControlsView` and `ShapeControlsView` take `@Binding var theme` instead, a two-way reference to the value that still lives in `ContentView`. The binding, not the type, is what grants write access back to the original. The rule of thumb is simple: if a child only reads, pass the value; if a child must change it, pass a binding.

The `$` prefix is what connects the two sides. On a `@State` property, `$theme` is the projected value, a `Binding<Theme>` that reads and writes the underlying storage. Passing `$theme` to a control view satisfies its `@Binding`. Inside those views we go one level deeper: `$theme.hue` projects a `Binding<Double>` to a single sub-property, which is what a `Slider` expects. So when a slider moves, it writes through the binding to `ContentView`'s `@State`, the source of truth changes, and the card, reading that same value, redraws. One value, several editors, no manual synchronization.

Loading the palettes introduced two more pieces. `.task` attaches asynchronous work to the view's lifetime: it starts when the view appears and is canceled automatically if the view goes away, which spares us from managing a `Task` by hand. Its closure runs on the main actor, so assigning the result to our `@State` is safe. We also marked `loadPresets()` as `@MainActor`. In plain terms, `@MainActor` just means *run this on the main thread*, the one thread where UI work must happen. State that drives the UI must be touched on the main actor, and the annotation keeps the function and the value it returns there. In a real app the `Task.sleep` would be a `URLSession` call or a database read; we will use `.task` for genuine networking in the *Async/await with Remote APIs* section of *Chapter 8*.

We used `.onChange(of: selectedPresetName)` to bridge two pieces of state: the picker's selection and the theme. When the selected name changes, we apply the matching preset or reset to the default. `.onChange` is for side effects that should happen in response to a change, which is different from a value we can simply derive. The `accentColor` on `Theme`, for instance, is a computed property because it is always a function of the current theme, not something we react to. Reaching for `.onChange` when a computed property would do is a common way to end up with state that drifts out of sync.

A few smaller decisions are worth calling out. The slider ranges live on the type as `static` constants because they describe what the editor allows, not what a particular theme is. The interactive previews lean on `@Previewable @State`, which lets a preview own mutable state, so the preview canvas behaves like a tiny running app; we cover that in the *Previews with Mock Data* section of *Chapter 3*. The `Picker` follows the same tagging pattern from the *Pickers, Toggles, and SF Symbols* section of *Chapter 1*, where each option supplies the value written into the selection binding. This pattern works well when a single view owns the state, but once several unrelated views need to share and mutate the same model, threading bindings through every layer becomes tedious. That is the problem the Observation framework solves, and it is where we go next in the *Observation Framework* section.



Declare `@State` properties as `private`, the way we did with `theme`, `presets`, and `isLoading`. The view that declares the state owns it, and `private` makes that ownership explicit: nothing outside the view can set the value directly and bypass SwiftUI's change tracking. Child views receive only what we choose to give them, a read-only value or a binding.

See also

- `State` documentation: https://x.swiftuicookbook.com/f0hhs
- `Binding` documentation: https://x.swiftuicookbook.com/32obq
- `Slider` documentation: https://x.swiftuicookbook.com/8ktlu
- `onChange(of:initial:_:)` documentation: https://x.swiftuicookbook.com/f9mxt
- `task(name:priority:file:line:_:)` documentation: https://x.swiftuicookbook.com/szdvp
- `MainActor` documentation: https://x.swiftuicookbook.com/fkmun

Observation Framework

In the previous section the theme lived inside a single view, which is the simplest place for state to be. Real apps are rarely that tidy: a model usually has to be read by one screen, edited by another, and summarized by a third, all looking at the same data. For years the SwiftUI answer was a class conforming to `ObservableObject` with `@Published` properties, and it worked, but it had a blunt edge. Any change to any `@Published` property told every view watching that object to re-render, whether or not the view actually used the property that changed. The Observation framework, introduced in iOS 17, replaces that with something far more precise: mark a class `@Observable` and SwiftUI tracks exactly which properties each view reads, then re-renders only the views affected by a given change.

In this recipe, we build a monthly budget screen backed by one shared model. A summary header shows the spent and remaining amounts, a list shows each expense, an editor changes a single expense, and a settings screen changes the monthly limit, and all four read from the same object.

That gives us a realistic place to see observation work: we keep the model in an `@Observable` class, hand editable references to child views with `@Bindable` , exclude a constant from tracking with `@ObservationIgnored` , and then lift the hood with `Self._printChanges()` and `withObservationTracking()` to watch the fine-grained updates for ourselves.

This recipe builds directly on the *State and Binding Basics* section, so if `@State` and the `$` projection are not yet familiar, it is worth readin g that section first.

Getting ready

Create a new SwiftUI iOS app named **BudgetTracker** .

How to do it...

We will build the data first, then the views that read it, then the views that edit it, and finish by looking under the hood at how SwiftUI decides what to re-render. These are the steps:

1. Start with the smaller model. Create a new file `Expense.swift` in a `Models` folder. First, define an `ExpenseCategory` enum that pairs each category with an SF Symbol, then an `@Observable` class for a single expense. The `@Observable` macro rewrites the class so SwiftUI can track reads and writes of its properties. We mark `id` with `@ObservationIgnored` because it is a constant that never changes, so there is no reason to track it:

```
import Foundation
enum ExpenseCategory: String, CaseIterable, Identifiable {
  case food = "Food"
  case transport = "Transport"
  case entertainment = "Entertainment"
  case bills = "Bills"
  case other = "Other"
```

```
  var id: String { rawValue }
  var symbol: String {
    switch self {
    case .food: "fork.knife"
    case .transport: "car.fill"
    case .entertainment: "music.note"
    case .bills: "doc.text.fill"
    case .other: "tag.fill"
    }
  }
}
@Observable
class Expense: Identifiable {
  @ObservationIgnored let id = UUID()
  var title: String
  var amount: Double
  var category: ExpenseCategory
  init(title: String, amount: Double, category: ExpenseCategory)
    self.title = title
    self.amount = amount
    self.category = category
  }
}
```

2. Create `Budget.swift` in the same folder. The `Budget` is also `@Observable` and holds an array of `Expense` objects, so observation nests: a change to an expense's amount flows up through the budget's computed `total` and `remaining` . The `monthlyLimitRange` is a `static` constant because it describes what the settings slider allows, not a property of any one budget:

```
import Foundation
@Observable
class Budget {
  var name: String
  var monthlyLimit: Double
  var expenses: [Expense]
  init(name: String, monthlyLimit: Double, expenses: [Expense] =
    self.name = name
    self.monthlyLimit = monthlyLimit
    self.expenses = expenses
```

```
  }
  var total: Double {
    expenses.reduce(0) { $0 + $1.amount }
  }
  var remaining: Double {
    monthlyLimit - total
  }
  static let monthlyLimitRange: ClosedRange<Double> = 500...5000
  static var sample: Budget {
    Budget(name: "May Budget", monthlyLimit: 2000, expenses: [
      Expense(title: "Groceries", amount: 320, category: .food),
      Expense(title: "Gas", amount: 60, category: .transport),
      Expense(title: "Concert Tickets", amount: 150, category: .
      Expense(title: "Electricity", amount: 95, category: .bills
    ])
  }
}
```

3. Now, create the views that only display the model. Create `SummaryView.swift` in a `Views` folder. It takes the budget through a plain `let` because it only reads from it. Reading `budget.total`, `budget.remaining`, and `budget.monthlyLimit` here is what registers this view's interest in those specific properties:

```
import SwiftUI
struct SummaryView: View {
  let budget: Budget
  var body: some View {
    VStack {
      Text(budget.name)
        .font(.headline)
      HStack {
        VStack(alignment: .leading) {
          Text("Spent")
            .font(.caption)
            .foregroundStyle(.secondary)
          Text(budget.total, format: .currency(code: "USD"))
            .font(.title2.bold())
        }
        Spacer()
        VStack(alignment: .trailing) {
```

```
          Text("Remaining")
            .font(.caption)
            .foregroundStyle(.secondary)
          Text(budget.remaining, format: .currency(code: "USD"))
            .font(.title2.bold())
            .foregroundStyle(budget.remaining < 0 ? .red : .gree
        }
      }
      ProgressView(
        value: min(budget.total, budget.monthlyLimit),
        total: budget.monthlyLimit)
    }
  }
}
#Preview {
  SummaryView(budget: .sample)
    .padding()
}
```

4. Create `ExpenseRowView.swift`, also a read-only view. Each row reads only its own expense's `title`, `amount`, and `category`, so each row depends on its own data and nothing else:

```
import SwiftUI
struct ExpenseRowView: View {
  let expense: Expense
  var body: some View {
    HStack {
      Image(systemName: expense.category.symbol)
        .foregroundStyle(.tint)
        .frame(width: 28)
      Text(expense.title)
      Spacer()
      Text(expense.amount, format: .currency(code: "USD"))
        .foregroundStyle(.secondary)
    }
  }
}
#Preview {
  List {
    ExpenseRowView(
      expense: Expense(title: "Coffee", amount: 4.5, category: .
```

```
  }
}
```

5. Assemble the main screen. Open `ContentView.swift` and replace the template. The view creates the `Budget` and owns it with `@State`, then hands the same instance to `SummaryView` and to each `ExpenseRowView`. The toolbar button adds an expense, and `.onDelete`, covered in the *Lists and Sections* section of *Chapter 2*, removes one with a swipe:

```
import SwiftUI
struct ContentView: View {
  @State private var budget = Budget.sample
  var body: some View {
    NavigationStack {
      List {
        Section {
          SummaryView(budget: budget)
        }
        Section("Expenses") {
          ForEach(budget.expenses) { expense in
            ExpenseRowView(expense: expense)
          }
          .onDelete(perform: deleteExpenses)
        }
      }
      .navigationTitle("Budget")
      .toolbar {
        ToolbarItem(placement: .topBarTrailing) {
          Button(action: addExpense) {
            Image(systemName: "plus")
          }
        }
      }
    }
  }
  private func addExpense() {
    budget.expenses.append(
      Expense(title: "New Expense", amount: 0, category: .other)
  }
  private func deleteExpenses(at offsets: IndexSet) {
    budget.expenses.remove(atOffsets: offsets)
```

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

In the preview canvas, the summary and the expense list both appear, and adding or deleting an expense updates the total because `SummaryView` reads `budget.total`. We own the model with `@State` here for the same reason we owned the theme with `@State` in the previous section: the view that creates a value owns it.

6. Let the user edit an expense. Create `ExpenseEditorView.swift`. Unlike `ContentView`, this view does not create the expense; it receives one that already lives in the budget's array, and its whole job is to change it. That ownership difference decides how we declare the property. A plain `let` gives a read-only reference, which is enough to display the expense but not to write back to it, and a `TextField` needs a writable binding. If we try to bind one to a `let`, Xcode makes the point for us:

```
import SwiftUI
struct ExpenseEditorView: View {
  let expense: Expense
  var body: some View {
    TextField("Title", text: $expense.title)
  }
}
```

Xcode reports **Cannot find '$expense' in scope**. The `$` projection that produces a binding does not exist on a plain `let`. We used `$` on `@State` in the previous section, where the view owned the value; here the view does not own the expense, it only borrows a reference to one, so we reach for `@Bindable` instead. It exists for exactly this case: a

view that needs to write into an `@Observable` object handed to it from outside.

7. Mark the property `@Bindable var expense`, which gives the projection back, so `$expense.title`, `$expense.amount`, and `$expense.category` each write straight through to the shared object:

```
import SwiftUI
struct ExpenseEditorView: View {
  @Bindable var expense: Expense
  var body: some View {
    Form {
      Section("Details") {
        TextField("Title", text: $expense.title)
        TextField("Amount", value: $expense.amount,
                  format: .currency(code: "USD"))
          .keyboardType(.decimalPad)
      }
      Section("Category") {
        Picker("Category", selection: $expense.category) {
          ForEach(ExpenseCategory.allCases) { category in
            Label(category.rawValue, systemImage: category.symbo
              .tag(category)
          }
        }
      }
    }
    .navigationTitle("Edit Expense")
    .navigationBarTitleDisplayMode(.inline)
  }
}
#Preview {
  NavigationStack {
    ExpenseEditorView(
      expense: Expense(title: "Groceries", amount: 320, category
  }
}
```

8. Back in `ContentView`, wrap each row in a `NavigationLink` so tapping it opens the editor:

```
ForEach(budget.expenses) { expense in
  NavigationLink {
    ExpenseEditorView(expense: expense)
  } label: {
    ExpenseRowView(expense: expense)
  }
}
.onDelete(perform: deleteExpenses)
```

In the preview canvas, tap an expense and change its amount. When we return to the list, the row shows the new amount and the summary's total and remaining have updated too. We never wrote code to keep them in sync: the editor wrote through the binding to the shared `Expense`, and every view reading that data re-rendered.

9. Add the settings screen for the monthly limit. Create `SettingsView.swift`, which also takes `@Bindable var budget` so the slider can bind to `$budget.monthlyLimit`:

```
import SwiftUI
struct SettingsView: View {
  @Bindable var budget: Budget
  var body: some View {
    Form {
      Section("Budget") {
        TextField("Name", text: $budget.name)
      }
      Section("Monthly Limit") {
        Slider(value: $budget.monthlyLimit,
               in: Budget.monthlyLimitRange, step: 50)
        LabeledContent("Limit", value: budget.monthlyLimit,
                       format: .currency(code: "USD"))
      }
    }
    .navigationTitle("Settings")
    .navigationBarTitleDisplayMode(.inline)
  }
}
#Preview {
  NavigationStack {
```

```
      SettingsView(budget: .sample)
    }
  }
```

10. Add a leading toolbar item to `ContentView` that opens it:

```
ToolbarItem(placement: .topBarLeading) {
  NavigationLink {
    SettingsView(budget: budget)
  } label: {
    Image(systemName: "gearshape")
  }
}
```

In the preview canvas, open settings and drag the limit slider. The remaining amount in the summary tracks it. The finished screen and its two editing surfaces look like the following:

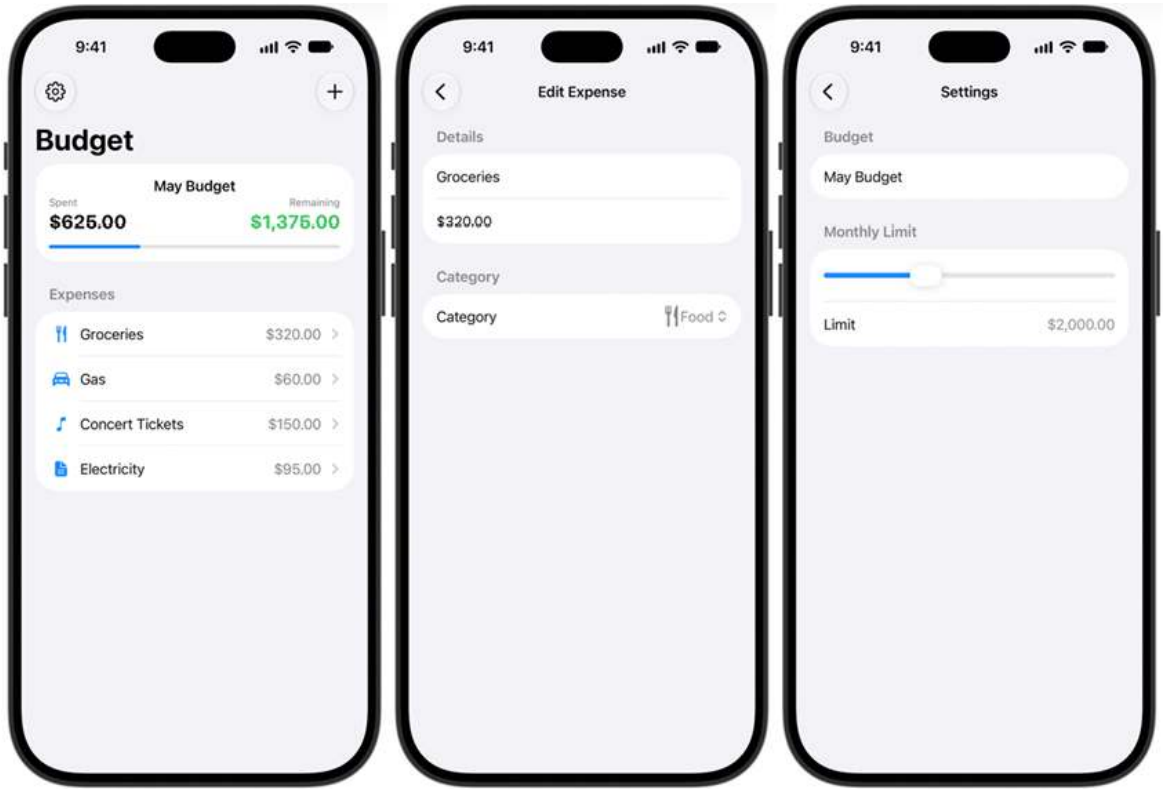


Figure 5.4: The budget screen, the expense editor, and the settings screen, all backed by one shared model

11. Now, let's prove the fine-grained behavior for ourselves, with the same `Self._printChanges()` helper we used in the *State and Binding Basics* section. Temporarily add it to the top of the `body` property of both `SummaryView` and `ExpenseRowView`:

```
var body: some View {
  let _ = Self._printChanges()
  // ... existing view content
}
```

    Run the app in Device Hub and drag the limit slider in settings. The console reports `SummaryView` re-evaluating, but not a single `ExpenseRowView`, because the rows never read `monthlyLimit`. The preview canvas reports the same thing (many readers prefer it, to avoid running the app in Device Hub), though it names the changed key path in a more raw form rather than the tidy view name:

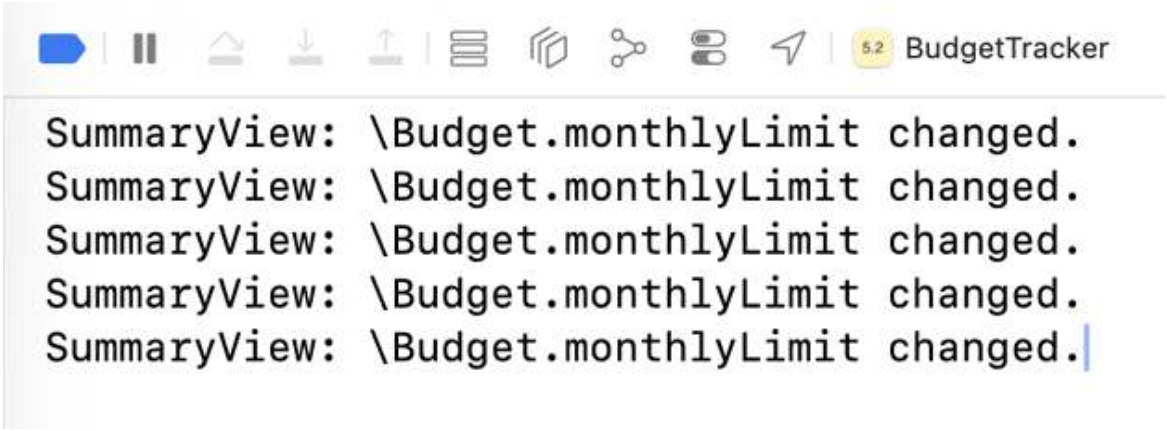


Figure 5.5: Dragging the monthly limit re-evaluates only the summary

12. Now, edit one expense's amount. This time `SummaryView` and that one row re-evaluate, because both read the amount, while the other rows stay silent:

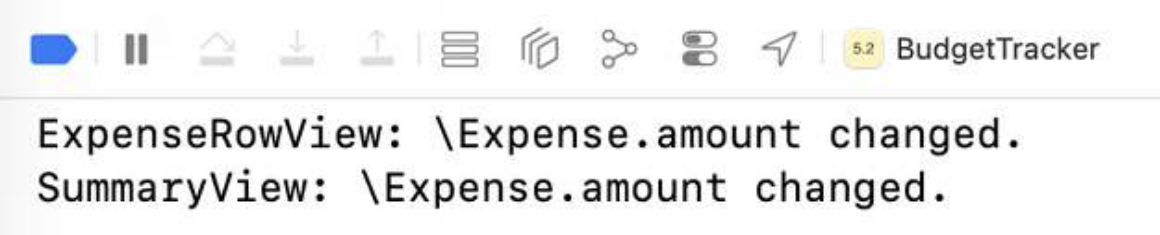


Figure 5.6: Editing one expense re-evaluates only that row and the summary

13. Now, remove the two `Self._printChanges()` lines. The tracking that drives those updates is available directly through `withObservationTracking()`, the primitive SwiftUI uses internally. Add a `trackRemaining(of:)` function to `ContentView` and call it when the screen appears with a `.task` modifier on the `List`:

```
struct ContentView: View {
  // ... existing properties
  var body: some View {
    NavigationStack {
      List {
        // ... summary and expense sections
      }
      .navigationTitle("Budget")
      .toolbar {
        // ... settings and add items
      }
      .task {
        trackRemaining(of: budget)
      }
    }
  }
  private func trackRemaining(of budget: Budget) {
    withObservationTracking {
      _ = budget.remaining
    } onChange: {
      print("remaining is about to change")
      Task { @MainActor in trackRemaining(of: budget) }
    }
  }
}
```



`withObservationTracking()` is a peek under the hood at the mechanism SwiftUI runs for every view's `body` on your behalf. If it feels like a lot at once, treat it as optional: everyday view code reaches for `.onChange(of:)`, which we just used, and never calls `withObservationTracking()` directly.

The closure reads `budget.remaining`, which registers a dependency on it. The `onChange` handler fires once, the next time any property feeding `remaining` changes, so we re-arm it from inside the handler. This is exactly what SwiftUI does around every view's `body`, except it does it for us. In view code we reach for `.onChange(of:)` instead; `withObservationTracking()` is the lower-level tool:

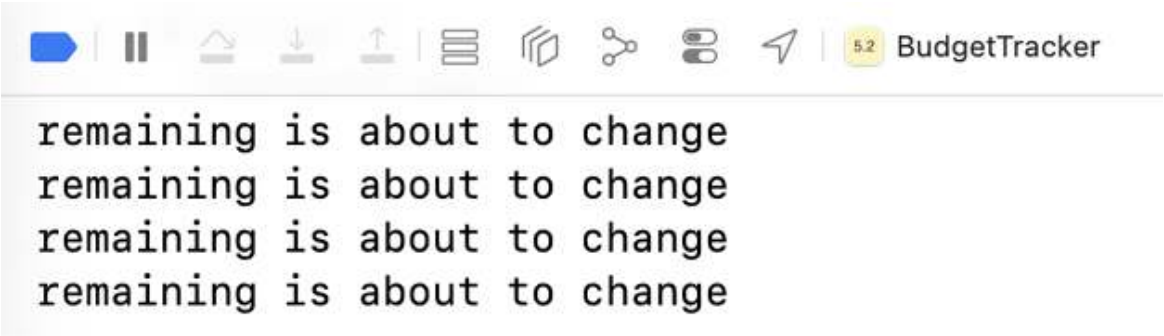


Figure 5.7: withObservationTracking() logging each time the remaining value is about to change

14. Remove the `trackRemaining(of:)` method and its `.task` call. Then run the app in Device Hub one more time and exercise it: add an expense, edit its amount and category, delete one with a swipe, and change the monthly limit in settings. Every screen stays consistent with the one shared model, and only the views that read a changed property redraw.

How it works...

The whole recipe turns on one decision: the budget is a reference type, an `@Observable class`, not a value type. `ContentView` creates it once and owns it with `@State`, then passes the same instance to `SummaryView`, to each `ExpenseRowView`, to the editor, and to settings. Every view holds a reference to the one object, so an edit made in any of them is immediately visible in all of them. This is the opposite of the value-type theme in the *State and Binding Basics* section, where each child received its own copy and a binding was the only way to write back.

That word *once* is now enforced for us. Built with Xcode 27, `@State` is the `State()` macro rather than a property wrapper, so a class stored in state initializes lazily, the first time SwiftUI instantiates the view. The distinction matters because SwiftUI views are lightweight values that the framework creates and discards constantly: before the macro, the `Budget.sample` expression ran on every one of those re-creations, allocating a fresh `Budget` on the heap only to throw it away, since `@State` keeps the first value it is handed. Now the expression runs a single time, when SwiftUI first instantiates `ContentView`, and the `Budget` it creates is owned for the life of the view. Mechanically, the macro keeps the initial value expression in a closure and hands it to a lazy storage type that holds the closure until SwiftUI first needs the value, then caches the result, and it adds an initialization accessor so the compiler treats the property as an ordinary stored value.

This matters more widely than the iOS 27 label suggests, because `@State` becomes the `State()` macro through the compiler we build with, not the operating system the app runs on. The lazy initialization arrives the moment a project moves to Xcode 27, whatever its deployment target, and that move is effectively mandatory: each year, around April, Apple requires apps and updates submitted to the App Store to be built with the latest SDK, so every team shipping updates adopts the new Xcode within months of its release. The improvement back-deploys with them. The

`@Observable` models we keep in `@State` are usable as far back as iOS 17, where the Observation framework began, and the macro initializes them once on each of those releases, not on iOS 27 alone.



New in iOS 27. The `State()` macro is what makes that once-per-lifetime initialization possible. Before Xcode 27, `@State` was a property wrapper that evaluated its default on every view `init`. Value types in `@State`, like the theme in the *State and Binding Basics* section, gain nothing visible from this; the win is for reference types such as our `Budget`.

That lazy initialization comes with one migration rule to carry over. Because the `State()` macro seeds a property from its inline default, assigning that same property again inside a custom `init` has no effect, and the inline value silently wins. This small view shows `0`, not `42`, when built with Xcode 27:

```
import SwiftUI
struct CounterView: View {
  @State private var counter = 0 // the macro seeds counter from this
  init() {
    counter = 42 // runs before the macro seeds counter, so it is ignored
  }
  var body: some View {
    Text("Counter: \(counter)") // shows 0, not 42
  }
}
```

Give a `@State` property its value in exactly one place: keep the inline default, or drop it and assign in `init`, but never both. The *Reusable Views and Modifiers* section of *Chapter 4* gathers the remaining Xcode 27 migration changes.

What makes this practical rather than wasteful is how the `@Observable` macro rewrites the class. Each stored property is backed by an observation registrar: a read routes through `registrar.access(self, keyPath:)` and a write through `registrar.withMutation(...)`, so tracking is keyed to individual properties rather than to the object as a whole. SwiftUI evaluates every view's `body` inside an observation scope, so it learns precisely which properties that `body` read: `SummaryView` read `total`, `remaining`, and `monthlyLimit`; an `ExpenseRowView` read its own expense's `title`, `amount`, and `category`. When a property is written, SwiftUI invalidates only the views that read that exact property. That is why dragging the limit slider re-evaluated `SummaryView` alone, and editing one expense re-evaluated that row and the summary but left the other rows untouched. The `Self._printChanges()` output made this visible, and `withObservationTracking()` is the same mechanism exposed directly.

Nesting deserves a closer look, because it is where the tracking feels almost too convenient. `Budget` holds an array of `Expense` objects, and both types are `@Observable`. When `SummaryView` reads `budget.total`, that computed property iterates the expenses and reads every `amount`, so the summary ends up tracking each amount as well as the array. An `ExpenseRowView`, by contrast, reads only its own expense. So editing a single expense's amount re-evaluates that one row, because it read that `amount`, and the summary, because it read the same `amount` through `total`, while every other row stays put. The dependency graph follows the actual reads, however deep they reach into nested objects, without any extra wiring from us.

Three property roles appear in the app, and the difference between them is the heart of data flow. A plain `let` (the budget in `SummaryView`, the expense in `ExpenseRowView`) is a read-only reference: the view observes the object but cannot replace or rebind it. `@State` (the budget in `ContentView`) marks the view as the owner that creates and keeps the object alive.

`@Bindable` (the expense in the editor, the budget in settings) is for a view that did not create the object but needs to write to it: it lets us project bindings such as `$expense.amount` and `$budget.monthlyLimit` from an observable object, the same way `$` works on `@State`. The parallel with `@Binding` from the *State and Binding Basics* section is worth naming: `@Binding` projects a writable reference into a value type owned by a parent's `@State`, while `@Bindable` does the same for a reference type, an `@Observable` object handed in from outside. Both give us the `$` projection, and which one we reach for depends only on whether the model is a value or a reference. We reached for `@ObservationIgnored` once, on `Expense.id`, because a constant identity never changes and there is nothing to gain from tracking it; the same applies to caches and other internal state that should not drive the UI.

The re-render mechanics are the ones from the *State and Binding Basics* section: a write invalidates the views that read the value, SwiftUI reruns their `body` and diffs the result, and view identity keeps each view's `@State` intact across the re-evaluation. What Observation changes is the granularity. SwiftUI records which properties each view's `body` reads, so when one of them changes, only the views that read it re-evaluate their `body`. That per-property precision is exactly what `Self._printChanges()` showed us, and it is the whole point of the framework.

The pattern that predates the Observation framework is still worth knowing. You will meet it across existing SwiftUI codebases, where a model class conforms to `ObservableObject`, marks its changing values with `@Published`, and is observed through `@StateObject` or `@ObservedObject`. Its limitation is granularity: any `@Published` change calls the object's `objectWillChange` publisher, which invalidates every view observing the object, regardless of which property changed or whether the view even read it. In a screen like ours, changing the monthly limit would have re-rendered every expense row. Apple built the Observation framework to

remove that over-rendering and, along the way, the boilerplate: there is no `@Published`, and the owning view uses plain `@State` rather than `@StateObject`. We will work with the `ObservableObject` model directly, alongside Combine, in the *Combine with Local Data* section , where it is the natural fit.

See also

- Observation framework documentation: https://x.swiftuicookbook.com/rajgl
- `State()` macro documentation: https://x.swiftuicookbook.com/c9pbw
- `Observable()` macro documentation: https://x.swiftuicookbook.com/leuag
- `Bindable` documentation: https://x.swiftuicookbook.com/8iht2
- `ObservationIgnored()` macro documentation: https://x.swiftuicookbook.com/37kuj
- `withObservationTracking(_:onChange:)` documentation: https://x.swiftuicookbook.com/wyotu
- *Migrating from the Observable Object protocol to the Observable macro* article: https ://x.swiftuicookbook.com/j4k6v

Combine with Local Data

The previous section showed the modern way to share a model: mark a class `@Observable` and let SwiftUI track it. You will still meet the pattern it replaced in nearly every existing SwiftUI codebase, though, so it is worth knowing well. Before the Observation framework, a shared model was a class conforming to `ObservableObject` with `@Published` properties, observed through `@StateObject` and `@ObservedObject`. That pattern is built on

Combine, Apple's framework for processing values over time, and Combine still earns its place for one job in particular: reacting to a stream of fast-changing input, like the text in a search field, without recomputing on every keystroke.

In this recipe, we build a movie browser: a searchable list of films backed by a view model, with a detail screen for each one. The search field is where Combine shines. Instead of filtering on every character, we debounce the input so the list updates only after the user pauses. The view model is an `ObservableObject` holding the query, the results, and a set of favorites in `@Published` properties; the list owns it with `@StateObject`, and the detail screen observes the same instance with `@ObservedObject`. Inside the model, a Combine pipeline turns the search text into filtered results, combines it with a sort order kept in a `CurrentValueSubject`, and delivers the output with `sink` and `assign`, holding the subscriptions in a set of `AnyCancellable`.

This is deliberately the counterweight to the *Observation Framework* section: the same goal, older tools. We use Combine here on local data to keep the focus on the operators themselves; its natural home is asynchronous work like networking, which we return to in the *Combine with Re mote Data* section of `Chapter 8`.

Getting ready

Create a new Swif tUI iOS app named **MovieBrowser** .

How to do it...

We will build the data first, then the view model that processes it, then the list that owns the model, and finally the detail screen that observes it. These are the steps:

1. Start with the data. Create `Movie.swift` in a `Models` folder. `Movie` is a simple value type, and a `catalog` of sample films gives the browser something to show. The `yearText` computed property formats the year as a plain string so it never picks up a thousands separator:

```
import Foundation
struct Movie: Identifiable {
  let id = UUID()
  let title: String
  let year: Int
  let genre: String
  let synopsis: String
  var yearText: String { String(year) }
}
extension Movie {
  static let catalog: [Movie] = [
    Movie(title: "Arrival", year: 2016, genre: "Sci-Fi",
          synopsis: "A linguist races to decode an alien languag
    Movie(title: "Coco", year: 2017, genre: "Animation",
          synopsis: "A boy journeys into the Land of the Dead.")
    Movie(title: "Dune", year: 2021, genre: "Sci-Fi",
          synopsis: "A noble heir confronts his destiny on Arrak
    Movie(title: "Inception", year: 2010, genre: "Thriller",
          synopsis: "A thief steals secrets from inside dreams.'
    Movie(title: "Parasite", year: 2019, genre: "Drama",
          synopsis: "A poor family schemes into a wealthy home.'
    // ... 10 more movies in the book's GitHub repository
  ]
}
```

2. Create `MovieStore.swift` in the same folder. This is the view model: a class conforming to `ObservableObject`, with the search `query`, the `results`, a `resultCount`, and the user's `favoriteIDs` exposed as `@Published` properties. A `SortOrder` enum and a `CurrentValueSubject` hold the current sort. The `filtered` helper does the searching and sorting, and the small methods change the sort and toggle favorites:

```
import Foundation
import Combine
enum SortOrder: String, CaseIterable, Identifiable {
  case title = "Title"
  case year = "Year"
  var id: String { rawValue }
}
class MovieStore: ObservableObject {
  @Published var query = ""
  @Published var results: [Movie] = []
  @Published var resultCount = 0
  @Published var favoriteIDs: Set<UUID> = []
  private let sortOrder = CurrentValueSubject<SortOrder, Never>(
  private let allMovies: [Movie]
  private var cancellables: Set<AnyCancellable> = []
  init(movies: [Movie] = Movie.catalog) {
    allMovies = movies
    results = MovieStore.filtered(movies, query: "", order: .tit
  }
  func setSort(_ order: SortOrder) {
    sortOrder.send(order)
  }
  func isFavorite(_ movie: Movie) -> Bool {
    favoriteIDs.contains(movie.id)
  }
  func toggleFavorite(_ movie: Movie) {
    if favoriteIDs.contains(movie.id) {
      favoriteIDs.remove(movie.id)
    } else {
      favoriteIDs.insert(movie.id)
    }
  }
  private static func filtered(_ movies: [Movie], query: String,
                               order: SortOrder) -> [Movie] {
    let trimmed = query.trimmingCharacters(in: .whitespaces)
    let matched = trimmed.isEmpty ? movies : movies.filter {
      $0.title.localizedCaseInsensitiveContains(trimmed)
    }
    switch order {
    case .title: return matched.sorted { $0.title < $1.title }
    case .year: return matched.sorted { $0.year > $1.year }
    }
  }
}
```

3. Now, make the model reactive. Inside `init`, after setting `results`, build a Combine pipeline on `$query`, the publisher behind the `@Published` search text. We `debounce` it so it emits only after the user stops typing for 0.3 seconds, `removeDuplicates` to ignore no-op changes, `combineLatest` it with the sort subject, `map` the pair into filtered, sorted results, and `sink` them into `results`. A first attempt often looks like this:

```
let debouncedQuery = $query
  .debounce(for: .seconds(0.3), scheduler: RunLoop.main)
  .removeDuplicates()
Publishers.CombineLatest(debouncedQuery, sortOrder)
  .map { [allMovies] query, order in
    MovieStore.filtered(allMovies, query: query, order: order)
  }
  .sink { [weak self] movies in
    self?.results = movies
  }
```

   There is nothing to run yet, because no view uses `MovieStore`, but the bug is already here, and Xcode even warns about it: **Result of call to 'sink(receiveValue:)' is unused**. Once we build the UI, the search would still do nothing. `sink` returns an `AnyCancellable`, and a subscription lives only as long as something holds that token. Here, we ignore the returned value, so it will be released the moment `init` finishes, canceling the subscription before a single keystroke arrives.

4. Keep the subscription alive by storing it in the `cancellables` set with `.store(in:)`. Here is the complete initializer, with a second small pipeline that keeps `resultCount` in step with `results` through `assign`:

```
init(movies: [Movie] = Movie.catalog) {
  allMovies = movies
  results = MovieStore.filtered(movies, query: "", order: .title
  let debouncedQuery = $query
    .debounce(for: .seconds(0.3), scheduler: RunLoop.main)
```

```
    .removeDuplicates()
  Publishers.CombineLatest(debouncedQuery, sortOrder)
    .map { [allMovies] query, order in
      MovieStore.filtered(allMovies, query: query, order: order)
    }
    .sink { [weak self] movies in
      self?.results = movies
    }
    .store(in: &cancellables)
  $results
    .map(\.count)
    .assign(to: &$resultCount)
}
```

5. Create `MovieRowView.swift` in a `Views` folder. Each row shows a film's title, genre, and year, with a star when it is a favorite. It takes plain values, not the store, because a row only displays:

```
import SwiftUI
struct MovieRowView: View {
  let movie: Movie
  let isFavorite: Bool
  var body: some View {
    HStack {
      VStack(alignment: .leading, spacing: 2) {
        Text(movie.title)
        Text("\(movie.genre) · \(movie.yearText)")
          .font(.caption)
          .foregroundStyle(.secondary)
      }
      Spacer()
      if isFavorite {
        Image(systemName: "star.fill")
          .foregroundStyle(.yellow)
      }
    }
  }
}
#Preview {
  List {
    MovieRowView(movie: Movie.catalog[0], isFavorite: true)
    MovieRowView(movie: Movie.catalog[1], isFavorite: false)
```

```
    }
}
```

6. Assemble the list. Open `ContentView.swift`. The view creates the store and owns it with `@StateObject`. The `.searchable` modifier, covered in the *Search and Filtering* section of *Chapter 2*, binds the search field to `$store.query`, which feeds the Combine pipeline; a toolbar menu changes the sort; and `ContentUnavailableView` covers the empty state. We show the rows directly for now and add navigation once the detail screen exists:

```
import SwiftUI
struct ContentView: View {
  @StateObject private var store = MovieStore()
  var body: some View {
    NavigationStack {
      List {
        Section("\(store.resultCount) movies") {
          ForEach(store.results) { movie in
            MovieRowView(movie: movie,
                         isFavorite: store.isFavorite(movie))
          }
        }
      }
      .navigationTitle("Movies")
      .searchable(text: $store.query, prompt: "Search titles")
      .toolbar {
        ToolbarItem(placement: .topBarTrailing) {
          Menu {
            ForEach(SortOrder.allCases) { order in
              Button(order.rawValue) {
                store.setSort(order)
              }
            }
          } label: {
            Image(systemName: "arrow.up.arrow.down")
          }
        }
      }
      .overlay {
        if store.results.isEmpty {
```

```
            ContentUnavailableView.search(text: store.query)
          }
        }
      }
    }
  }
  #Preview {
    ContentView()
  }
```

In the preview canvas, type in the search field and the list narrows after a brief pause; the sort menu reorders it. The favorites star will light up once we add the detail screen.

7. Create `MovieDetailView.swift` in the `Views` folder. The detail screen shows the film's information and a button to favorite it. Because it changes the shared model but does not own it, it takes the store with `@ObservedObject`, the counterpart to the list's `@StateObject`:

```
import SwiftUI
struct MovieDetailView: View {
  @ObservedObject var store: MovieStore
  let movie: Movie
  var body: some View {
    Form {
      Section {
        LabeledContent("Year", value: movie.yearText)
        LabeledContent("Genre", value: movie.genre)
      }
      Section("Synopsis") {
        Text(movie.synopsis)
      }
      Section {
        Button {
          store.toggleFavorite(movie)
        } label: {
          Label(store.isFavorite(movie) ? "Remove from Favorites
                  : "Add to Favorites",
                systemImage: store.isFavorite(movie) ? "star.sla
                  : "star")
        }
      }
```

```
    }
    .navigationTitle(movie.title)
    .navigationBarTitleDisplayMode(.inline)
  }
}
#Preview {
  NavigationStack {
    MovieDetailView(store: MovieStore(), movie: Movie.catalog[0]
  }
}
```

8. Back in `ContentView`, wrap each row in a `NavigationLink` so tapping a film opens its detail screen, handing the same store down:

```
ForEach(store.results) { movie in
  NavigationLink {
    MovieDetailView(store: store, movie: movie)
  } label: {
    MovieRowView(movie: movie,
                 isFavorite: store.isFavorite(movie))
  }
}
```

In the preview canvas, open a film and tap **Add to Favorites**. When we return to the list, the star appears on its row, because both screens observe the same store:

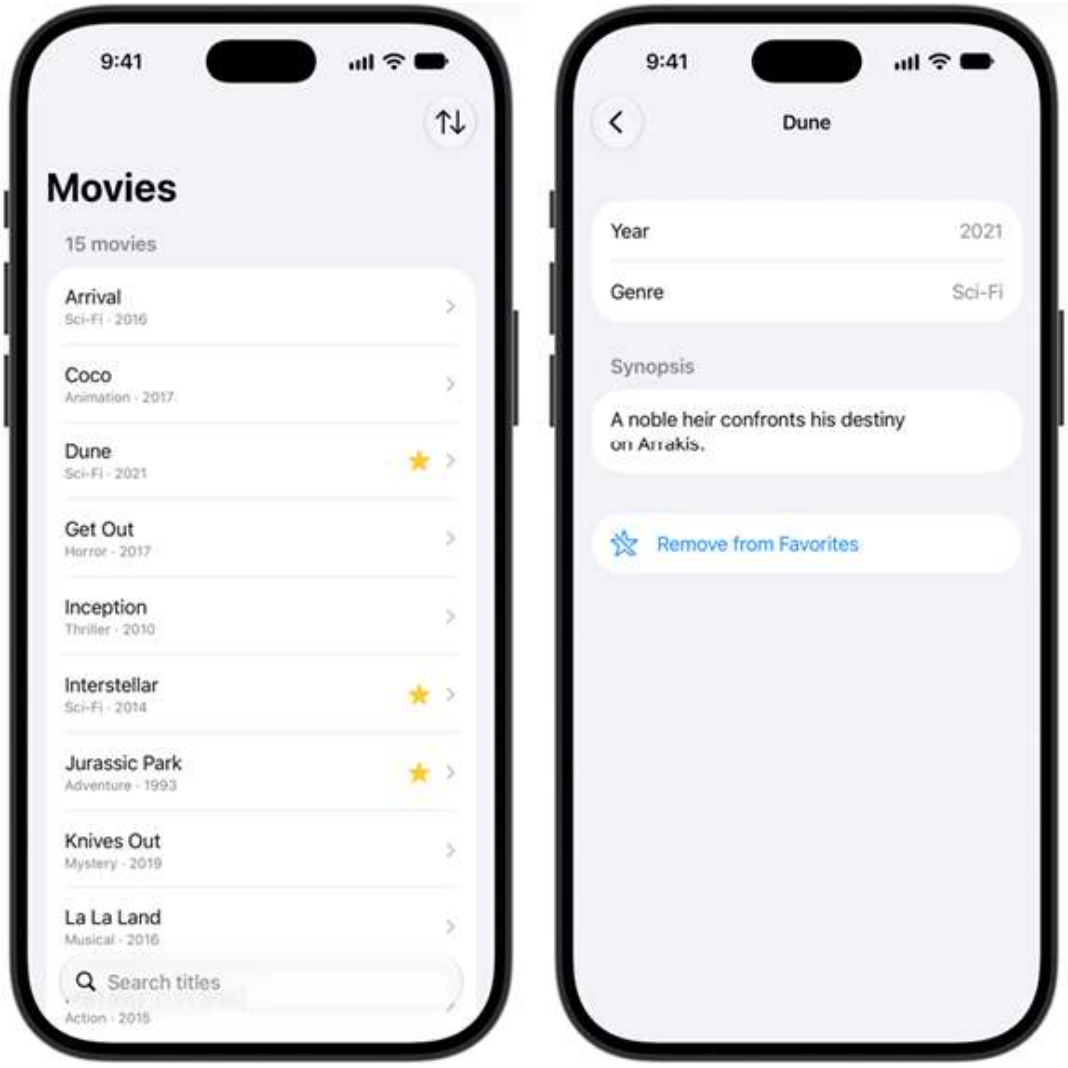


Figure 5.8: The movie list and a detail screen, both observing one MovieStore

9. Exercise the finished app: search for part of a title and watch the list narrow after a brief pause in typing, switch the sort between **Title** and **Year** , and favorite a few films. The search stays responsive because the pipeline debounces the input rather than filtering on every keystroke:

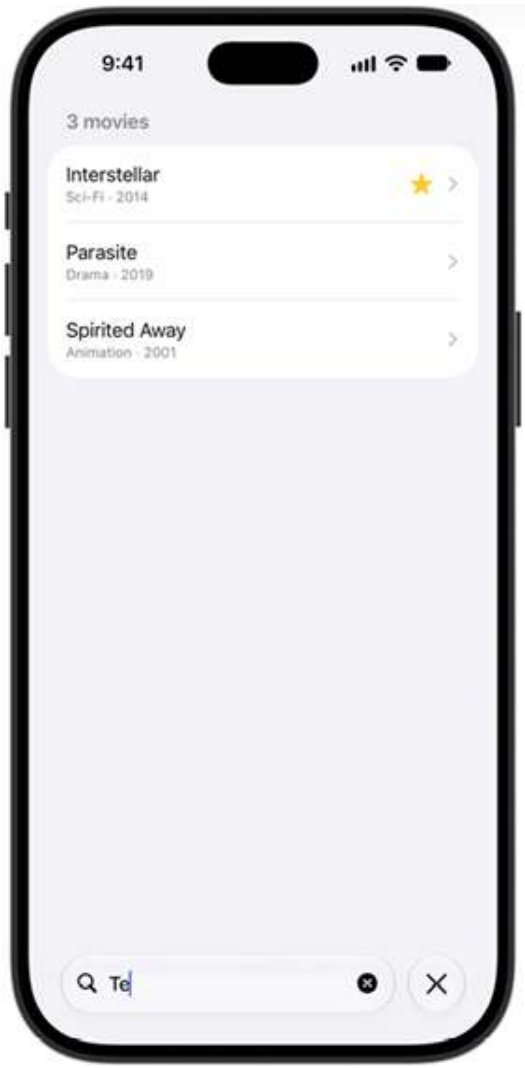


Figure 5.9: Searching narrows the list once typing pauses

How it works...

`MovieStore` is the kind of view model SwiftUI developers wrote for years: a class conforming to `ObservableObject`, with the values the UI cares about marked `@Published`. The `@Published` wrapper does two things. It announces changes through the object's `objectWillChange` publisher so that observing views re-render, and it exposes a Combine publisher for the property itself, reachable with the `$` prefix. That second part is what let us treat the search text as a stream.

Two property wrappers connect the model to the views, and the difference between them is about ownership. `ContentView` creates the store and marks it `@StateObject`, which tells SwiftUI to create the object once and keep it alive for the life of that view, even as the `body` reruns. `MovieDetailView` did not create the store; it was handed the same instance, so it marks it `@ObservedObject`, which observes without owning. The rule mirrors `@State` versus `@Binding` from the *State and Binding Basics* section: the creator owns, everyone else borrows. Using `@ObservedObject` for an object a view creates itself is a classic bug, because SwiftUI would recreate it on every re-render and its state would never stick.



A useful mantra is to create with `@StateObject` and receive with `@ObservedObject`. If a model ever seems to reset itself when its view updates, check whether a view that creates the object is holding it with `@ObservedObject` instead.

There is one important difference from the Observation framework, and it is the reason Apple introduced it. When any `@Published` property changes, `ObservableObject` fires a single `objectWillChange` event for the whole object, and every view observing that object re-renders, whether or not it read the property that changed. `@Observable` tracks reads per property, so it can re-render only the views that actually used the changed value. We watched that fine-grained behavior in the *Observation Framework* section with `Self._printChanges()`; the same experiment here would show every observer of `MovieStore` waking up on any change. For a screen this size it does not matter, but it is why new code should prefer `@Observable`.

The search itself is a Combine pipeline, and reading it from top to bottom tells the whole story. `$query` is the publisher behind the `@Published` search text. `debounce` holds each value for 0.3 seconds and forwards it only if nothing newer arrives, so typing quickly produces one update instead of

one per keystroke. The `scheduler` we hand it, `RunLoop.main`, decides where the value is delivered: on the main thread, which is where we must be to touch `results`. A pipeline that did heavier work off the main thread would hop back with `.receive(on:)` before that final assignment. `removeDuplicates` drops values identical to the last, `combineLatest` pairs the debounced query with the current sort so a change to either reruns the search, and `map` turns that pair into the filtered, sorted array. `sink` is the subscriber that receives each array and assigns it to `results`. A subscription lives only as long as something holds its `AnyCancellable`, which is why we `store` it in the `cancellables` set; when the store is deallocated, the set empties and the subscription ends.

The sort order is a `CurrentValueSubject`, one of Combine's two subjects, which are publishers we push values into by hand. We chose it because `combineLatest` needs an initial value from each of its inputs before it can emit, and a `CurrentValueSubject` always has a current value, here `.title`, that it hands to new subscribers immediately. Its sibling, `PassthroughSubject`, holds no value and simply relays each event to whoever is subscribed at that moment, which suits one-off actions like a button tap where there is nothing to remember between events. The rule of thumb: reach for `CurrentValueSubject` when there is a current state to read, and `PassthroughSubject` for fire-and-forget events.

The smaller pipeline, `$results.map(\.count).assign(to: &$resultCount)`, shows the other common subscriber: `assign` writes each value straight into a property, here another `@Published` one, with no closure and no separate cancellable to hold. As for when to reach for any of this today, ordinary shared state is simpler and more efficient with the `@Observable` model from the previous section. Combine earns its keep when you genuinely have a stream to transform over time, like debouncing input here, or merging and decoding network responses, which is exactly where we pick it up again in the *Combine with Remote Data* section of *Chapter 8*.

See also

- `ObservableObject` documentation: https://x.swiftuicookbook.com/yy5bv
- `Published` documentation: https://x.swiftuicookbook.com/208a9
- `StateObject` documentation: https://x.swiftuicookbook.com/jxr7x
- `ObservedObject` documentation: https://x.swiftuicookbook.com/ntgqh
- `CurrentValueSubject` documentation: https://x.swiftuicookbook.com/da28i
- `debounce(for:scheduler:options:)` documentation: https://x.swiftuicookbook.com/12h3v

6

App-Wide Settings and Accessibility

Every iOS app I have worked on reached a point where the same handful of values, the signed-in user, the current theme, a shared player or cart, needed to be readable from many places at once. In UIKit I passed references through initializers, built singletons reluctantly, and overrode `traitCollectionDidChange(_:)` whenever a system setting changed. SwiftUI takes a different bet: there is one place, the environment, that any view can read from, and any ancestor can write to, and the framework keeps it consistent as state and system settings change.

This chapter is about both halves of that pipe. The first three recipes put things into the environment and read them back: the iOS 17 `@Observable` macro for a shared model, the older `ObservableObject` for legacy codebases, and the values like `colorScheme` and `dynamicTypeSize` that arrive there from the system without anyone setting them. The fourth recipe turns to accessibility, the surface where adapting to the user matters most, where VoiceOver labels, hints, custom rotors, and focus state turn a view that reads correctly into one that everyone can navigate.

We start with a music player that holds the library of songs, the current song, and the playback state in one shared model. The first recipe builds it with `@Observable`, injects it with `.environment(_:)`, reads it with `@Environment(MusicPlayer.self)`, and lets a slider edit the volume through `@Bindable`. The second recipe rebuilds the same player on `ObservableObject`, `@Published`, `@StateObject`, `@EnvironmentObject`, and `.environmentObject(_:)`, the toolkit any legacy codebase still lives with, and `Self._printChanges()` shows side by side how much more selectively the Observation framework re-renders than `ObservableObject`. The third recipe pivots to the values the system already shares: a settings dashboard reads `colorScheme`, `dynamicTypeSize`, the two size classes, `locale`, and the `dismiss` and `openURL` actions, then defines a custom environment value of its own with the `@Entry` macro introduced with iOS 18.

The last recipe is an accessible recipe browser, the chapter's deliberate close on building apps that work for everyone. On each row we compose `.accessibilityLabel`, `.accessibilityHint`, `.accessibilityValue`, and `.accessibilityElement(children:)`. We expose custom actions through `.accessibilityAction(named:)` alongside the visual `.swipeActions`. We mark headings with `.accessibilityAddTraits(.isHeader)` and register two custom rotors with `.accessibilityRotor`. When an action fails, `@AccessibilityFocusState` brings VoiceOver focus to an error message. We verify each piece with Xcode's Accessibility Inspector, the right tool for testing accessibility on a simulated device.

By the end of this chapter, you should be comfortable passing an `@Observable` model via the environment, recognizing the older `ObservableObject` pattern when you meet it in legacy code, reading the system's environment values to adapt a view to the user's

preferences, defining your own values with `@Entry` when a setting should flow down the view hierarchy, and laying down the accessibility surface that makes an app usable by everyone, not just the audiences a designer happens to have in mind.

The recipes we'll cover in this chapter are as follows:

- Sharing Data with the Environment Using Observation
- Sharing Data with the Environment Using ObservableObject
- Reading and Defining Environment Values
- Accessibility APIs

Technical requirements

The code in this chapter is based on Xcode 27.0 and iOS 27.0. You can download and install the latest version of Xcode from the App Store. All the code examples for this chapter can be found on GitHub at https://x.swiftuicookbook.com/ch06 .

Sharing Data with the Environment Using Observation

Some state belongs to a single screen, but plenty of it belongs to the whole app: the signed-in user, the current theme, a shopping cart, or a download queue. The awkward part is delivery: getting that shared model down to every view that needs it. When a model is needed three or four levels deep in the view hierarchy, the usual fix is to pass it into every view initializer along the way, threading it through views that do not use it just so a distant child can reach it. That

plumbing is tedious to write and easy to break, and it couples views to data they never touch.

SwiftUI's environment solves this by letting us place a value at the top of the hierarchy and read it from any descendant, with nothing passed in between. In this recipe, we build a music player app backed by one shared model: a scrollable library of tracks, a mini-player bar pinned to the bottom, a now-playing sheet, and a pair of status indicators in the navigation bar. Each of those surfaces reads and edits the same playback state, so we inject one `@Observable` object with `.environment(_:)`, read it anywhere with `@Environment(MusicPlayer.self)`, and turn it into editable bindings with `@Bindable`. Because every surface looks at the same object, a change made in one is reflected in all of them with no synchronization code of our own.

This recipe uses the Observation framework explained in the *Observation Framework* section of *Chapter 5*. If `@Observable` and `@Bindable` are not yet familiar, it is worth reading that section first; here we focus on the environment as the way to deliver an observable model across an ap p.

Getting ready

Create a new SwiftUI iOS app named **MusicPlaye r** .

How to do it...

We will build the model first, inject it once at the root, then add the screens that read it one at a time, finishing with the two navigation-bar indicators that prove every surface stays in sync. These are the steps:

1. Start with the data. Create a new file `Track.swift` in a `Models` folder. A `Track` is a simple value type; the `artwork` is an SF Symbol name and `tint` a color, which lets us build a varied library without bundling image assets. The `library` static array gives us something to show:

```
import SwiftUI
struct Track: Identifiable {
  let id = UUID()
  let title: String
  let artist: String
  let artwork: String
  let tint: Color
}
extension Track {
  static let library: [Track] = [
    Track(title: "Midnight City Lights", artist: "Neon Aver
          artwork: "building.2.fill", tint: .pink),
    Track(title: "Ocean Drive", artist: "The Sunsets",
          artwork: "water.waves", tint: .teal),
    Track(title: "Mountain Echo", artist: "Pinecrest",
          artwork: "mountain.2.fill", tint: .green),
    Track(title: "Desert Bloom", artist: "Mirage",
          artwork: "sun.max.fill", tint: .orange),
    Track(title: "Electric Pulse", artist: "Voltage",
          artwork: "bolt.fill", tint: .yellow),
    Track(title: "Velvet Sky", artist: "Aurora Lane",
          artwork: "moon.stars.fill", tint: .indigo),
    Track(title: "Paper Planes", artist: "Origami",
          artwork: "paperplane.fill", tint: .blue),
    Track(title: "Wildflower", artist: "Meadowlark",
          artwork: "leaf.fill", tint: .mint)
  ]
}
```

2. Create `MusicPlayer.swift` in the same folder. This is the shared model: an `@Observable` class that holds the track list, what is

playing, whether playback is active, and the volume, with small methods to start a track, toggle play and pause, and skip. One instance of this class will back the entire app. The `preview` helper in the extension instantiates a player with a track already playing, which we reuse in the previews further down:

```
import SwiftUI
@Observable
class MusicPlayer {
  var tracks: [Track] = Track.library
  var currentTrack: Track?
  var isPlaying = false
  var volume = 0.5
  func play(_ track: Track) {
    currentTrack = track
    isPlaying = true
  }
  func togglePlayPause() {
    guard currentTrack != nil else { return }
    isPlaying.toggle()
  }
  func playNext() {
    guard let current = currentTrack,
          let index = tracks.firstIndex(where: { $0.id == c
    else { return }
    play(tracks[(index + 1) % tracks.count])
  }
  func playPrevious() {
    guard let current = currentTrack,
          let index = tracks.firstIndex(where: { $0.id == c
    else { return }
    play(tracks[(index - 1 + tracks.count) % tracks.count])
  }
}
extension MusicPlayer {
  static var preview: MusicPlayer {
    let player = MusicPlayer()
    player.play(Track.library[0])
    return player
```

```
  }
}
```

3. Inject the model once, at the top of the app. Open `MusicPlayerApp.swift` and create the player with `@State` so the app owns it for its lifetime, then hand it to the environment with `.environment(_:)`. From here on, every view inside `ContentView` can reach this exact instance without passing it through a single initializer:

```
import SwiftUI
@main
struct MusicPlayerApp: App {
  @State private var player = MusicPlayer()
  var body: some Scene {
    WindowGroup {
      ContentView()
        .environment(player)
    }
  }
}
```

4. Build the library list, starting with the row it shows. Create `TrackRowView.swift` in a `Views` folder. The row reads the shared player so it can highlight the track that is currently playing and show an animated `waveform` next to it:

```
import SwiftUI
struct TrackRowView: View {
  let track: Track
  @Environment(MusicPlayer.self) private var player
  private var isCurrent: Bool { player.currentTrack?.id ==
  var body: some View {
    HStack(spacing: 12) {
      Image(systemName: track.artwork)
```

```
          .font(.title2)
          .foregroundStyle(.white)
          .frame(width: 48, height: 48)
          .background(track.tint.gradient, in: .rect(cornerRa
        VStack(alignment: .leading, spacing: 2) {
          Text(track.title)
            .font(.body.weight(.medium))
            .foregroundStyle(isCurrent
                             ? AnyShapeStyle(.tint) : AnyShap
          Text(track.artist)
            .font(.subheadline)
            .foregroundStyle(.secondary)
        }
        Spacer()
        if isCurrent && player.isPlaying {
          Image(systemName: "waveform")
            .foregroundStyle(.tint)
            .symbolEffect(.variableColor.iterative)
        }
      }
    }
  }
  #Preview {
    @Previewable @State var player = MusicPlayer.preview
    List {
      TrackRowView(track: Track.library[0])
      TrackRowView(track: Track.library[1])
    }
    .environment(player)
  }
```

The row only reads from the player, so a plain property is enough; it never writes back.

5. Now, build the screen itself. Open `ContentView.swift` and replace the template. The view reads the shared player with `@Environment(MusicPlayer.self)`, lists the tracks using the row, and starts one playing on tap. Notice that nothing was passed in: the property wrapper looks the player up by its type:

```
import SwiftUI
struct ContentView: View {
  @Environment(MusicPlayer.self) private var player
  var body: some View {
    NavigationStack {
      List(player.tracks) { track in
        TrackRowView(track: track)
          .contentShape(.rect)
          .onTapGesture { player.play(track) }
      }
      .navigationTitle("Library")
    }
  }
}
#Preview {
  ContentView()
    .environment(MusicPlayer())
}
```

Each `#Preview` injects its own player, because a view that reads the environment needs something there to read. With the preview canvas in live mode, tap a track: its row turns the accent color and the waveform animates. The preview canvas is enough for the next few steps, so there is no need to run the app in Device Hub until we exercise the finished app. `ContentView` and `TrackRowView` are two separate views reading the same object, neither of them passed it in.

6. Add the mini-player bar. Create `MiniPlayerView.swift` in the `Views` folder. This bar is a sibling of the list, not a child, yet it reads the same player with no value handed to it. It shows the current track, toggles playback, and signals that it was tapped through a binding so the parent can present the now-playing sheet:

```
import SwiftUI
struct MiniPlayerView: View {
  @Environment(MusicPlayer.self) private var player
  @Binding var showNowPlaying: Bool
  var body: some View {
    if let track = player.currentTrack {
      HStack(spacing: 12) {
        Image(systemName: track.artwork)
          .foregroundStyle(.white)
          .frame(width: 40, height: 40)
          .background(track.tint.gradient, in: .rect(corner
        VStack(alignment: .leading) {
          Text(track.title)
            .font(.subheadline.weight(.medium))
            .lineLimit(1)
          Text(track.artist)
            .font(.caption)
            .foregroundStyle(.secondary)
            .lineLimit(1)
        }
        Spacer()
        Button {
          player.togglePlayPause()
        } label: {
          Image(systemName: player.isPlaying ? "pause.fill"
            .font(.title2)
        }
        .buttonStyle(.plain)
      }
      .padding(.horizontal)
      .padding(.vertical, 8)
      .background(.bar)
      .contentShape(.rect)
      .onTapGesture { showNowPlaying = true }
    }
  }
}
#Preview {
  @Previewable @State var player = MusicPlayer.preview
  MiniPlayerView(showNowPlaying: .constant(false))
    .environment(player)
}
```

The bar uses `.background(.bar)`, the frosted material the system uses behind bars, so the list blurs as it scrolls underneath.

7. Now, show the bar. Back in `ContentView`, add the state that tracks whether the sheet is open, and pin the bar to the bottom edge with `.safeAreaInset`. The bar only appears once something is playing:

```
struct ContentView: View {
  @Environment(MusicPlayer.self) private var player
  @State private var showNowPlaying = false
  var body: some View {
    NavigationStack {
      List(player.tracks) { track in
        // ... existing TrackRowView row
      }
      .navigationTitle("Library")
      .safeAreaInset(edge: .bottom) {
        if player.currentTrack != nil {
          MiniPlayerView(showNowPlaying: $showNowPlaying)
        }
      }
    }
  }
}
```

Tap a track in the preview canvas again. The bar slides up showing that track, and its play and pause button toggles the same `isPlaying` the list row reads, so the row's waveform starts and stops in step with the bar. Neither view knows about the other. They only share the model.

8. Add the now-playing screen. Create `NowPlayingView.swift` in the `Views` folder. This is where the user edits the model rather than

just reading it, so the volume slider needs a writable binding. As we saw in the *Observation Framework* section, a view that did not create an observable object but needs to write to it reaches for `@Bindable`, which gives back the `$` projection so `$player.volume` writes straight through to the shared object. The screen also carries transport controls and is presented as a half-height sheet, so the library stays visible behind it:

```
import SwiftUI
struct NowPlayingView: View {
  @Environment(MusicPlayer.self) private var player
  var body: some View {
    @Bindable var player = player
    VStack(spacing: 28) {
      if let track = player.currentTrack {
        HStack(spacing: 16) {
          Image(systemName: track.artwork)
            .font(.largeTitle)
            .foregroundStyle(.white)
            .frame(width: 80, height: 80)
            .background(track.tint.gradient, in: .rect(corr
          VStack(alignment: .leading, spacing: 4) {
            Text(track.title)
              .font(.title3.bold())
            Text(track.artist)
              .font(.subheadline)
              .foregroundStyle(.secondary)
          }
          Spacer()
        }
      }
      HStack(spacing: 48) {
        Button { player.playPrevious() } label: {
          Image(systemName: "backward.fill").font(.title2)
        }
        Button { player.togglePlayPause() } label: {
          Image(systemName: player.isPlaying
                ? "pause.circle.fill" : "play.circle.fill")
            .font(.system(size: 56))
```

```
        }
        Button { player.playNext() } label: {
          Image(systemName: "forward.fill").font(.title2)
        }
      }
      .buttonStyle(.plain)
      VStack {
        Text("\(Int(player.volume * 100))%")
          .font(.caption.weight(.medium))
          .foregroundStyle(.secondary)
          .monospacedDigit()
          .contentTransition(.numericText())
        HStack {
          Image(systemName: "speaker.fill")
            .foregroundStyle(.secondary)
          Slider(value: $player.volume, in: 0...1)
          Image(systemName: "speaker.wave.3.fill")
            .foregroundStyle(.secondary)
        }
      }
      Spacer(minLength: 0)
    }
    .padding()
    .presentationDetents([.medium, .large])
    .presentationDragIndicator(.visible)
  }
}
#Preview {
  @Previewable @State var player = MusicPlayer.preview
  NowPlayingView()
    .environment(player)
}
```

The transport buttons call methods on the player, and the slider binds to `$player.volume`, so every control writes to the one shared object. The first line inside the `body` is the one to remember. `@Environment(MusicPlayer.self)` hands back a read-only reference with no `$` projection, so `Slider(value: $player.volume)` does not compile until `@Bindable var player = player` shadows it

with a bindable one. Forgetting that line is the usual reason a binding to a model in the environment cannot be found. Reusing the same name is deliberate, and is the form Apple's own documentation uses, so the rest of the `body` reads exactly as it would without it.

9. Now, present it. Attach a `.sheet` to the `NavigationStack` in `ContentView`, driven by the `showNowPlaying` state the bar already flips:

```
NavigationStack {
  // ... List with safeAreaInset bar
}
.sheet(isPresented: $showNowPlaying) {
  NowPlayingView()
}
```

   Back in the preview canvas, start a track and tap the bar. The sheet rises to roughly half the screen, leaving the library visible behind it. Tap the previous and next buttons and watch the highlighted row change underneath the sheet, and drag the volume slider to see the percentage update. The sheet receives the same player automatically; SwiftUI passes the environment into presented sheets, so we did not re-inject anything.

10. Now, add the first navigation-bar indicator. We want a control at the top of the list that both shows whether music is playing and lets the user toggle it, even while the sheet is covering the lower half of the screen. Attach a `.toolbar` to the `List` with a trailing item: a `Button` whose icon reflects `player.isPlaying`, animating a `waveform` while playing and showing a `pause.fill` that breathes when paused to invite a tap. We set `.foregroundStyle(.tint)`

because a toolbar button's content otherwise renders in the default label color rather than the accent:

```
        .navigationTitle("Library")
        .toolbar {
          ToolbarItem(placement: .topBarTrailing) {
            if player.currentTrack != nil {
              Button {
                player.togglePlayPause()
              } label: {
                Image(systemName: player.isPlaying ? "wavefor
                  .foregroundStyle(.tint)
                  .symbolEffect(.variableColor.iterative,
                                isActive: player.isPlaying)
                  .symbolEffect(.breathe, isActive: !player.i
              }
            }
          }
        }
```

In the preview canvas, open the now-playing sheet. Tapping pause in the sheet flips the toolbar icon to a breathing `pause.fill`, and tapping that toolbar icon resumes playback, which the sheet's controls and the list row reflect at once. The toolbar button and the sheet's transport controls are two views calling `togglePlayPause()` on the same model.

11. Add the second indicator: a volume readout that pops up while the user drags the slider, which makes the data sharing visible. It takes three small additions to `ContentView`, shown together here: state for whether the indicator is showing plus a `volumeSymbol` that mirrors the level, a leading toolbar item (the `.fixedSize()` keeps the percentage from being clipped), and an `.onChange(of:)` that

reveals the indicator on any volume change and hides it 1.2 seconds after the last one:

```
struct ContentView: View {
  @Environment(MusicPlayer.self) private var player
  @State private var showNowPlaying = false
  @State private var showVolumeIndicator = false
  @State private var volumeHideTask: Task<Void, Never>?
  private var volumeSymbol: String {
    switch player.volume {
    case 0: "speaker.slash.fill"
    case ..<0.34: "speaker.wave.1.fill"
    case ..<0.67: "speaker.wave.2.fill"
    default: "speaker.wave.3.fill"
    }
  }
  var body: some View {
    NavigationStack {
      List(player.tracks) { track in
        // ... existing TrackRowView row
      }
      .navigationTitle("Library")
      .toolbar {
        ToolbarItem(placement: .topBarLeading) {
          if showVolumeIndicator {
            HStack(spacing: 4) {
              Image(systemName: volumeSymbol)
              Text("\(Int(player.volume * 100))%")
                .monospacedDigit()
                .contentTransition(.numericText())
            }
            .font(.subheadline.weight(.medium))
            .foregroundStyle(.tint)
            .lineLimit(1)
            .fixedSize()
          }
        }
        // ... play and pause item from the previous step
      }
      // ... safeAreaInset bar from step 7
    }
    .sheet(isPresented: $showNowPlaying) {
```

```
            NowPlayingView()
        }
        .onChange(of: player.volume) {
            volumeHideTask?.cancel()
            withAnimation { showVolumeIndicator = true }
            volumeHideTask = Task {
                try? await Task.sleep(for: .seconds(1.2))
                guard !Task.isCancelled else { return }
                withAnimation { showVolumeIndicator = false }
            }
        }
    }
}
```

For the complete `ContentView`, see the project in the book's GitHub repository. In the preview canvas, open the sheet and drag the volume slider. Because the sheet only covers the lower half, the navigation bar stays visible, and the volume indicator pops in there with the level icon and a live percentage that tracks the slider. The slider sets `player.volume` in the sheet, and `ContentView`, a view in a different part of the hierarchy, reacts to that same property through its `.onChange`. That is the environment doing its job.

12. Now, run the app in Device Hub and exercise everything together: tap tracks in the list, play and pause from the mini bar, the sheet, and the toolbar, skip with the sheet's transport controls, and drag the volume. Every surface stays consistent because they all read and write one shared object. The finished library with its mini-player bar and the now-playing sheet look like the following:



Figure 6.1: The library with the mini-player bar and the now-playin g half-sheet, all backed by one shared model

How it works...

The whole recipe rests on one move: `MusicPlayerApp` creates the player with `@State` and hands it to `.environment(player)` at the root of the scene. Placing an object in the environment makes it available to every view below, and reading it is a matter of declaring `@Environment(MusicPlayer.self)`, which looks the object up by its type.

`ContentView`, `TrackRowView`, `MiniPlayerView`, `NowPlayingView`, and both toolbar items each declare that one line and receive the same instance, none of them taking the player as an initializer parameter. That is the difference from the *Observation Framework* section, where we passed the budget explicitly into each child: the environment is for state that is genuinely app-wide, where threading it through every initializer would be noise.

Ownership and lifetime stay with the root. `@State private var player` in the app makes `MusicPlayerApp` the owner that creates the player and keeps it alive, exactly as a view owns a value it creates with `@State`, covered in detail in the *State and Binding Basics* section of *Chapter 5*. The descendants only read; they hold no claim on the object's lifetime. Reading by type does come with one rule worth remembering: the object must actually be in the environment, or the lookup traps at runtime. That is why each `#Preview` injects its own player, and it is the same requirement the older `@EnvironmentObject` carries, which we return to in the *Sharing Data with the Environment Using ObservableObject* section.

Reading is enough for most of the screens, but `NowPlayingView` needs to write, and that is the job of `@Bindable`. Writing `@Bindable var player = player` at the top of the `body` re-wraps the environment object so we can project bindings from it, which is what lets `Slider(value: $player.volume)` edit the shared value directly. We covered the difference between a read-only reference and a `@Bindable` one in the *Observation Framework* section; here the same distinction decides which views can mutate the player and which only display it.

The synchronization that makes the app feel alive is not something we wrote; it falls out of every surface reading one `@Observable` object.

When the sheet's slider sets `player.volume`, SwiftUI invalidates the views that read `volume`, so the percentage in the sheet, the percentage in the toolbar, and the level icon all update together. When any control calls `togglePlayPause()`, the mini-bar button, the toolbar's waveform, and the playing row's waveform all flip, because each reads `isPlaying`. The two navigation-bar indicators are the clearest demonstration: they sit in `ContentView`, the sheet that drives them sits in a separate part of the hierarchy, and the only thing connecting them is the shared model. The `.onChange(of: player.volume)` on `ContentView` is what gives the volume indicator its transient pop-in, reacting to a value a completely different view edited. Because tracking is per property, this stays efficient: dragging the volume does not re-render the eight library rows, since none of them read `volume`. That fine-grained behavior is the heart of the Observation framework, explored in the *Observation Framework* section.

A few smaller pieces round out the screen. Presenting `NowPlayingView` with `.presentationDetents ([.medium, .large])` is what keeps the library visible behind the sheet at the `.medium` detent, so skipping tracks visibly updates the highlighted row underneath. Drag the sheet up to `.large` and it covers the screen, hiding the list until the user pulls it back down. SwiftUI forwards the environment into the sheet automatically either way, so the presented view reads the same player without re-injection. The `.symbolEffect(.breathe)` on the paused toolbar icon and `.variableColor` on the playing waveform are status cues driven by `isPlaying`, and the `.contentTransition(.numericText())` animates the percentage as it changes. None of these touch the data flow; th ey simply make the shared state legible at a glance.

See also

- `environment(_:)` documentation: https://x.swiftuicookbook.com/vowgy
- `Environment` documentation: https://x.swiftuicookbook.com/wpufx
- `Bindable` documentation: https://x.swiftuicookbook.com/8iht2
- `presentationDetents(_:)` documentation: https://x.swiftuicookbook.com/v2531
- `symbolEffect(_:options:isActive:)` documentation: https://x.swiftuicookbook.com/6subc
- Discover Observation in Swif tUI (WWDC 2023): https://x.swiftuicookbook.com/5lt9g

Sharing Data with the Environment Using ObservableObject

The previous section shared the player with the Observation framework, the modern default on iOS 17 and later. For years before that, the way to share a reference-type model in SwiftUI was a class conforming to `ObservableObject`, and you will still meet it constantly: in code written before iOS 17, in libraries that target older systems, and in any project that has to run on iOS 16 or earlier. Knowing it is not optional, and moving between the two is something I find myself doing on most codebases I join.

In this recipe, we rebuild the same **MusicPlayer** app, the library, the mini-player bar, the now-playing sheet, and the two navigation-bar

indicators, on top of `ObservableObject` instead of `@Observable`. The interface does not change at all; what changes is the model and the property wrappers that move it through the app. We mark the model `ObservableObject` with `@Published` properties, own it with `@StateObject`, inject it with `.environmentObject`, and read it with `@EnvironmentObject`. Building the two versions side by side is the fastest way to see what the Observation framework streamlined.

This recipe starts from the app built in the *Sharing Data with the Environment Using Observation* section, so it helps to have that one finished first. The `ObservableObject` and `@Published` basics come from the *Combine with Local Data* section of *`Chapter 5`*; here we focus on sharing such a model through the environment.

Getting ready

This recipe rebuilds the **MusicPlayer** app from the previous section using `ObservableObject`. Start from a copy of that project. The `Track` model is unchanged, and so is the `body` of every view, so we change only three things: the model, the app's injection, and the declaration each view uses to read the player.

How to do it...

We change the model first, then the injection at the app root, and finally how each view reads the player. These are the steps:

1. Convert the model. Open `MusicPlayer.swift` and replace the `@Observable` class with one that conforms to `ObservableObject`, giving each stored property the `@Published` wrapper. The file now

imports `Combine`, where `ObservableObject` and `@Published` live. The methods and the `preview` helper are unchanged:

```
import Combine
class MusicPlayer: ObservableObject {
  @Published var tracks: [Track] = Track.library
  @Published var currentTrack: Track?
  @Published var isPlaying = false
  @Published var volume = 0.5
  func play(_ track: Track) {
    currentTrack = track
    isPlaying = true
  }
  func togglePlayPause() {
    guard currentTrack != nil else { return }
    isPlaying.toggle()
  }
  func playNext() {
    guard let current = currentTrack,
          let index = tracks.firstIndex(where: { $0.id == c
    else { return }
    play(tracks[(index + 1) % tracks.count])
  }
  func playPrevious() {
    guard let current = currentTrack,
          let index = tracks.firstIndex(where: { $0.id == c
    else { return }
    play(tracks[(index - 1 + tracks.count) % tracks.count])
  }
}
extension MusicPlayer {
  static var preview: MusicPlayer {
    let player = MusicPlayer()
    player.play(Track.library[0])
    return player
  }
}
```

Without `import Combine` the class will not compile. The `objectWillChange` publisher that `ObservableObject` requires is synthesized from the `@Published` properties only when Combine is in scope.

2. Update the app root. In `MusicPlayerApp.swift`, own the player with `@StateObject` instead of `@State`, and inject it with `.environmentObject` instead of `.environment`:

```
import SwiftUI
@main
struct MusicPlayerApp: App {
  @StateObject private var player = MusicPlayer()
  var body: some Scene {
    WindowGroup {
      ContentView()
        .environmentObject(player)
    }
  }
}
```

`@StateObject` is the `ObservableObject` equivalent of `@State` for a model the view creates and keeps alive for its lifetime.

3. Change how the views read the player. Every view that read the model with `@Environment(MusicPlayer.self)` now uses `@EnvironmentObject`. In `TrackRowView`, for example:

```
struct TrackRowView: View {
  let track: Track
  @EnvironmentObject var player: MusicPlayer
  // ... the rest of the view is unchanged
}
```

Make the same one-line change in `ContentView` and `MiniPlayerView`. Nothing else in those views changes.

4. Simplify the now-playing screen. In the previous section, `NowPlayingView` made its volume slider editable by adding `@Bindable var player = player` at the top of its `body`. `@EnvironmentObject` already vends bindings through its projection, so that line is no longer needed; `$player.volume` keeps working without it:

```
struct NowPlayingView: View {
  @EnvironmentObject var player: MusicPlayer
  var body: some View {
    VStack(spacing: 28) {
      // ... rest of the view, with Slider(value: $player.v
    }
    // ... padding and presentation modifiers
  }
}
```

   That projection is the same one `@StateObject` and `@ObservedObject` expose; `@EnvironmentObject` is that wrapper for a model arriving through the environment.

5. In the preview canvas, the app behaves exactly as it did in the previous section: the same library, mini-player bar, half-sheet, and navigation-bar indicators, all driven by one shared object. The difference is now entirely internal, in how much SwiftUI redraws when a value changes, and we can make it visible.
6. Watch the over-rendering for ourselves, with the same `Self._printChanges()` helper used in the *Observation Framework*

section of *Chapter 5*. Temporarily add it to the top of `TrackRowView`'s `body`:

```
var body: some View {
  let _ = Self._printChanges()
  // ... existing row content
}
```

Run the app in Device Hub for this one, rather than the preview canvas, which re-evaluates views for its own reasons and would muddy the count we are about to read. Start a track and tap the play/pause button. That single tap toggles `isPlaying`, and the console reports all eight rows re-evaluating at once, even though only the playing row has any reason to, because each `@Published` write invalidates every view observing the object. In the Observation version from the previous section, the same tap re-evaluates only the playing row. Remove the `Self._printChanges()` line before moving on.



Figure 6.2: A single play/pause tap re -evaluates every library row, not only the one that is playing

How it works...

The shape of the app is unchanged from the *Sharing Data with the Environment Using Observation* section: one reference-type model, created once and shared through the environment, read by every screen. Only the three property wrappers differ, and they map one-to-one onto the modern ones. `@StateObject` owns the model in the app, the counterpart of `@State`; `.environmentObject` injects it, the counterpart of `.environment`; and `@EnvironmentObject` reads it in each view, the counterpart of `@Environment(MusicPlayer.self)`. The model itself is a class conforming to `ObservableObject` with each stored property marked `@Published`.

What genuinely differs is how a change propagates. Each `@Published` property funnels every write into the object's single `objectWillChange` publisher, and SwiftUI invalidates every view observing that object through `@EnvironmentObject`, regardless of which property changed or whether the view even read it. In our app that means a single play/pause tap re-renders all eight library rows at once, even though only the playing row reads `isPlaying`, exactly as the `Self._printChanges()` output showed; dragging the volume, which no row reads at all, re-renders the same eight, where the Observation version re-rendered none. That coarse, whole-object invalidation is exactly what the Observation framework set out to remove with the per-property tracking we saw in the *Observation Framework* section of *Chapter 5*; the visible result is identical, but the work SwiftUI does underneath is not.

The bindings are the other difference worth naming. `@EnvironmentObject` exposes a projection, so `$player.volume` produces a

`Binding` directly, which is why the volume slider works and why we could delete the `@Bindable` line that the Observation version needed. `@Bindable` exists precisely to give an `@Observable` object that same projection; with `ObservableObject` the wrapper already provides it, the same way `@StateObject` and `@ObservedObject` do.

Two requirements come with this approach. The model file must `import Combine`, because the `objectWillChange` publisher is synthesized there; without it the type silently fails to conform. And `@EnvironmentObject` traps at runtime if no matching object was injected by an ancestor, the same hazard `@Environment(MusicPlayer.self)` carries, which is why every preview injects its own player with `.environmentObject`. The distinction between `@StateObject` for the owner and `@ObservedObject` for a view that receives an object it does not own is covered in the *Combine with Local Data* section of *Chapter 5*. One trap to avoid is reaching for plain `@State` to own an `ObservableObject`: `@State` keeps the reference but does not subscribe to `objectWillChange`, so a view reading the object would not refresh when a `@Published` property changes. `@State` is for value types and, since iOS 17, for `@Observable` objects; an `ObservableObject` needs `@StateObject`.

Which to reach for is mostly a question of deployment target. For iOS 16 and earlier, `ObservableObject` is the only option, and it was the standard approach for years before the Observation framework, so it still turns up across older codebases. Coming across it in a project that began a few years ago but has since dropped support for iOS 16 is a good opportunity to migrate that model to `@Observable`. For iOS 17 and later, the Observation framework is the better default: finer-grained updates and less ceremony, with no `@Published`, no Combine import, and plain `@State` for ownership. As this recipe showed, the

migration between them is mechanical: swap the model's annotations and the three property wrappers, and delete `@Bindable`.

See also

- `ObservableObject` documentation: https://x.swiftuicookbook.com/yy5bv
- `Published` documentation: https://x.swiftuicookbook.com/208a9
- `StateObject` documentation: https://x.swiftuicookbook.com/jxr7x
- `EnvironmentObject` documentation: https://x.swiftuicookbook.com/pckny
- `environmentObject(_:)` documentation: https://x.swiftuicookbook.com/awsxi
- *Migrating from the Observable Object protocol to the Observable macro* article: https://x.swiftuicookbook.com/j4k6v

Reading and Defining Environment Values

The environment is not only a place for the models we inject ourselves. The system fills it with the settings that describe how a person has set up their device: whether they are in light or dark mode, how large they want text, which language and region they use, and which way their layout reads. It also places a few ready-made actions there, such as dismissing the current view or handing a URL off to the system. Reading those system values and actions is how an

app adapts to the person using it, without having to pass any of those settings down through every view's initializer. In UIKit we reached for `traitCollection`, overrode `traitCollectionDidChange(_:)` to react to appearance and size changes, and called `UIApplication.shared.open(_:)` to open a link. In SwiftUI all of this arrives declaratively through `@Environment`, and the view recomputes on its own when a value changes.

In this recipe, we build a settings dashboard. At the top sits a diagnostics panel that reads the current appearance, text size, width and height size classes, locale, and layout direction. It rearranges itself from a two-column grid into a single column when the user picks one of the accessibility text sizes in the Settings app, reacting to that outside change with no extra wiring on our part. Below it, a grid of category cards takes its column count not from an initializer but from a custom environment value we define with the `@Entry` macro, so a control elsewhere in the view can change the density of the whole grid at once. A modal sheet uses the environment's `dismiss` and `openURL` actions and reads the `locale` to show the current region. Along the way we override these values in previews, which is the fastest way to see dark mode, a large text size, or another language without touching device settings.

Between the three sections we end up using the environment in every direction it offers. The previous two recipes wrote to it, injecting first our `@Observable` and then our `ObservableObject` model so any descendant could read the shared player. In this recipe, we read what the system already places in the environment, the user's appearance, text size, locale, and so on, and we add a value of our own with the `@Entry` macro. The keyed lookup that makes every one of those reads

and writes work is the same; what changes is whether we own the value or the system does, and whether a missing read falls back to a default or trips a runtime trap.

Getting ready

Create a new SwiftUI iOS app named **SettingsDashboard** . We read several built-in environment values and define a custom one, so it helps to have read the *Sharing Data with the Environment Using Obse rvation* section first to see how the environment carries a shared model.

How to do it...

We start with the diagnostics panel that reads the system's environment values. Then comes a custom environment value of our own, along with the grid that reads that value. Next, we put together `ContentView` , which combines the panel, the grid, and a density control that sets the custom value. Finally, we add an About sheet that uses the environment's two built-in actions. These are the steps:

1. Start with the diagnostics panel, the part that reads what the system puts in the environment. Create a new file `EnvironmentInfoView.swift` in a `Views` folder. It declares six `@Environment` properties, one per value we want to show, and lays them out as labeled rows:

```
import SwiftUI
struct EnvironmentInfoView: View {
  @Environment(\.colorScheme) private var colorScheme
  @Environment(\.dynamicTypeSize) private var dynamicTypeSi
```

```
@Environment(\.horizontalSizeClass) private var horizonta
@Environment(\.verticalSizeClass) private var verticalSiz
@Environment(\.locale) private var locale
@Environment(\.layoutDirection) private var layoutDirecti
var body: some View {
  VStack(alignment: .leading, spacing: 12) {
    Text("Current Environment")
      .font(.headline)
    infoRows
  }
  .padding()
  .frame(maxWidth: .infinity, alignment: .leading)
  .background(.regularMaterial, in: .rect(cornerRadius: 1
}
@ViewBuilder private var infoRows: some View {
  let appearance = colorScheme == .dark ? "Dark" : "Light
  row("Appearance", value: appearance, icon: "circle.left
  row("Text Size", value: "\(dynamicTypeSize)", icon: "te
  let width = label(for: horizontalSizeClass)
  row("Width", value: width, icon: "arrow.left.and.right"
  let height = label(for: verticalSizeClass)
  row("Height", value: height, icon: "arrow.up.and.down")
  row("Locale", value: locale.identifier, icon: "globe")
  let direction = layoutDirection == .leftToRight ? "LTR"
  row("Direction", value: direction, icon: "text.alignlef
}
private func row(_ label: String, value: String, icon: St
  HStack {
    Image(systemName: icon)
      .foregroundStyle(.tint)
      .frame(width: 22)
    VStack(alignment: .leading, spacing: 1) {
      Text(label)
        .font(.caption)
        .foregroundStyle(.secondary)
      Text(value)
        .font(.subheadline.weight(.medium))
    }
  }
}
private func label(for sizeClass: UserInterfaceSizeClass?
  switch sizeClass {
  case .compact: "Compact"
```

```
    case .regular: "Regular"
    default: "Unknown"
    }
  }
}
#Preview {
  EnvironmentInfoView()
    .padding()
}
```

2. The panel reads fine, but at large text sizes a two-column grid would crowd the values. Make it adapt: when the text size reaches an accessibility size, stack the rows in a single column; otherwise lay them out in a two-column grid. Only the `body` changes:

```
var body: some View {
  VStack(alignment: .leading, spacing: 12) {
    Text("Current Environment")
      .font(.headline)
    if dynamicTypeSize.isAccessibilitySize {
      VStack(alignment: .leading, spacing: 12) {
        infoRows
      }
    } else {
      let columns = [GridItem(.flexible(), alignment: .lead
                     GridItem(.flexible(), alignment: .lead
      LazyVGrid(columns: columns, spacing: 12) {
        infoRows
      }
    }
  }
  .padding()
  .frame(maxWidth: .infinity, alignment: .leading)
  .background(.regularMaterial, in: .rect(cornerRadius: 16)
}
// ... infoRows, row, and label are unchanged
```

3. The grid below the panel needs something to show. Create `SettingCategory.swift` in a `Models` folder with a small model and a handful of sample categories:

```
import SwiftUI
struct SettingCategory: Identifiable {
  let id = UUID()
  let name: String
  let icon: String
  let tint: Color
}
extension SettingCategory {
  static let all: [SettingCategory] = [
    SettingCategory(name: "Display", icon: "sun.max.fill",
    SettingCategory(name: "Sounds", icon: "speaker.wave.2.f
    SettingCategory(name: "Privacy", icon: "lock.shield.fil
    SettingCategory(name: "Network", icon: "wifi", tint: .b
    SettingCategory(name: "Storage", icon: "internaldrive.f
    SettingCategory(name: "Battery", icon: "battery.100perc
  ]
}
```

4. Now, define the value that will drive the grid's column count. Because it is not one of the system's values, we add it ourselves. Create `DashboardEnvironment.swift` in the `Models` folder and declare a custom environment value with the `@Entry` macro:

```
import SwiftUI
extension EnvironmentValues {
  @Entry var dashboardColumns: Int = 2
}
```

The `2` is the default value the grid uses until an ancestor sets a different value.

5. Create `SettingsCardsView.swift` in the `Views` folder. Notice that it does not take a column count as a parameter. It reads `dashboardColumns` from the environment, so whoever displays the grid gets to decide the number of columns:

```
import SwiftUI
struct SettingsCardsView: View {
  @Environment(\.dashboardColumns) private var dashboardCol
  private let categories = SettingCategory.all
  private var columns: [GridItem] {
    Array(repeating: GridItem(.flexible(), spacing: 12),
          count: dashboardColumns)
  }
  var body: some View {
    LazyVGrid(columns: columns, spacing: 12) {
      ForEach(categories) { category in
        VStack(spacing: 8) {
          Image(systemName: category.icon)
            .font(.title2)
            .foregroundStyle(category.tint)
          Text(category.name)
            .font(.subheadline)
            .multilineTextAlignment(.center)
        }
        .frame(maxWidth: .infinity)
        .padding(.vertical, 20)
        .background(.regularMaterial, in: .rect(cornerRadiu
      }
    }
  }
}
#Preview {
  ScrollView {
    SettingsCardsView()
      .environment(\.dashboardColumns, 2)
      .padding()
  }
}
```

6. With the pieces in place, assemble them in `ContentView`. Wrap a `ScrollView` in a `NavigationStack`, stack the panel and the grid, and inject a column count into the environment for the grid to read:

```
struct ContentView: View {
  @State private var cardColumns = 2
  var body: some View {
    NavigationStack {
      ScrollView {
        VStack(spacing: 20) {
          EnvironmentInfoView()
          SettingsCardsView()
            .environment(\.dashboardColumns, cardColumns)
        }
        .padding()
      }
      .background(Color(.systemGroupedBackground))
      .navigationTitle("Settings")
    }
  }
}
```

   For now, we inject a fixed value of `2`, matching the default. Next, we let the user change it with a control.

7. Give the user control over the column count. Add a segmented `Picker` bound to `cardColumns`, between the panel and the grid. Because the grid reads its column count from the environment, moving the picker reflows it right away:

```
struct ContentView: View {
  @State private var cardColumns = 2
  var body: some View {
    NavigationStack {
      ScrollView {
        VStack(spacing: 20) {
          EnvironmentInfoView()
```

```
            layoutControl
            SettingsCardsView()
              .environment(\.dashboardColumns, cardColumns)
          }
          .padding()
        }
        // ... background and navigationTitle
      }
    }
    private var layoutControl: some View {
      VStack(alignment: .leading, spacing: 8) {
        Text("Card Columns")
          .font(.headline)
        Picker("Card Columns", selection: $cardColumns) {
          Text("1").tag(1)
          Text("2").tag(2)
          Text("3").tag(3)
        }
        .pickerStyle(.segmented)
      }
      .frame(maxWidth: .infinity, alignment: .leading)
    }
}
```

8. Now, the About sheet. Create `AboutView.swift` in the `Views` folder. It reads `locale` to show the region and uses the environment's `dismiss` action to close itself through a standard Liquid Glass close button:

```
import SwiftUI
struct AboutView: View {
  @Environment(\.dismiss) private var dismiss
  @Environment(\.locale) private var locale
  var body: some View {
    NavigationStack {
      Form {
        Section("About") {
          LabeledContent("Version", value: "4.0.0")
          LabeledContent("Region", value: locale.region?.ic
        }
```

```
      }
      .navigationTitle("About")
      .toolbarTitleDisplayMode(.inline)
      .toolbar {
        ToolbarItem(placement: .confirmationAction) {
          Button(role: .close) { dismiss() }
        }
      }
    }
  }
}
#Preview {
  AboutView()
}
```

9. Add a Support section with two links. `openURL` hands a URL to the system, so the first button opens a website in Safari:

```
@Environment(\.dismiss) private var dismiss
@Environment(\.openURL) private var openURL
@Environment(\.locale) private var locale
Form {
  // ... About section
  Section("Support") {
    Button {
      openURL(URL(string: "https://developer.apple.com")!)
    } label: {
      Label("Visit Website", systemImage: "safari")
    }
    Button {
      openURL(URL(string: "mailto:feedback@example.com")!)
    } label: {
      Label("Send Feedback", systemImage: "envelope")
    }
  }
}
```

Tapping Send Feedback, though, does nothing in the simulator, and the console logs a warning, because the simulator has no Mail app to handle a `mailto:` link. This is a good moment to build the app onto a real iPhone, where Mail is present and the same button opens a pre-addressed message. Whenever a recipe leans on a system app or a capability the simulator does not ship, running on a device is the only way to see the real behavior.

10. Handle that case instead of letting it fail silently. `openURL` has a completion-handler form that reports whether the URL could be opened, and when it could not, we show an alert. The new state goes with the other properties above the `body`, the completion handler replaces the Send Feedback button's action inside the `Support` section, and the `.alert` attaches to the `Form`:

```
struct AboutView: View {
  // ... existing dismiss, openURL, and locale properties
  @State private var showMailError = false
  var body: some View {
    NavigationStack {
      Form {
        // ... About section
        Section("Support") {
          // ... Visit Website button
          Button {
            openURL(URL(string: "mailto:feedback@example.co
              showMailError = !accepted
            }
          } label: {
            Label("Send Feedback", systemImage: "envelope")
          }
        }
      }
      // ... navigationTitle, toolbarTitleDisplayMode, and
      .alert("Can't Send Feedback", isPresented: $showMailE
```

```
            Button("OK", role: .cancel) { }
        } message: {
            Text("This device has no email account set up.")
        }
    }
  }
}
```

11. Present the sheet from `ContentView`. The new state joins `cardColumns` above the `body`, and the toolbar button and the `.sheet` attach to the `ScrollView`, below the `.navigationTitle` we already set in step 6:

```
struct ContentView: View {
  // ... existing cardColumns property
  @State private var showingAbout = false
  var body: some View {
    NavigationStack {
      ScrollView {
        // ... the panel, the layout control, and the cards
      }
      .background(Color(.systemGroupedBackground))
      .navigationTitle("Settings")
      .toolbar {
        ToolbarItem(placement: .topBarTrailing) {
          Button {
            showingAbout = true
          } label: {
            Image(systemName: "info.circle")
          }
        }
      }
      .sheet(isPresented: $showingAbout) {
        AboutView()
      }
    }
  }
  // ... layoutControl is unchanged
}
```

12. Environment values are easy to exercise without leaving Xcode: override them in previews. Add a few `#Preview` blocks that force dark mode, an accessibility text size, and a Spanish locale:

```
#Preview("Light") {
  ContentView()
}
#Preview("Dark") {
  ContentView()
    .preferredColorScheme(.dark)
}
#Preview("Large Text") {
  ContentView()
    .environment(\.dynamicTypeSize, .accessibility3)
}
#Preview("Spanish") {
  ContentView()
    .environment(\.locale, Locale(identifier: "es_ES"))
}
```

    Each preview shows how the dashboard responds: the panel collapsing at the large text size, and the locale row reading `es_ES` under Spanish. If one of them comes up ignoring its override, showing the standard text size in the Large Text preview for instance, that is the preview canvas holding on to a stale build rather than a mistake in the code. Editing the file to force a rebuild, or closing and reopening the preview canvas, sets it right.

13. Run the app in Device Hub. The column control and the About sheet both work in the preview canvas, but handing a URL off to Safari needs a running system to hand it to, so this step needs Device Hub. Move the Card Columns control and watch the grid

reflow between one, two, and three columns, all driven by the custom environment value. Open the About sheet from the info button to see the two actions at work: the close button dismisses the sheet, and Visit Website hands the link to Safari.

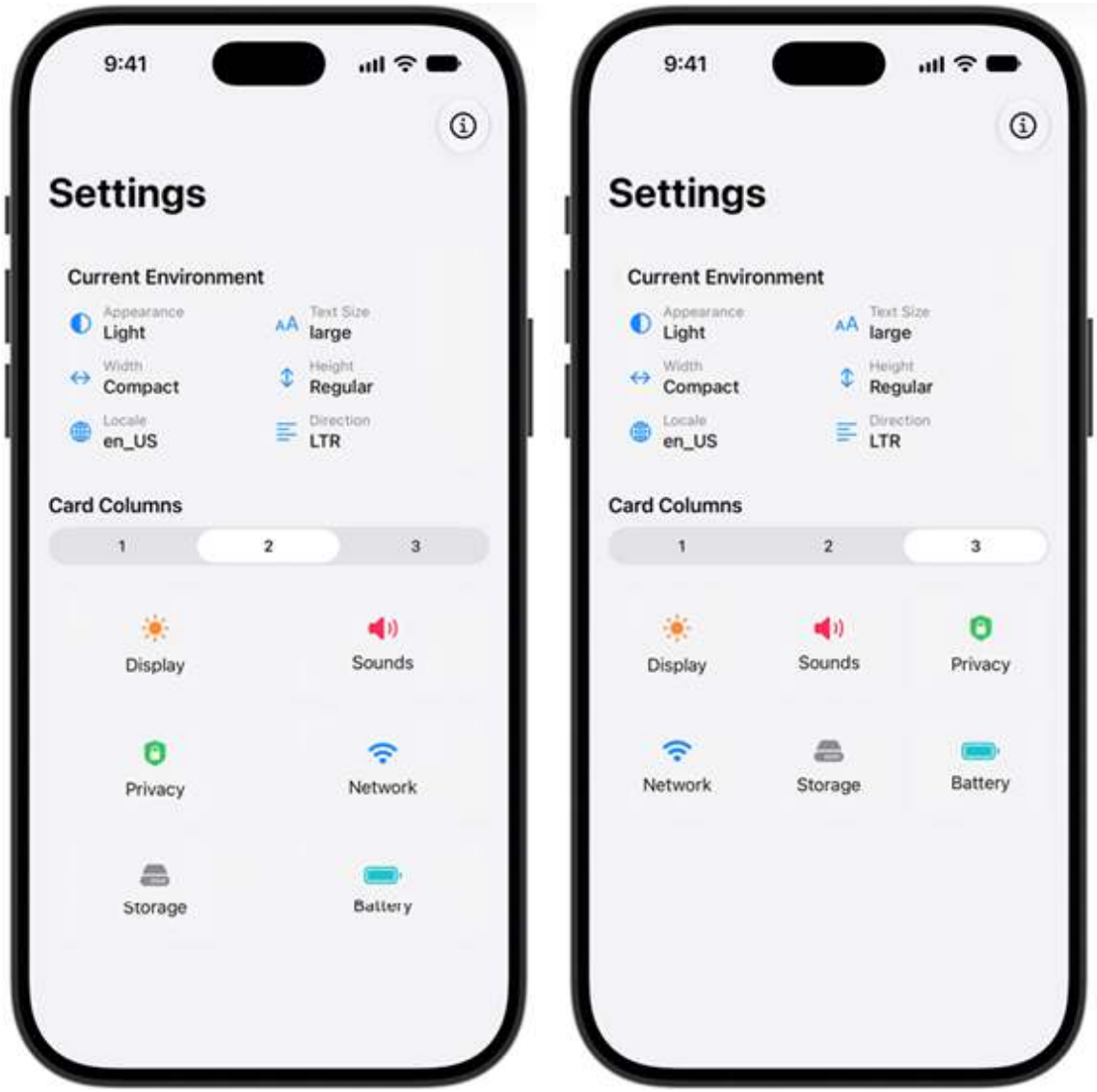


Figure 6.3: The Card Columns control reflows the grid through a custom environment value

14. Raise the text size past the accessibility threshold, either with the Large Text preview or with **Device** > **Accessibility** > **Increase**

Text Size in Device Hub, and the panel rearranges itself into a single col umn:

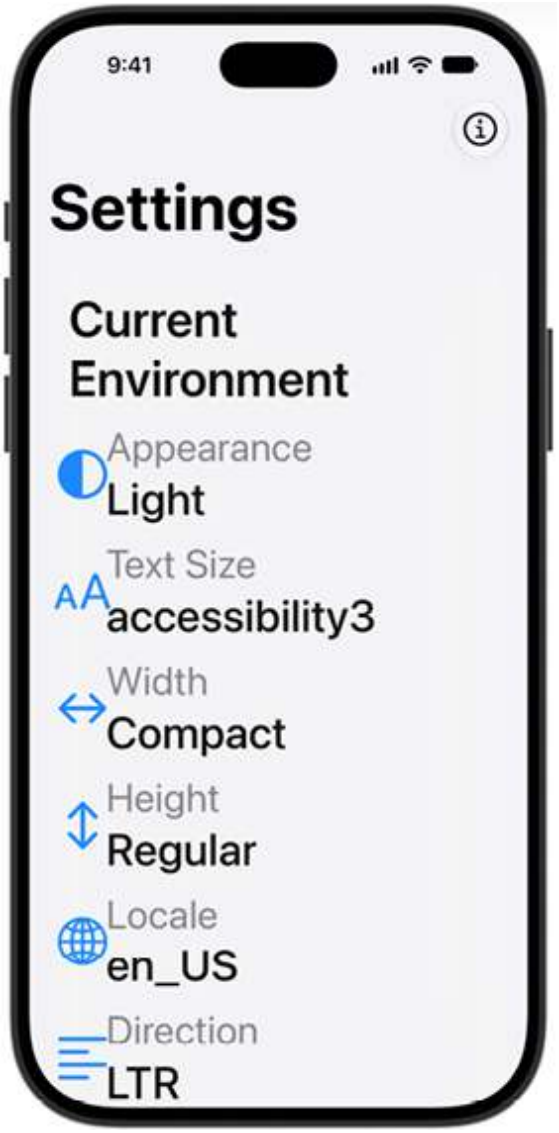


Figure 6.4: At an accessibility text size, the panel collapses to a single column

How it works...

Every value in our panel arrives the same way: we declare a property with `@Environment` and a key path, and SwiftUI reads the current value from the view's environment. When the user changes any of these, in

Settings, in Control Center, or by rotating the device, the view recomputes and the panel stays current with no work from us. This is the same environment we used in the previous two sections to carry a shared model; the difference here is that the values were placed there by the system rather than by us.

The values we read, `colorScheme`, `dynamicTypeSize`, `horizontalSizeClass`, `verticalSizeClass`, `locale`, and `layoutDirection`, describe the device's appearance and the user's preferences. We branched on `dynamicTypeSize.isAccessibilitySize` to switch the panel from a two-column grid to a single column, the kind of adaptation that matters most on iPhone, where text can grow far beyond the default. The two size classes are the main lever for adapting between a compact iPhone layout and a roomier iPad one; on iPhone in portrait the width is always compact, so we surface the values here rather than branch on them. Overriding any of these in a `#Preview`, as we did with `.preferredColorScheme(.dark)`, an accessibility `dynamicTypeSize`, and a Spanish `locale`, is the quickest way to check those states without changing device settings.

Two of the things we read are not data but actions. `dismiss` is a closure the environment hands to whatever presented the current view; calling it from the About sheet's close button is enough to dismiss the sheet, with no binding passed down. `openURL` hands a URL to the system. We used its plain form for the website link and its completion-handler form for the feedback link: the closure receives a Boolean telling us whether anything could open the URL, so instead of failing silently when no mail account exists, we raise an alert. The close button itself uses `Button(role: .close)`, which renders the standard close glyph.

Our card grid reads a value the system does not provide, `dashboardColumns`, which we defined in a single line with the `@Entry` macro. `@Entry` is a convenience: it generates the boilerplate that custom environment values required before iOS 18. Written out by hand, our one-line declaration expands to roughly this:

```
private struct DashboardColumnsKey: EnvironmentKey {
  static let defaultValue = 2
}
extension EnvironmentValues {
  var dashboardColumns: Int {
    get { self[DashboardColumnsKey.self] }
    set { self[DashboardColumnsKey.self] = newValue }
  }
}
```

The macro creates a private key type that carries the default value, then a computed property on `EnvironmentValues` whose getter and setter store and read the value by that key. That default, `2`, is what a view reads when no ancestor has set the value, which is why a keyed environment value never traps the way a missing injected object does in the *Sharing Data with the Environment Using Observation* section. We set the value with `.environment(\.dashboardColumns, cardColumns)` in `ContentView` and read the value with `@Environment(\.dashboardColumns)` in the grid, exactly the pair of calls we used for the system values. The payoff is loose coupling: the control that sets the density and the grid that reads it never reference each other, so neither has to pass the value through the views in between. For something a direct child needs, a plain initializer parameter is clearer. The environment earns its place when a setting is read in many places or deep down the view

hierarchy , like a theme, a layout density, or the shared model we built in the previous two sections.

See also

- `EnvironmentValues` documentation: https://x.swiftuicookbook.com/trzq2
- `Environment` documentation: https://x.swiftuicookbook.com/wpufx
- `Entry()` documentation: https://x.swiftuicookbook.com/vo8zw
- `DismissAction` documentation: https://x.swiftuicookbook.com/9gp6i
- `OpenURLAction` documentation: https://x.swiftuicookbook.com/mi2eu
- `DynamicTypeSize` documentation: https://x.swiftuicookbook.com/8984e

Accessibility APIs

In UIKit, accessibility used to feel like a separate pass at the end of the project. We set `accessibilityLabel` , `accessibilityHint` , and `accessibilityValue` on individual `UIView` instances, overrode `accessibilityElements` or `accessibilityActivate` where the defaults were not enough, and built a `UIAccessibilityCustomAction` array for behaviors that lived behind gestures. The work was real, easy to forget, and only visible when VoiceOver was running. In SwiftUI, accessibility lives in the same place as everything else: as view modifiers we add

as we build. `.accessibilityLabel`, `.accessibilityHint`, `.accessibilityValue`, and `.accessibilityElement(children:)` decide what VoiceOver hears at each focus stop. `.accessibilityAddTraits`, `.accessibilityAction(named:)`, and `.accessibilityRotor` decide how a user navigates between them. `@AccessibilityFocusState` shifts focus when an event changes what should be announced next. Treating accessibility as we go, rather than at the end, keeps the work small and the result honest.

In this recipe, we build a recipe browser, a list of dishes grouped by category. Each row carries a difficulty indicator drawn from five small dots, the recipe name, the prep time, the category, and a yellow star when the recipe is a favorite. The toolbar carries a plus-sign button that adds another sample from a small pool; when the pool runs out, an error row appears at the bottom of the list and VoiceOver focus jumps to it. Two swipe gestures expose favorite-toggle and delete; both also live as VoiceOver custom actions on each row, so users navigating with VoiceOver reach them through the actions menu instead of a swipe. Two custom rotors, `Favorites` and `Quick Recipes`, let VoiceOver users jump between favorited recipes or recipes that take fifteen minutes or less to make.

To verify our work as we go we use Xcode's Accessibility Inspector, which attaches to the simulator and exposes the same information VoiceOver would announce, including an audible preview of each element's label, value, and hint. The Inspector also runs a built-in accessibility audit and simulates settings like larger text and increased contrast. Two behaviors are felt rather than inspected, the rotor twist gesture and the focus cursor jumping in response to `@AccessibilityFocusState`. The Inspector confirms their wiring in the

accessibility tree, and VoiceOver i tself, which Device Hub in Xcode 27 can run inside the simulator, is how a user experiences them.

Getting ready

Create a new SwiftUI iOS app named **RecipeBrowser** . The recipe stands on its own and does not build on the previous sections. We run the app in Device Hub so Xcode's Accessibility Inspector can attach to it and verify the work as we go.

Open the Inspector now, before writing any accessibility code, from **Xcode** > **Open Developer Tool** > **Accessibility Inspector** , and point it at the app as it grows. Almost every modifier we add from step 2 onward changes something we can hear or see there, and checking each one as we go beats adding modifiers on faith and discovering at the end tha t VoiceOver says something we did not intend. Step 14 walks through the Inspector's tabs in full.

How to do it...

We build the recipe browser bottom-up: the data first, then the smallest accessible piece (the difficulty indicator), then the row, and finally the assembled view with the toolbar, the rotors, and the error state. After each accessibility modifier we pause to consider what VoiceOver would say, and at the end we confirm it through the Accessibility Inspector. These are the steps:

1. Create `Recipe.swift` in a `Models` folder. The model carries everything a row needs to render and to announce: the name, the category, the prep time, an integer difficulty, a mutable `isFavorite`

, and an ingredients list. A static `samples` array seeds the list with eight recipes spanning categories from Italian to Japanese:

```
import Foundation
struct Recipe: Identifiable, Hashable {
  let id = UUID()
  let name: String
  let category: String
  let prepTime: Int
  let difficulty: Int
  var isFavorite: Bool
  let ingredients: [String]
}
extension Recipe {
  static let samples: [Recipe] = [
    Recipe(
      name: "Pasta Carbonara",
      category: "Italian",
      prepTime: 25,
      difficulty: 3,
      isFavorite: true,
      ingredients: [
        "Spaghetti", "Eggs", "Pecorino", "Guanciale", "Blac
      ]
    ),
    // ... seven more recipes (Caesar Salad, Chicken Tikka
    // Greek Salad, Beef Tacos, Margherita Pizza, Pad Thai,
  ]
}
```

Only the first sample is printed here to save space. Copy the full eight-recipe `samples` array from this chapter's project in the book's GitHub repository rather than inventing the rest by hand, because the steps that follow assume the complete set.

2. Build the smallest custom control next. Create `DifficultyIndicatorView.swift` in a `Views` folder. Five dots fill from

left to right based on the level. The dots are visual, so VoiceOver gets a single element with a label and a value through `accessibilityElement(children: .ignore)`:

```
import SwiftUI
struct DifficultyIndicatorView: View {
  let level: Int
  var body: some View {
    HStack(spacing: 3) {
      ForEach(1...5, id: \.self) { dot in
        Circle()
          .fill(dot <= level ? Color.orange : Color.gray.op
          .frame(width: 8, height: 8)
      }
    }
    .accessibilityElement(children: .ignore)
    .accessibilityLabel("Difficulty")
    .accessibilityValue("\(level) out of 5")
  }
}
#Preview {
  VStack(spacing: 16) {
    DifficultyIndicatorView(level: 1)
    DifficultyIndicatorView(level: 3)
    DifficultyIndicatorView(level: 5)
  }
  .padding()
}
```

The `.ignore` value drops the five circles from the accessibility tree, since reading them one by one would say nothing useful, and the label and value land in their place.

3. Now, the row that uses it. Create `RecipeRowView.swift` in the `Views` folder. We start with the visual structure alone, no accessibility modifiers yet, so we have a clean place to layer them on:

```
import SwiftUI
struct RecipeRowView: View {
  let recipe: Recipe
  var body: some View {
    HStack(spacing: 12) {
      DifficultyIndicatorView(level: recipe.difficulty)
      VStack(alignment: .leading, spacing: 4) {
        Text(recipe.name)
          .font(.headline)
        HStack(spacing: 4) {
          Label("\(recipe.prepTime) min", systemImage: "clc
            .labelStyle(.titleAndIcon)
          Text("•")
          Text(recipe.category)
        }
        .font(.caption)
        .foregroundStyle(.secondary)
      }
      Spacer()
      if recipe.isFavorite {
        Image(systemName: "star.fill")
          .foregroundStyle(.yellow)
      }
    }
  }
}
#Preview {
  List {
    RecipeRowView(recipe: Recipe.samples[0])
    RecipeRowView(recipe: Recipe.samples[1])
  }
}
```

Without any tuning, VoiceOver would announce each subview in turn (the difficulty's value, the name, the time label, the category, the star), which is busier than it needs to be. The next steps bring the row down to one well-composed sentence.

4. Combine the row into a single accessibility element with a label written for VoiceOver and a hint that explains what tapping it would do. A `composedLabel` helper builds the sentence:

```
var body: some View {
  HStack(spacing: 12) {
    // ... existing subviews
  }
  .accessibilityElement(children: .combine)
  .accessibilityLabel(composedLabel)
  .accessibilityHint("Double tap to view recipe details")
}
private var composedLabel: String {
  var parts = [recipe.name, recipe.category]
  parts.append("\(recipe.prepTime) minutes prep time")
  parts.append("Difficulty \(recipe.difficulty) of 5")
  if recipe.isFavorite { parts.append("Favorited") }
  return parts.joined(separator: ", ")
}
```

   With these three modifiers in place, VoiceOver would announce the row as *Pasta Carbonara, Italian, 25 minutes prep time, Difficulty 3 of 5, Favorited, 3 out of 5, Double tap to view recipe details.* Most of that is what we want, but the trailing "3 out of 5" is the difficulty indicator's value bleeding through. The next step takes care of it.

5. `accessibilityElement(children: .combine)` merges children's labels into the parent label but leaves their values intact, which is why the indicator's `3 out of 5` appended itself to the announcement. We have already written `Difficulty 3 of 5` into `composedLabel`, so the indicator's contribution is redundant. Add `.accessibilityHidden(true)` on the indicator inside the row to remove it from the row's accessibility tree (the indicator type still

owns its label and value for any future use elsewhere). Give the favorite star the same modifier: the label already says the recipe is a favorite, so VoiceOver can skip the star:

```
HStack(spacing: 12) {
  DifficultyIndicatorView(level: recipe.difficulty)
    .accessibilityHidden(true)
  // ... the name, prep time, and category
  Spacer()
  if recipe.isFavorite {
    Image(systemName: "star.fill")
      .foregroundStyle(.yellow)
      .accessibilityHidden(true)
  }
}
// ... the accessibility modifiers and composedLabel are ur
```

VoiceOver now reads *Pasta Carbonara, Italian, 25 minutes prep time, Difficulty 3 of 5, Favorited. Double tap to view recipe details.* That is the one-sentence row we wanted.

6. Make the row's two behaviors, favorite and delete, reachable from VoiceOver's actions menu. A VoiceOver user opens that menu by swiping one finger up or down on a focused element, a gesture distinct from the horizontal swipe-to-reveal we will add for users not using VoiceOver in the next step. Add two callbacks to the view, two `.accessibilityAction(named:)` modifiers, and a `favoriteActionName` helper that switches the name with the state:

```
struct RecipeRowView: View {
  let recipe: Recipe
  let onToggleFavorite: () -> Void
  let onDelete: () -> Void
```

```
  var body: some View {
    HStack(spacing: 12) {
      // ... existing subviews
    }
    .accessibilityElement(children: .combine)
    .accessibilityLabel(composedLabel)
    .accessibilityHint("Double tap to view recipe details")
    .accessibilityAction(named: favoriteActionName) { onTog
    .accessibilityAction(named: "Delete") { onDelete() }
  }
  private var favoriteActionName: String {
    recipe.isFavorite ? "Remove from favorites" : "Add to f
  }
  // ... composedLabel is unchanged
}
```

The row now takes two more arguments, so the preview from step 3 needs them too. Passing empty closures, `RecipeRowView(recipe: Recipe.samples[0], onToggleFavorite: {}, onDelete: {})`, keeps it compiling and rendering while we work.

7. For users not using VoiceOver the same two behaviors are reachable through swipe gestures. Add `.swipeActions` blocks below the accessibility modifiers. The leading edge reveals the favorite toggle (yellow); the trailing edge reveals delete, and `Button(role: .destructive)` gives it the standard red tint without us writing any color:

```
var body: some View {
  HStack(spacing: 12) {
    // ... existing subviews
  }
  // ... combine, label, hint, accessibility actions
  .swipeActions(edge: .leading) {
    Button {
      onToggleFavorite()
    } label: {
```

```
            Label(
              recipe.isFavorite ? "Unfavorite" : "Favorite",
              systemImage: recipe.isFavorite ? "star.slash" : "st
            )
          }
          .tint(.yellow)
          .accessibilityLabel(favoriteActionName)
        }
        .swipeActions(edge: .trailing) {
          Button(role: .destructive) {
            onDelete()
          } label: {
            Label("Delete", systemImage: "trash")
          }
        }
    }
```

The same two behaviors are now reachable two ways: a swipe gesture in the `List` for users not using VoiceOver, and a VoiceOver action in the menu for users who are. Neither path is more "accessible" than the other; they are two complete inputs to the same outcome. Treating both as first-class entry points keeps the design coherent and avoids the trap of bolting an accessibility layer onto the app at the end.

One subtlety is worth naming. Swipe-action buttons are exposed to VoiceOver as accessibility actions automatically, using each button's visible label as the action name. Without intervention the favorite swipe button would create a near-duplicate of the action we already defined in step 6 (the swipe button's visible "Favorite" or "Unfavorite" alongside our descriptive "Add to favorites" or "Remove from favorites"). The `.accessibilityLabel(favoriteActionName)` we added overrides what VoiceOver hears for the swipe button, matching it to the action

name we registered, and the system collapses them into one. The delete swipe button does not need the same treatment because its visible "Delete" already matches our `.accessibilityAction(named: "Delete")`.

8. Create `RecipeDetailView.swift` in the `Views` folder. The row will navigate here when tapped, so the existing hint becomes accurate. The view uses `LabeledContent` for the structured fields, which are accessible out of the box. The difficulty row reuses `DifficultyIndicatorView` for the visual instead of plain text; we hide its accessibility element so it does not duplicate the row's announcement, and set an explicit `accessibilityValue` on the `LabeledContent` so VoiceOver hears one clean label-and-value:

```
import SwiftUI
struct RecipeDetailView: View {
  let recipe: Recipe
  var body: some View {
    List {
      Section {
        LabeledContent("Category", value: recipe.category)
        LabeledContent("Prep time", value: "\(recipe.prepTi
        LabeledContent("Difficulty") {
          DifficultyIndicatorView(level: recipe.difficulty)
            .accessibilityHidden(true)
        }
        .accessibilityValue("\(recipe.difficulty) of 5")
      }
      Section("Ingredients") {
        ForEach(recipe.ingredients, id: \.self) { ingredier
          Text(ingredient)
        }
      }
    }
    .navigationTitle(recipe.name)
  }
}
```

```
#Preview {
  NavigationStack {
    RecipeDetailView(recipe: Recipe.samples[0])
  }
}
```

9. Move to `ContentView.swift` and assemble the main view. The list is held in a `@State` array seeded with the first five of `Recipe.samples`, because the favorite and delete actions we wired into the row mutate it. Wrap a `List` in a `NavigationStack`, group recipes by category with a `Dictionary(grouping:by:)`, wrap each row in a `NavigationLink` that targets `RecipeDetailView`, and give each section header the `.isHeader` trait so VoiceOver users can navigate by section through the built-in headings rotor:

```
struct ContentView: View {
  @State private var recipes = Array(Recipe.samples.prefix(
  private var groupedRecipes: [(category: String, items: [R
    Dictionary(grouping: recipes, by: \.category)
      .sorted(by: { $0.key < $1.key })
      .map { (category: $0.key, items: $0.value) }
  }
  var body: some View {
    NavigationStack {
      List {
        ForEach(groupedRecipes, id: \.category) { group in
          Section {
            ForEach(group.items) { recipe in
              NavigationLink {
                RecipeDetailView(recipe: recipe)
              } label: {
                RecipeRowView(
                  recipe: recipe,
                  onToggleFavorite: { toggleFavorite(recipe
                  onDelete: { delete(recipe) }
                )
              }
```

```
            }
          } header: {
            Text(group.category)
              .accessibilityAddTraits(.isHeader)
          }
        }
      }
      .navigationTitle("Recipes")
    }
  }
  private func toggleFavorite(_ recipe: Recipe) {
    if let index = recipes.firstIndex(of: recipe) {
      recipes[index].isFavorite.toggle()
    }
  }
  private func delete(_ recipe: Recipe) {
    recipes.removeAll { $0.id == recipe.id }
  }
}
```

10. Add the toolbar's plus button and the pool of spare recipes that drives it. The button uses the `Button(_:systemImage:)` initializer, which renders as the system image in a toolbar and uses the title as VoiceOver's accessibility label automatically. The `.accessibilityHint` explains what happens when tapped. A separate `availableToAdd` array holds the three spare samples, and a new `addRecipe()` method moves one of them into `recipes` each time the button is tapped. Note that `delete(_:)` changes here as well: rather than simply dropping the recipe, it now hands the removed one back to the pool, so the plus button can offer it again later:

```
@State private var recipes = Array(Recipe.samples.prefix(5))
@State private var availableToAdd = Array(Recipe.samples.dr
// ... groupedRecipes is unchanged
var body: some View {
```

```
  NavigationStack {
    List {
      // ... existing grouped sections
    }
    .navigationTitle("Recipes")
    .toolbar {
      ToolbarItem(placement: .primaryAction) {
        Button("Add", systemImage: "plus") {
          addRecipe()
        }
        .accessibilityHint("Adds a sample recipe to your li
      }
    }
  }
}
// ... toggleFavorite is unchanged
private func delete(_ recipe: Recipe) {
  if let index = recipes.firstIndex(of: recipe) {
    let removed = recipes.remove(at: index)
    availableToAdd.append(removed)
  }
}
private func addRecipe() {
  guard !availableToAdd.isEmpty else { return }
  recipes.append(availableToAdd.removeFirst())
}
```

Tapping the plus button adds the next spare recipe to the list, again, and again. When the pool is empty, nothing happens, yet.

11. VoiceOver lets a user twist two fingers on the screen to cycle through rotors, which are categories of elements that share a property: headings, links, form fields, and so on. We can register our own. Add a `Favorites` rotor that lists the favorited recipes and a `Quick Recipes` rotor that lists recipes at fifteen minutes or less:

```
var body: some View {
  NavigationStack {
    List {
      // ... existing grouped sections
    }
    .navigationTitle("Recipes")
    .accessibilityRotor("Favorites") {
      ForEach(recipes.filter(\.isFavorite)) { recipe in
        AccessibilityRotorEntry(recipe.name, id: recipe.id)
      }
    }
    .accessibilityRotor("Quick Recipes") {
      ForEach(recipes.filter { $0.prepTime <= 15 }) { recip
        AccessibilityRotorEntry(
          "\(recipe.name), \(recipe.prepTime) minutes",
          id: recipe.id
        )
      }
    }
    // ... existing toolbar
  }
}
```

12. When the pool runs out, the user deserves to know. Add an `errorMessage` state property and an `@AccessibilityFocusState` Boolean property that drives where VoiceOver's cursor sits. When the cap fires we move the cursor to the error row so a VoiceOver user is not left wondering why the button did nothing. The error row collapses into a single accessibility element with an explicit label and the `.isStaticText` trait so it is hoverable in the Inspector and a clean target for the focus state. All three methods change in this step: `addRecipe()` trades its `guard` for an `if` and `else` so an empty pool sets `errorMessage` instead of returning silently, and `toggleFavorite(_:)` and `delete(_:)` each

clear the message so a stale error does not linger after the next successful action:

```
@State private var recipes = Array(Recipe.samples.prefix(5)
@State private var availableToAdd = Array(Recipe.samples.dr
@State private var errorMessage: String?
@AccessibilityFocusState private var isErrorFocused: Bool
// ... groupedRecipes is unchanged
var body: some View {
  NavigationStack {
    List {
      // ... existing grouped sections
      if let errorMessage {
        Section {
          HStack {
            Image(systemName: "exclamationmark.triangle.fil
            Text(errorMessage)
          }
          .font(.caption)
          .foregroundStyle(.red)
          .accessibilityElement(children: .combine)
          .accessibilityLabel(errorMessage)
          .accessibilityAddTraits(.isStaticText)
          .accessibilityFocused($isErrorFocused)
        }
      }
    }
    // ... navigationTitle, rotors, toolbar
  }
}
private func toggleFavorite(_ recipe: Recipe) {
  if let index = recipes.firstIndex(of: recipe) {
    recipes[index].isFavorite.toggle()
  }
  errorMessage = nil
}
private func delete(_ recipe: Recipe) {
  if let index = recipes.firstIndex(of: recipe) {
    let removed = recipes.remove(at: index)
    availableToAdd.append(removed)
  }
  errorMessage = nil
```

```
}
private func addRecipe() {
  if availableToAdd.isEmpty {
    errorMessage = "No more sample recipes to add"
    isErrorFocused = true
  } else {
    recipes.append(availableToAdd.removeFirst())
    errorMessage = nil
  }
}
```

Setting `isErrorFocused = true` is enough; SwiftUI moves the VoiceOver cursor to whichever element binds `$isErrorFocused`, our error row here. The Inspector confirms the error element exists with the right label at the right moment; the live cursor jump is something only VoiceOver itself shows.

13. Run the app in Device Hub. The Accessibility Inspector attaches to a running app in the next step, so this is the moment to launch one, although the interactions themselves work just as well in the preview canvas. Add a couple of recipes to see them appear under their categories, swipe a row to reveal the favorite and delete actions, and keep tapping the plus button until the pool runs out to see the error row appear:

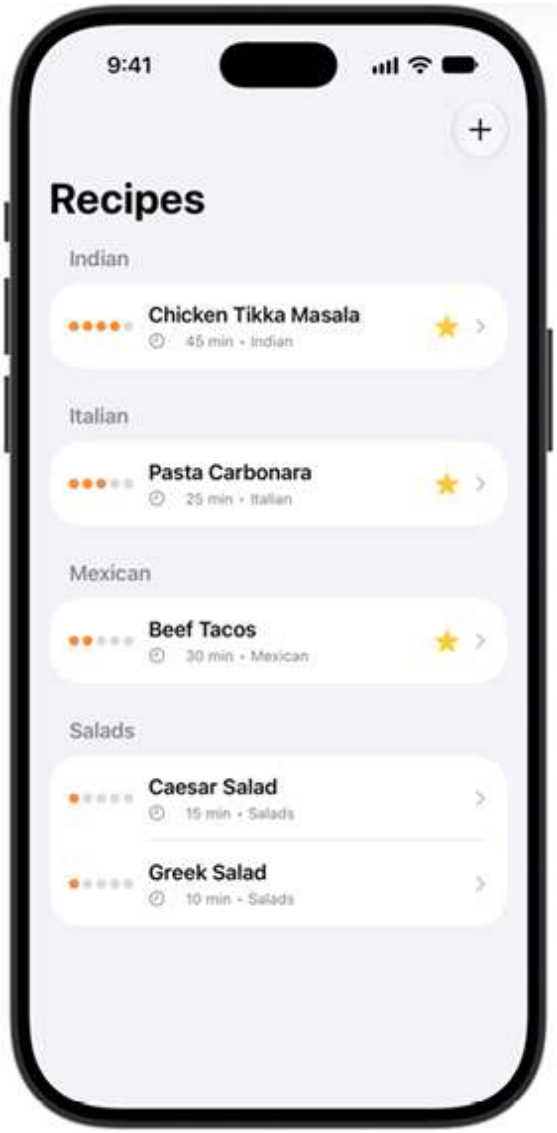


Figure 6.5: The finished recipe browser with four categories, three favorited recipes, and the toolbar's plus button

14. With the app running, open Xcode's Accessibility Inspector from **Xcode** > **Open Developer Tool** > **Accessibility Inspector** . Use the target dropdown at the top left of its toolbar to point it at the simulator (**Simulator** , then **RecipeBrowser**). Click the text-bubble icon to enable the audible speech preview, then click the crosshair to enter inspection mode.

Hovering the Pasta Carbonara row plays VoiceOver's announcement: *Pasta Carbonara, Italian, 25 minutes prep time, Difficulty 3 of 5, Favorited, Button. Double tap to view recipe details.* The trailing *Button* comes from the `NavigationLink` that now wraps the row, and it tells a VoiceOver user that the row can be activated. Expanding the Inspector's Actions section lists three custom actions for the row: `Activate` (which opens `RecipeDetailView`), `Delete`, and either `Remove from favorites` (for a favorited recipe) or `Add to favorites` (otherwise). Hovering a category title shows the Header trait. The Audit tab runs a full sweep against the running view hierarchy and surfaces the issues it finds; we will read its results in the next step. The Settings tab, the rightmost of the three icons at the top right of the Inspector window, lists common accessibility options to test, from **Increase contrast** and **Reduce motion** to **Bold text** and **Button shapes**, above a **Dynamic Type** slider, so we can preview how the list reads at the accessibility text sizes:

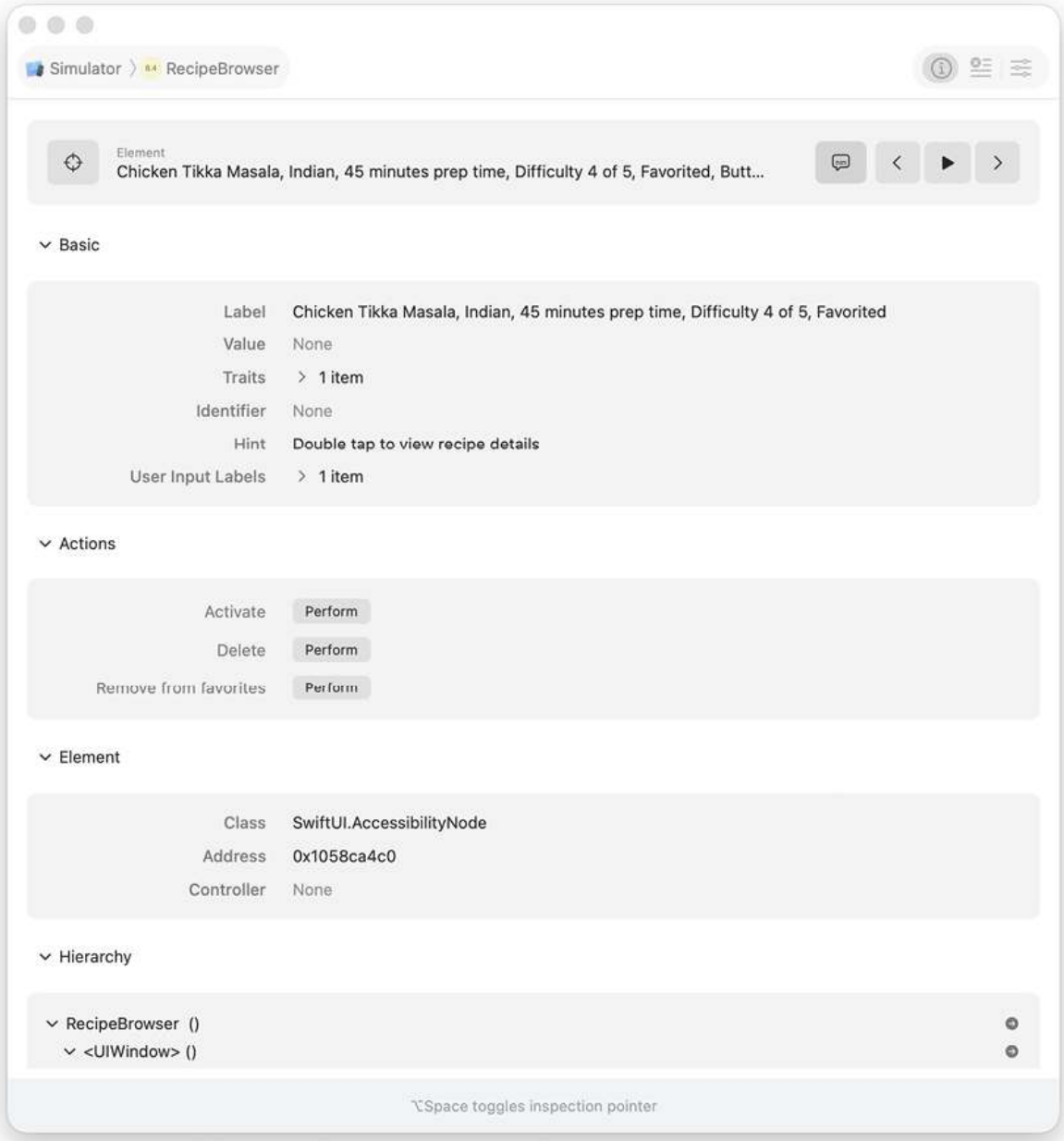


Figure 6.6: The Accessibility Inspector previewing a row's composed announcement and its custom actions

New in Xcode 27. Device Hub can run VoiceOver in the simulator. Choose **Device** > **Accessibility** > **VoiceOver** , dismiss the prompt that introduces



it, and VoiceOver reads the running app aloud. The same menu toggles **Reduce Motion** , **Reduce Transparency** , **Increase Contrast** , and **Show Borders** , and steps the text size up or down, so the accessibility settings a user changes in the Settings app are a menu away while the app runs.

15. Switch the Inspector to the Audit tab and click **Run Audit** , which sweeps the running app's view hierarchy and reports every accessibility problem it can detect. The result comes back with a few dozen warnings in two categories. The Contrast warnings, most of them marked as nearly passing, flag colors that fall just short of the recommended contrast against their background, a design decision best settled per project, often by choosing a darker fill or replacing `.foregroundStyle(.secondary)` with `.foregroundStyle(.primary)` . The Dynamic Type warnings, one for each piece of text on screen, deserve a closer look before we change anything. Select one and the Inspector highlights the label it refers to, such as a recipe name, yet that label already uses a text style like `.headline` , which scales with the user's text size on its own. Confirm it in the Settings tab: drag the **Dynamic Type** slider toward the largest size, and every label grows. These warnings come from the audit not recognizing how SwiftUI sizes its text, not from a problem in our code. The audit is a starting point, and its findings are worth checking before we act on them.

    What the slider does reveal is a problem the audit could not flag at all: the difficulty dots stay at a fixed eight points while the text around them grows, because they are hidden from

VoiceOver and so never reach the audit. Text styles take care of text. Any other dimension that should grow with the text needs `@ScaledMetric`, which scales a value proportionally with the user's Dynamic Type setting. Switch the dot dimension in `DifficultyIndicatorView` to one:

```
struct DifficultyIndicatorView: View {
  let level: Int
  @ScaledMetric private var dotSize: CGFloat = 8
  var body: some View {
    HStack(spacing: 3) {
      ForEach(1...5, id: \.self) { dot in
        Circle()
          .fill(dot <= level ? Color.orange : Color.gray.opacity
          .frame(width: dotSize, height: dotSize)
      }
    }
    // ... accessibility modifiers unchanged
  }
}
```

Drag the **Dynamic Type** slider again, and the dots now grow in step with the text. Run the audit once more and the counts stay the same, which is the lesson of this step: the audit checks what the accessibility elements report, while how a view's own dimensions scale is something only a look at the running app confirms. Pairing the automated sweep with a hands-on check at the largest text sizes is what catches the problems no audit can see.

How it works...

The row's announcement comes from three modifiers in cooperation. `accessibilityElement(children: .combine)` merges the five subviews into a single accessibility element and gathers their labels into one combined label; we replace that combined label with our own `composedLabel`, a comma-separated sentence written for VoiceOver's reading order. `accessibilityHint` adds the trailing `Double tap to view recipe details`, read after a short delay. VoiceOver always announces a focused element in the same order, label, value, hint, which is why the leftover value on the difficulty indicator landed between `Favorited` and the hint until we suppressed it with `accessibilityHidden(true)` inside the row. The indicator's own label and value are still defined on the type, ready for any context that does not have a parent label of its own.

The `children` parameter on `accessibilityElement` has three values that decide what happens to a parent's descendants. `.combine` keeps them in the tree but merges them into the parent's element, which is the right default for an information row like ours. `.ignore` drops them entirely and lets the parent provide its own label and value, which is exactly what the difficulty indicator wants: five dots are visual furniture, and `Difficulty, 3 out of 5` is what should be spoken. `.contain` keeps the children individually focusable inside a named container; that fits a shipping-address block or a credit-card form, where each line is still its own focus stop. The right choice is rarely about accessibility alone, it is about how a person navigating with VoiceOver would think about the shape of the content.

The two ways to take an action on a row, the swipe and the actions menu, sit side by side because they serve different inputs to the same outcome. `.swipeActions`, covered in the *Lists and Sections* section of

Chapter 2, is a List gesture available to anyone navigating by touch; `.accessibilityAction(named:)` adds the same action to VoiceOver's actions menu, the place a VoiceOver user reaches with a vertical swipe on a focused element. The favorite action even changes its name with the recipe's state, `Add to favorites` or `Remove from favorites`, so the menu reads as the next thing the user can do, not as the toggle's full state machine. The delete button uses the `Button(role: .destructive)` role for its red tint without any color code. Traits work the same way; `accessibilityAddTraits(.isHeader)` on a heading text makes VoiceOver's headings rotor surface it as a navigation stop. SwiftUI's `Section` header text gets the trait by default, so the explicit add in step 9 is here to teach the API rather than to change behavior; on a plain heading text outside a list it would not be optional.

Two of the modifiers go further. `accessibilityRotor` plugs into VoiceOver's two-finger twist gesture and lets us define our own categories of elements; the `Favorites` and `Quick Recipes` rotors here build entries from filtered slices of `recipes`, each pointing back at a row by `id`. iOS 15 added `@AccessibilityFocusState`, the accessibility counterpart of `@FocusState`. Where `@FocusState` moves the keyboard, `@AccessibilityFocusState` moves the VoiceOver cursor; setting `isErrorFocused = true` after the pool runs out brings VoiceOver to the error row at once, so a VoiceOver user is not left tapping the same button to nothing. Both are queryable in the Accessibility Inspector, but the user-felt experience, the twist that cycles between rotors and the cursor that jumps to the error, needs VoiceOver itself.

The toolbar's `Button("Add", systemImage: "plus")` is the modern shorthand: the system image is what we see in the toolbar, the title is the accessibility label VoiceOver reads. With no extra work the button

is already labeled `Add`; we only added the hint that explains what tapping it does. The Accessibility Inspector is how we verified every piece without leaving Device Hub: it attaches to the running simulator, lets us hover an element to hear the announcement it would produce, and runs a full audit. One modifier this recipe had no use for is still worth knowing. `.accessibilityIdentifier()` attaches a stable string to a view that VoiceOver never speaks. It exists for UI tests, which use it to find an element no matter what label the user hears or which language the app runs in. Set it whenever a view has to be addressable from a test, and keep it clearly apart from `.accessibilityLabel`, which is what a person actually hears. The trap to avoid is over-instrumenting. A view that already presents text and standard controls is already accessible; VoiceOver reads its labels, modifiers, and states for free. We add modifiers when a default would be wrong or missing, like a row composed of many pieces, a custom control with no native announcement, or a non-obvious interaction that s hould be hand-written. Accessibility is then a small, intentional surface area rather than a parallel implementation of the UI.

See also

- `Accessibility` documentation: https://x.swiftuicookbook.com/xucw5
- `accessibilityLabel(_:)` documentation: https://x.swiftuicookbook.com/c5anq
- `accessibilityElement(children:)` documentation: https://x.swiftuicookbook.com/qj8m4

- `accessibilityRotor(_:entries:)` documentation: https://x.swiftuicookbook.com/t2epx
- `AccessibilityFocusState` documentation: https://x.swiftuicookbook.com/7j24h
- `ScaledMetric` documentation: https://x.swiftuicookbook.com/4l9aj
- Build accessible apps with SwiftUI and UIKit (WWDC 2023): https://x.swiftuicookbook.com/qq6dw

7

Data Persistence and Local Storage

Every app I have shipped eventually had to persist something after the user closed it: a preference here, a list of records there, a draft still in progress. For years that meant reaching into a scattered toolbox, writing `UserDefaults` keys by hand for the small values, archiving objects with `NSCoding`, or setting up Core Data with its `.xcdatamodeld` file and a stack of boilerplate for anything larger. The tools have since consolidated and most of that boilerplate has fallen away, so the real question now is rarely how to persist data, but which of several good options fits the data in front of us.

This chapter walks that decision from the most convenient store to the most hands-on. We begin with the property wrappers that persist a value with almost no code, drop to the store beneath them when persistence outgrows a single view, step up to a full model layer once the app needs to query and relate records, and end with plain files when we want to own every byte. Each recipe builds a small, focused app, so the trade-offs between the approaches, how durable each one is, how well it scales, and how much machinery it asks for in return, stay easy to compare.

We start with a small settings app built around a single form. It stores a display name, a notifications toggle, a theme, and a sort order with `@AppStorage`, keeps a scratch note in `@SceneStorage` for per-scene restoration, and drives the app's light or dark appearance from the saved theme with `preferredColorScheme(_:)`. The second recipe is a game-night score tracker that uses `UserDefaults` directly: an `@Observable` store owns the roster, encodes it with `PropertyListEncoder` because an array is not a native property-list type, and seeds a sensible starting value with `register(defaults:)`.

For data the app creates, edits, and queries, we move up to SwiftData with a to-do app grouped by category. A `@Model` class describes a task, `.modelContainer(for:)` wires up the store, and a `@Query` keeps the list in sync, with its iOS 27 sectioned form grouping the rows by category. We insert and delete through the `@Environment(\.modelContext)`. The chapter ends with a journaling app that owns its storage outright: an `@Observable` store keeps every entry in a single JSON file under `URL.documentsDirectory`, converting entries with `JSONEncoder` and `JSONDecoder` and writing each save atomically.

By the end of this chapter, you should be able to reach for the right persistence tool without hesitating: a property wrapper when a value lives in a view, `UserDefaults` directly when it belongs to a model, SwiftData when the app needs to query and relate records, and a file in the file system when the data is one collection you would rather own end to end.

In this chapter, we will cover the following recipes:

- AppStorage and SceneStorage
- UserDefaults

- SwiftData
- Local Files

Technical requirements

The code in this chapter is based on Xcode 27.0 and iOS 27.0. You can download and install the latest version of Xcode from the App Store. All the code examples for this chapter can be found on GitHub at https://x.swiftuicookbook.com/ch07 .



Throughout this chapter, the preview canvas is the right place to check layout and interaction, but not persistence. Xcode runs a preview in its own copy of the app, installed on a separate simulator it keeps just for previews, so whatever a preview stores stays in that copy: it persists between refreshes of the preview canvas, it never reaches the app you run in Device Hub, and it is not necessarily visible to another preview, because Xcode can host each preview in a different copy. A preview also cannot be quit and relaunched, which is the one check that shows persistence working. So, the recipes use the preview canvas to build each view and Device Hub whenever a value should survive a relaunch. A preview that comes up already showing a value you entered earlier is behaving as designed, not reading stale state from your code.

AppStorage and SceneStorage

Almost every app needs to remember a handful of small choices between launches: a preferred theme, a display name, whether notifications are on. For years I stored these in `UserDefaults` by hand, reading a key when a screen appeared, writing it back when something changed, and then refreshing the interface myself so it reflected the new value. SwiftUI removes that last chore. The `@AppStorage` property wrapper binds a value directly to `UserDefaults`, so a write persists immediately and every view that reads the same key updates on its own.

In this recipe, we build **AppPreferences**, a small settings app with three tabs. A settings form stores a display name, a notifications toggle, a theme, and a default sort order with `@AppStorage`, alongside a launch counter that ticks up each time the app starts. A second tab keeps a scratch note backed by `@SceneStorage`, the per-scene companion to `@AppStorage`. A third, read-only tab reads the same keys back to show that the values are shared across the whole app. Because two of our settings are enumerations rather than strings or numbers, we lean on `RawRepresentable` to store them, and we drive the app's light or dark appearance from the stored theme with `preferredColorScheme(_:)`.

The two wrappers look almost identical in code, yet they persist in very different places, and choosing between them is the main idea of this recipe. `@AppStorage` is durable and app-wide: its values live in `UserDefaults` and survive relaunches and even app updates. `@SceneStorage` is for per-scene state restoration: the system saves and restores it as it manages a scene, but it is scoped to that scene and

meant for transient interface state rather than long-term storage. By the end, you should know which of the two to reach for, how to store custom types in either, and how a single stored value can steer the look of an entire app .

Getting ready

Create a new SwiftUI iOS app named **AppPreferences** .

How to do it...

We build the app one piece at a time. We start with the settings form, since that is where most of our stored values live, then add the scratch-note and summary tabs, and finish with the `TabView` that hosts all three. These are the steps:

1. Start with the settings form, the home for most of our stored values. Create `SettingsView.swift` in a `Views` folder. It opens with two `@AppStorage` properties, a display name and a launch counter, shown in a `Profile` section. The counter starts at zero; we wire it to increment when we assemble the app in step 9:

```
import SwiftUI
struct SettingsView: View {
  @AppStorage("username") private var username = ""
  @AppStorage("launchCount") private var launchCount = 0
  var body: some View {
    NavigationStack {
      Form {
        Section("Profile") {
          TextField("Username", text: $username)
          LabeledContent("Launches", value: "\(launchCount
        }
```

```
      }
      .navigationTitle("Settings")
    }
  }
}
#Preview {
  SettingsView()
}
```

Open the preview canvas and type a name: `@AppStorage` writes each keystroke to `UserDefaults` as it happens, with no save step. The launch count stays at zero until step 9 wires it up. The note in *Technical requirements* explains why this preview, like every other one in the chapter, keeps its own storage.

2. Add a `Preferences` section with a notifications switch. The new `@AppStorage` property goes with the other two above `body`, and the section follows `Profile` inside the `Form`. Its `Bool` reads `true` until the key exists, and toggling it writes straight to `UserDefaults`:

```
struct SettingsView: View {
  @AppStorage("username") private var username = ""
  @AppStorage("notificationsEnabled") private var notifica
  @AppStorage("launchCount") private var launchCount = 0
  var body: some View {
    NavigationStack {
      Form {
        // ... Profile section
        Section("Preferences") {
          Toggle("Notifications", isOn: $notificationsEnab
        }
      }
      .navigationTitle("Settings")
    }
```

```
  }
}
```

In the preview canvas, flip the switch a few times: each change is written the moment it happens.

3. Our next two settings are choices from a fixed set, a natural fit for an enum. Because `@AppStorage` cannot store an arbitrary type, we make the enum `RawRepresentable` with a `String` raw value, which both wrappers support. Create `AppTheme.swift` in a `Models` folder:

```
import SwiftUI
enum AppTheme: String, CaseIterable, Identifiable {
  case light
  case dark
  case system
  var id: String { rawValue }
  var displayName: String {
    rawValue.capitalized
  }
  var colorScheme: ColorScheme? {
    switch self {
    case .light: .light
    case .dark: .dark
    case .system: nil
    }
  }
}
```

The `colorScheme` property maps each case to a SwiftUI `ColorScheme`, with `nil` meaning "follow the system." For the label we reuse `rawValue.capitalized`, since `light`, `dark`, and `system` already read as the titles we want.

4. Add an `Appearance` section between `Profile` and `Preferences`, with a `Picker` bound to a new `@AppStorage` theme property, declared first among the stored properties. Storing an enum looks exactly like storing a string, because SwiftUI persists its raw value:

```
struct SettingsView: View {
  @AppStorage("appTheme") private var theme: AppTheme = .s
  // ... the username, notificationsEnabled, and launchCou
  var body: some View {
    NavigationStack {
      Form {
        // ... Profile section
        Section("Appearance") {
          Picker("Theme", selection: $theme) {
            ForEach(AppTheme.allCases) { theme in
              Text(theme.displayName).tag(theme)
            }
          }
        }
        Section("Preferences") {
          Toggle("Notifications", isOn: $notificationsEnab
        }
      }
      .navigationTitle("Settings")
    }
  }
}
```

   Back in the preview canvas, the picker is there, but choosing **Dark** does not recolor anything yet: the stored theme reaches the whole app only when step 9 hands it to `preferredColorScheme(_:)`.

5. Create the second enum, `SortOption.swift` in the `Models` folder. This one needs a label its raw value cannot provide:

```
import Foundation
enum SortOption: String, CaseIterable, Identifiable {
  case name
  case dateAdded
  case priority
  var id: String { rawValue }
  var displayName: String {
    switch self {
    case .name: "Name"
    case .dateAdded: "Date Added"
    case .priority: "Priority"
    }
  }
}
```

Here, we map each case to a display string with a `switch`, because `dateAdded` should read as **Date Added**, which `rawValue.capitalized` would render as "Dateadded." We come back to this contrast in *How it works*.

6. Add the default-sort `Picker` to the `Preferences` section, backed by a new `@AppStorage` property declared just above `launchCount`. The pattern is identical to the theme picker:

```
struct SettingsView: View {
  // ... the theme, username, and notificationsEnabled pro
  @AppStorage("sortOption") private var sortOption: SortOp
  @AppStorage("launchCount") private var launchCount = 0
  var body: some View {
    NavigationStack {
      Form {
        // ... Profile and Appearance sections
        Section("Preferences") {
          Toggle("Notifications", isOn: $notificationsEnab
          Picker("Default Sort", selection: $sortOption) {
            ForEach(SortOption.allCases) { option in
              Text(option.displayName).tag(option)
            }
```

```
            }
          }
        }
        .navigationTitle("Settings")
      }
    }
  }
```

The settings form is now complete, so check all three sections in the preview canvas before moving on.

7. Now, the second tab, a scratch note that shows off `@SceneStorage`. Create `DraftView.swift` in the `Views` folder. It reads like `@AppStorage`, but the value is tied to the scene rather than to `UserDefaults`:

```
import SwiftUI
struct DraftView: View {
  @SceneStorage("draftNote") private var draftNote = ""
  var body: some View {
    NavigationStack {
      VStack(alignment: .leading) {
        Text("This draft is saved per window and restored
          .font(.caption)
          .foregroundStyle(.secondary)
          .padding(.horizontal)
        TextEditor(text: $draftNote)
          .padding(.horizontal)
      }
      .navigationTitle("Draft")
    }
  }
}
#Preview {
  DraftView()
}
```

In the preview canvas, the editor accepts text like any `TextEditor`, but what sets `@SceneStorage` apart only shows in the running app, which we come back to in step 11.

8. Add a read-only summary tab. Create `InfoView.swift` in the `Views` folder. It declares the same `@AppStorage` keys as the settings form, which is the whole point: reading a key anywhere in the app gives the same value, so this tab mirrors the form with nothing passed between them:

```
import SwiftUI
struct InfoView: View {
  @AppStorage("appTheme") private var theme: AppTheme = .s
  @AppStorage("username") private var username = ""
  @AppStorage("notificationsEnabled") private var notifica
  @AppStorage("sortOption") private var sortOption: SortOp
  @AppStorage("launchCount") private var launchCount = 0
  var body: some View {
    NavigationStack {
      List {
        LabeledContent("Username",
                       value: username.isEmpty ? "Not set"
        LabeledContent("Theme", value: theme.displayName)
        LabeledContent("Default Sort", value: sortOption.d
        LabeledContent("Notifications",
                       value: notificationsEnabled ? "On"
        LabeledContent("Launches", value: "\(launchCount)"
      }
      .navigationTitle("Current Settings")
    }
  }
}
#Preview {
  InfoView()
}
```

9. Finally, assemble the three tabs. Open `ContentView.swift` and replace the template with a `TabView` that hosts the three views. Two `@AppStorage` properties above `body` read the theme and the launch count, so the view can drive the app's appearance from the stored theme with `preferredColorScheme(_:)` and bump the counter once when the app appears:

```
import SwiftUI
struct ContentView: View {
  @AppStorage("appTheme") private var theme: AppTheme = .s
  @AppStorage("launchCount") private var launchCount = 0
  var body: some View {
    TabView {
      SettingsView()
        .tabItem { Label("Settings", systemImage: "gear")
      DraftView()
        .tabItem { Label("Draft", systemImage: "note.text"
      InfoView()
        .tabItem { Label("Info", systemImage: "info.circle
    }
    .preferredColorScheme(theme.colorScheme)
    .onAppear { launchCount += 1 }
  }
}
#Preview {
  ContentView()
}
```

   We keep the `TabView` deliberately plain here; its styling and richer tab features are covered in the *TabViews and Liquid Glass* section of *Chapter 11*.

10. Run the app in Device Hub, because the check that matters here is quitting and relaunching, which the preview canvas cannot do. Type a name, switch the theme between **Light** and **Dark** and

watch the whole app recolor, then open the **Info** tab to see the same values reflected back. Quit the app from the app switcher and launch it again: the settings are still there, with the launch counter one higher.

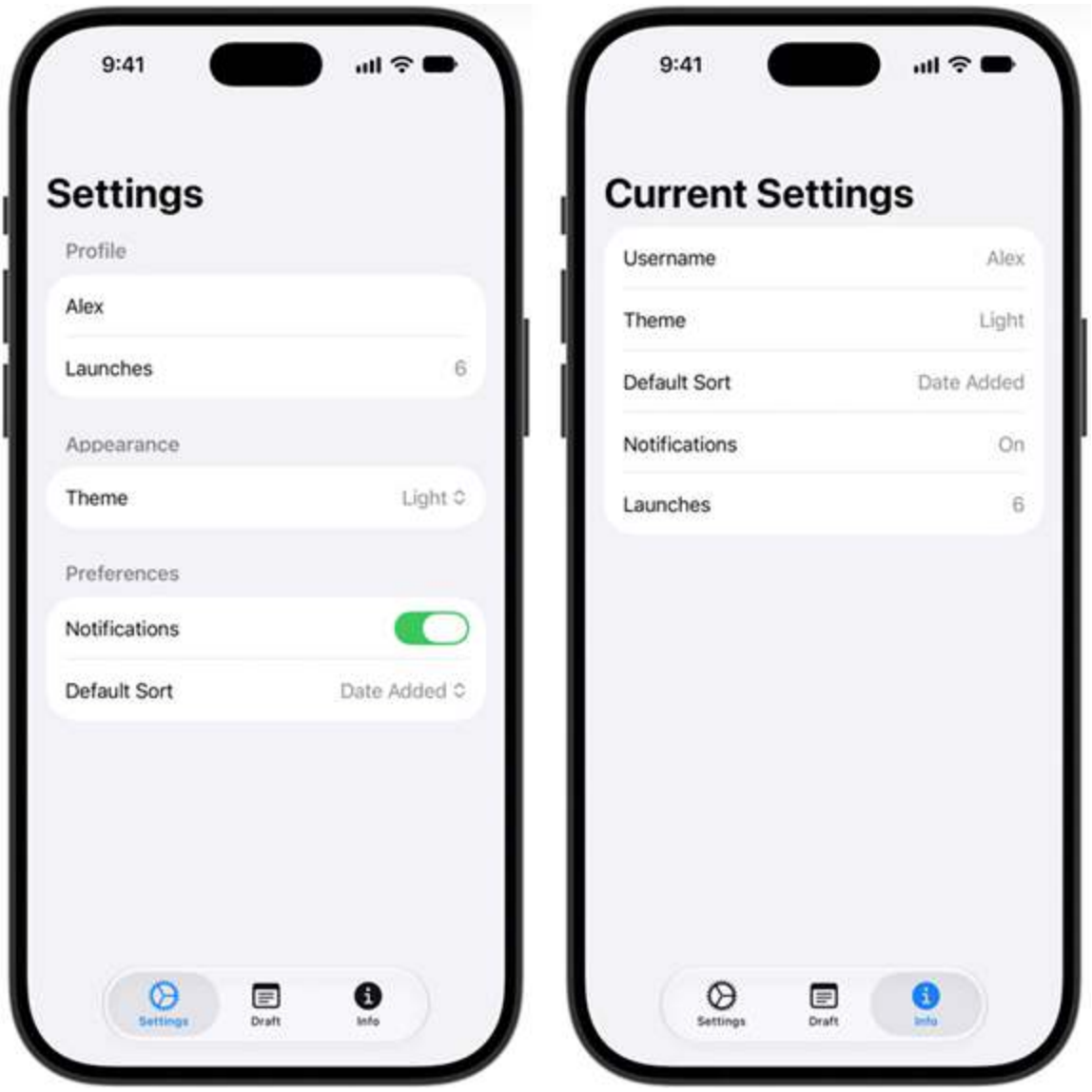


Figure 7.1: The settings form and the read-only Info tab share the same stored values

11. Now, put `@SceneStorage` through the interruption it exists for. Type a few words in the **Draft** tab, return to the Home Screen so the app moves to the background, stop the app with the stop

button in Xcode, and run it again: the draft comes back, restored with the scene. That is the distinction in practice. `@AppStorage` holds data the app keeps for good, while `@SceneStorage` holds interface state the system hands back when it restores a scene, and it is the system, not your code, that decides when that state is kept or discarded. On an iPad, where the app can open more than one window, a new window opens as a copy of the current one, draft included, while a theme change in any window rec olors them all.

How it works...

Every setting in our form is a property wrapped with `@AppStorage`, and each one is a thin, reactive view onto `UserDefaults`. When we bind `$username` to a `TextField` or `$notificationsEnabled` to a `Toggle`, editing the control writes the new value to the default storage right away, and SwiftUI invalidates every view that reads that key so the user interface stays in sync with the data. We never touch `UserDefaults` directly and never refresh anything by hand. Out of the box the wrapper supports the property-list types, `Bool`, `Int`, `Double`, `String`, `URL`, and `Data`, along with their optionals.

Our theme and sort settings are enum cases, and `UserDefaults` cannot store those directly. `RawRepresentable` is the bridge: when a type has a `String` or `Int` raw value, SwiftUI stores that raw value and rebuilds the case when it reads the key back, which is why `@AppStorage("appTheme")` needs no custom persistence code. The interesting part is the label. `AppTheme` can reuse `rawValue.capitalized`, because `light`, `dark`, and `system` already produce the titles we want.

`SortOption` needs a `switch`, because `dateAdded` must appear as **Date Added** rather than "Dateadded." Use the raw value only when the stored key already reads like display text; switch to explicit labels as soon as the label diverges from the key, whether for a multi-word name or a localized string. For a structured value rather than an enum, you can make a `Codable` type conform to `RawRepresentable` by encoding it to a JSON string, but implement `encode(to:)` and `init(from:)` yourself so synthesized `Codable` does not recurse through `rawValue`.

`@SceneStorage` in the draft tab looks almost identical to `@AppStorage`, and that resemblance is the trap. It participates in the system's state restoration instead of writing to `UserDefaults`. When a scene moves to the background, SwiftUI saves its `@SceneStorage` values, and it restores them when the system brings that scene back, including after it has quietly terminated the app to reclaim memory, which is what step 11 reproduced by stopping the app from Xcode. Because the system decides when that saved state is kept or thrown away, never rely on `@SceneStorage` for anything the user expects to find again. The other differences that matter are scope and durability. `@SceneStorage` is per scene, so each iPadOS window holds its own draft, starting from a copy of the window it was opened from, while `@AppStorage` shares one value across all of them. It is also less durable: updating the app clears it, while `@AppStorage` values live in the app's container and survive an update. Use `@SceneStorage` to restore transient interface state when the user returns to a screen, such as a selected row, a half-typed note, or a scroll position. It works only inside views, since it has no store a model can read. Use `@AppStorage` or a file for data that must outlive the app's installation.

The `Info` tab makes the app-wide nature of `@AppStorage` concrete. It declares the very same keys as the settings form, holds no state of its own, and still shows the current values, because both views read one shared store. The stored theme reaches further still: `ContentView` reads `appTheme` and hands its `colorScheme` to `preferredColorScheme(_:)`, so changing the theme in the form recolors every tab at once. The launch counter is the simplest case of all, a stored `Int` we raise in `onAppear`, the kind of one-line persistence that, in UIKit, used to need a few lines of `UserDefaults` plumbing. The price of the convenience is the key strings repeated across three views, where a typo in one of them silently reads a different key. We keep the literals here so each wrapper stays self-explanatory. The next recipe gathers its keys into a `DefaultsKey` enum of constants, and the same enum works with `@AppStorage`, which accepts any `String` as its key.

Both wrappers are built for small values, and neither is secure. `UserDefaults` keeps its contents as plain text in the app's container. When we need the full `UserDefaults` API, including reading and writing from a model rather than a view, we use it directly in the *UserDefaults* section.



`@AppStorage` is backed by `UserDefaults`, which is not encrypted. Use it for preferences and other non-sensitive state, and store credentials or tokens in th e Keychain instead.

See also

- `AppStorage` documentation: https://x.swiftuicookbook.com/pf7ed
- `SceneStorage` documentation: https://x.swiftuicookbook.com/ej82j
- `UserDefaults` documentation: https://x.swiftuicookbook.com/14exi
- `RawRepresentable` documentation: https://x.swiftuicookbook.com/hs0fa
- `preferredColorScheme(_:)` documentation: https://x.swiftuicookbook.com/qvlji
- *Restoring your app's state with SwiftUI* article: https://x.swiftuicookbook.com/te25k

UserDefaults

In the *AppStorage and SceneStorage* section we stored a handful of settings using `@AppStorage`, a property wrapper backed by `UserDefaults`. That wrapper is the right tool as long as the value lives inside a view. Apps outgrow that quickly. The moment persistence belongs to a model, a service, or any code that runs before a view exists, we work with `UserDefaults` directly: the same store `@AppStorage` wraps, but with its full API and no requirement to sit inside a `View`.

In this recipe, we build a generic game-night score tracker. It is deliberately not tied to one sport: each player carries a list of integer round scores along with a running total and an average, so the same model fits a card game, a board game, or any pastime scored in points across rounds rather than as a single two-team match. An `@Observable` class named **ScoreboardStore** owns the roster and reads

and writes it straight through `UserDefaults.standard`. Because an array of players is not one of the property-list types `UserDefaults` stores natively, we encode it to `Data` with `PropertyListEncoder` and read it back with `PropertyListDecoder`. A **Show Averages** toggle writes a plain `Bool` the moment it changes using a `didSet` observer. We seed that switch with a sensible starting value through `register(defaults:)` at launch, and a **Reset** button clears the saved roster with `removeObject(forKey:)`.

What ties this recipe together is where the storage logic lives. Here, our `UserDefaults` access sits in a model that the views simply observe, which is how most production apps keep it in one place instead of scattering keys across the interface. By the end, you should be comfortable storing custom `Codable` values in `UserDefaults`, registering fallback defaults, and removing keys, and clear on when a view-level wrapper is enough and when to work with the store directly.

Getting ready

Create a new SwiftUI iOS app named **ScoreTracker**.

How to do it...

We build the app in layers, starting with the data and finishing with the views. We model a single player, then create the store that loads and saves the whole roster, and only once persistence works do we add the views that show it. That order keeps the part that matters most, getting data into and out of storage correctly, in front of us first. These are the steps:

1. Start with the data we want to persist. Create `Player.swift` in a `Models` folder. A player has a name and a list of round scores, conforms to `Codable` so the system can turn it into storable data, and conforms to `Identifiable` so SwiftUI can track it in a list. Two computed properties derive the total and the average on the fly:

```
import Foundation
struct Player: Codable, Identifiable {
  var id = UUID()
  var name: String
  var scores: [Int] = []
  var totalScore: Int {
    scores.reduce(0, +)
  }
  var averageScore: Double {
    scores.isEmpty ? 0 : Double(totalScore) / Double(score
  }
}
```

2. Now, the heart of the recipe. Create `ScoreboardStore.swift` in the `Models` folder. We keep our storage keys together in a small `DefaultsKey` enum so the rest of the code never repeats a raw string, then build an `@Observable` class that holds the roster and knows how to persist it. The private `save()` encodes the players to `Data` with `PropertyListEncoder`, and `loadPlayers()` decodes them back, returning an empty array when nothing is stored yet:

```
import SwiftUI
enum DefaultsKey {
  static let players = "savedPlayers"
  static let showAverages = "showAverages"
```

```
}
@Observable
final class ScoreboardStore {
  var players: [Player] = []
  init() {
    players = Self.loadPlayers()
  }
  private func save() {
    let encoder = PropertyListEncoder()
    if let data = try? encoder.encode(players) {
      UserDefaults.standard.set(data, forKey: DefaultsKey.
    }
  }
  private static func loadPlayers() -> [Player] {
    let decoder = PropertyListDecoder()
    guard let data = UserDefaults.standard.data(forKey: De
          let players = try? decoder.decode([Player].self,
    else { return [] }
    return players
  }
}
```

3. Add the one preference the app remembers, a switch for showing averages instead of totals. Unlike the roster, this is a single `Bool`, so it does not need encoding: a `didSet` observer writes it straight to `UserDefaults` whenever it changes. We also read its saved value back in `init`:

```
var players: [Player] = []
var showAverages: Bool {
  didSet {
    UserDefaults.standard.set(showAverages, forKey: Defaul
  }
}
init() {
  showAverages = UserDefaults.standard.bool(forKey: Defaul
  players = Self.loadPlayers()
}
```

4. With persistence in place, add the methods that change the roster. Each one mutates the `players` array and then calls `save()`, so every edit is persisted as it happens, with no separate save step for the rest of the app to remember:

```
func addPlayer(name: String) {
  players.append(Player(name: name))
  save()
}
func addScore(_ score: Int, to player: Player) {
  guard let index = players.firstIndex(where: { $0.id == p
  else { return }
  players[index].scores.append(score)
  save()
}
func removePlayers(at offsets: IndexSet) {
  players.remove(atOffsets: offsets)
  save()
}
```

5. Add one more method to wipe the scoreboard. Clearing the array empties the list, and `removeObject(forKey:)` deletes the stored data so the roster does not reappear on the next launch:

```
func resetAll() {
  players.removeAll()
  UserDefaults.standard.removeObject(forKey: DefaultsKey.p
}
```

6. Open `ScoreTrackerApp.swift` and register a default for the averages switch. `register(defaults:)` runs before any view appears and supplies fallback values for keys that have never

been set, so the toggle starts in the on position on a fresh install instead of defaulting to `false`:

```
import SwiftUI
@main
struct ScoreTrackerApp: App {
  init() {
    UserDefaults.standard.register(defaults: [DefaultsKey.
  }
  var body: some Scene {
    WindowGroup {
      ContentView()
    }
  }
}
```

7. Now, the views. Create `PlayerRowView.swift` in a `Views` folder for a single row. It shows the player's name with either their average or their total on the trailing edge, depending on the switch we just persisted:

```
import SwiftUI
struct PlayerRowView: View {
  let player: Player
  let showAverage: Bool
  var body: some View {
    HStack {
      Text(player.name)
      Spacer()
      if showAverage {
        Text("Avg: \(player.averageScore, specifier: "%.1f
          .foregroundStyle(.secondary)
      } else {
        Text("Total: \(player.totalScore)")
          .foregroundStyle(.secondary)
      }
    }
```

```
  }
}
#Preview {
  List {
    PlayerRowView(player: Player(name: "Alex", scores: [10
                  showAverage: true)
    PlayerRowView(player: Player(name: "Sam", scores: [12,
                  showAverage: false)
  }
}
```

The preview shows both variants of the row, one with an average and one with a total, so we can check the layout before there is a store to feed it.

8. Create `PlayerDetailView.swift` in the `Views` folder for a single player's scores. Rather than holding a copy of the player, it looks the player up in the store by `id` every time `body` runs, so a newly recorded score appears at once. It lists each round, offers a field to add another score, and shows a short summary:

```
import SwiftUI
struct PlayerDetailView: View {
  let playerID: Player.ID
  let store: ScoreboardStore
  @State private var newScore = ""
  private var player: Player? {
    store.players.first { $0.id == playerID }
  }
  var body: some View {
    if let player {
      List {
        Section("Scores") {
          if player.scores.isEmpty {
            Text("No scores yet.")
              .foregroundStyle(.secondary)
          } else {
```

```
            ForEach(Array(player.scores.enumerated()), id:
              index, score in
              LabeledContent("Round \(index + 1)", value:
            }
          }
        }
        Section("Add Score") {
          HStack {
            TextField("Score", text: $newScore)
              .keyboardType(.numberPad)
            Button("Add") {
              if let score = Int(newScore) {
                store.addScore(score, to: player)
                newScore = ""
              }
            }
            .disabled(Int(newScore) == nil)
          }
        }
        Section("Summary") {
          LabeledContent("Total", value: "\(player.totalSc
          LabeledContent("Average",
                         value: String(format: "%.1f", pla
          LabeledContent("Games Played", value: "\(player.
        }
      }
      .navigationTitle(player.name)
    }
  }
}
#Preview {
  let store = ScoreboardStore()
  store.players = [Player(name: "Alex", scores: [10, 25, 1
  return NavigationStack {
    PlayerDetailView(playerID: store.players[0].id, store:
  }
}
```

Its preview seeds a store with one player, so in the preview canvas we can add a score and watch the **Summary** section

update.

9. With the pieces ready, assemble the main screen. Open `ContentView.swift` and replace the template. The view creates the store in a `@State` property, the one place the app instantiates `ScoreboardStore`, and its `body` wraps a `List` in a `NavigationStack`. The first section holds the **Show Averages** toggle bound to the store, and the second lists the players, each a `NavigationLink` to its detail view, with swipe-to-delete wired to the store:

```
import SwiftUI
struct ContentView: View {
  @State private var store = ScoreboardStore()
  var body: some View {
    NavigationStack {
      List {
        Section {
          Toggle("Show Averages", isOn: $store.showAverage
        }
        Section("Players") {
          if store.players.isEmpty {
            Text("No players yet. Tap + to add one.")
              .foregroundStyle(.secondary)
          } else {
            ForEach(store.players) { player in
              NavigationLink {
                PlayerDetailView(playerID: player.id, stor
              } label: {
                PlayerRowView(player: player, showAverage:
              }
            }
            .onDelete(perform: store.removePlayers)
          }
        }
      }
      .navigationTitle("Score Tracker")
    }
  }
```

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

In the preview canvas the list is empty until the next step adds a way to create players, but the **Show Averages** toggle already persists, since the store writes it on every change.

10. Add a way to create players. Two new `@State` properties, declared below the store, drive a **New Player** alert, and a toolbar button opens it. Both modifiers attach to the `List`, after `navigationTitle`. Tapping **Add** trims the entry and asks the store to append the new player, which saves the roster automatically:

```
struct ContentView: View {
  @State private var store = ScoreboardStore()
  @State private var showingAddPlayer = false
  @State private var newPlayerName = ""
  var body: some View {
    NavigationStack {
      List {
        // ... the Show Averages and Players sections
      }
      .navigationTitle("Score Tracker")
      .toolbar {
        ToolbarItem(placement: .topBarTrailing) {
          Button("Add Player", systemImage: "plus") {
            showingAddPlayer = true
          }
        }
      }
      .alert("New Player", isPresented: $showingAddPlayer)
        TextField("Name", text: $newPlayerName)
        Button("Add") {
          let name = newPlayerName.trimmingCharacters(in:
          if !name.isEmpty {
```

```
            store.addPlayer(name: name)
          }
          newPlayerName = ""
        }
        Button("Cancel", role: .cancel) {
          newPlayerName = ""
        }
      } message: {
        Text("Enter the player's name.")
      }
    }
  }
}
```

11. Finally, make **Reset** safe. Because clearing the board cannot be undone, a third `@State` property drives a confirmation alert, a second `ToolbarItem` joins the first inside the existing `toolbar`, and the new alert attaches right after the **New Player** one. The destructive **Delete All Players** action calls `resetAll()`, while **Cancel** leaves the roster untouched. The button is disabled while there is nothing to clear:

```
struct ContentView: View {
  // ... the store, showingAddPlayer, and newPlayerName pr
  @State private var showingResetConfirmation = false
  var body: some View {
    NavigationStack {
      List {
        // ... the Show Averages and Players sections
      }
      .navigationTitle("Score Tracker")
      .toolbar {
        // ... the Add Player item
        ToolbarItem(placement: .topBarLeading) {
          Button("Reset", role: .destructive) {
            showingResetConfirmation = true
          }
```

```
            .disabled(store.players.isEmpty)
        }
      }
      // ... the New Player alert
      .alert("Reset Scoreboard?", isPresented: $showingRes
        Button("Delete All Players", role: .destructive) {
        Button("Cancel", role: .cancel) { }
      } message: {
        Text("This permanently removes every player and th
      }
    }
  }
}
```

12. Run the app in Device Hub, since the point is to quit and relaunch. Add a few players, open one and record a couple of rounds, and flip **Show Averages** to watch every row switch between its total and its average. Quit the app from the app switcher and launch it again: the roster is exactly where you left it. Tap **Reset** , confirm, and the board clears for good.



Figure 7.2: The roster on the main screen and a sing le player's recorded rounds

How it works...

Every bit of persistence in the app runs through `ScoreboardStore`, an `@Observable` class that talks to `UserDefaults.standard` directly. That is the difference from the previous recipe: `@AppStorage` can only live inside a view, while a store is a plain object the rest of the app can own, pass around, and test. The views hold it as a `@State` value and

read its `players` array, and because the class is `@Observable`, changing that array refreshes the list with no manual notification. One object owns the keys and the read and write logic, and the interface simply observes it. The Observation framework behind `@Observable` gets a full treatment in the *Observation Framework* section of *Chapter 5*.

`UserDefaults` natively stores only the property-list types, numbers, strings, dates, data, and arrays or dictionaries of those, so our `[Player]` array cannot go in as-is. The bridge is `Codable`: `PropertyListEncoder` turns the array into a `Data` value that `UserDefaults` accepts, and `PropertyListDecoder` reconstructs it on the way out. We reach for the property-list encoder rather than `JSONEncoder` because `UserDefaults` is itself backed by a property list, so the data round-trips in its native format. Every method that changes the roster calls `save()` right away, and `loadPlayers()` returns an empty array whenever the key is missing or cannot be decoded, which is exactly what we want on a first launch. One detail is worth a note: `remove(atOffsets:)` comes from SwiftUI rather than Foundation, so the store imports `SwiftUI`, which is why a file that otherwise reads like a plain model starts with that import.

The averages switch shows the simpler side of `UserDefaults`. A `Bool` is already a property-list type, so `showAverages` skips encoding entirely, and its `didSet` writes the new value with `set(_:forKey:)` the instant it changes. Reading it back is where `register(defaults:)` earns its place. The `bool(forKey:)` method returns `false` for a key that was never set, which would leave the toggle off on a fresh install, so registering a default of `true` in the app's `init` gives the key a fallback before the store ever reads it, since the app initializes before any view. Registered defaults are not persisted; they sit in front of the

stored values as a fallback layer. Keeping both key strings in the `DefaultsKey` enum means the `register` call and every read share one constant, which rules out the classic bug where a mistyped key silently reads back nothing.

Resetting the board is two actions in one: `players.removeAll()` empties the in-memory array so the list updates at once, and `removeObject(forKey:)` deletes the stored data so the roster does not return on the next launch. The detail screen leans on that same single source of truth in a quieter way. Instead of capturing a copy of the player, `PlayerDetailView` looks it up in the store by `id` each time its `body` runs, so a score added through the store appears immediately and the two screens never disagree. Keep in mind that `UserDefaults` keeps its contents as plain text and is meant for small, preference-sized values. For a large or relational data set, reach for the *SwiftData* section, and for documents stored as fi les, the *Local Files* section.

See also

- `UserDefaults` documentation: https://x.swiftuicookbook.com/14exi
- `PropertyListEncoder` documentation: https://x.swiftuicookbook.com/jmfbn
- `PropertyListDecoder` documentation: https://x.swiftuicookbook.com/wemnm
- `register(defaults:)` documentation: https://x.swiftuicookbook.com/u60hq
- `removeObject(forKey:)` documentation: https://x.swiftuicookbook.com/s7bj7

- `Codable` documentation: https://x.swiftuicookbook.com/sdzlw

SwiftData

The last two recipes persisted small values by hand, reading and writing keys ourselves. That works for preferences, but it does not scale to a list of records that the app creates, edits, and queries. For that we want a real model layer, and SwiftData is Apple's modern answer: we declare a Swift class as a model, and the framework gives us the storage, the change tracking, and the SwiftUI updates without a `.xcdatamodeld` file or the Core Data boilerplate it used to take.

In this recipe, we build **TaskManager** , a to-do app that keeps its tasks grouped into categories. A `@Model` class describes a task, `.modelContainer(for:)` wires up the store, and a `@Query` fetches the tasks and keeps the list in sync as they change. We insert and delete through the `@Environment(\.modelContext)` , and we group the list into per-category sections with the sectioned `@Query` that is new in iOS 27. On that list, we can mark a task done, expand a row to see when it was added, and swipe to delete. Adding a task is a `Form` presented in a sheet.

SwiftData has matured since its introduction, and iOS 27 adds the piece this recipe leans on: the sectioned query drives the grouped list directly. By the end, you should be comfortable defining a model, querying it reactively, sectioning the results, and knowing where SwiftData ends a nd your own view state begins.

Getting ready

Create a new SwiftUI iOS app named **TaskManager** . When Xcode offers a storage option in the project template, choose **None** , not **SwiftData** . The **SwiftData** template scaffolds a sample model and a container we would only delete and rewrite, and setting them up by hand keeps every piece of the SwiftDat a stack visible as we build it.

How to do it...

We build the app from the model up: first the task model, then the container that stores it, then the views that show and edit it, finishing with the sectioned list that ties them together. These are the steps:

1. Start with the model. Create `TaskItem.swift` in a `Models` folder, and add `import SwiftData` next to Foundation. The `@Model` macro turns a plain Swift class into a persisted type: SwiftData stores every stored property, tracks changes, and makes the class observable so views refresh when a task changes. Notice that it is a class, not a struct like the models in the previous two recipes, and that is not optional: SwiftData tracks each task by identity, so when a row toggles `isDone` it has to change the very object the store holds, where a struct would hand the row a copy and the store would never see the change. We keep `category` as a `String` because we will group by it shortly; the rest of the properties describe the task and when it was created:

```
import Foundation
import SwiftData

```

```
@Model
final class TaskItem {
  var title: String
  var isDone: Bool
  var priority: Int
  var category: String
  var createdAt: Date
  init(title: String, category: String, priority: Int = 0)
    self.title = title
    self.category = category
    self.priority = priority
    self.isDone = false
    self.createdAt = Date()
  }
}
```

2. Tell the app where its data lives. In `TaskManagerApp.swift`, add `import SwiftData` and attach `.modelContainer(for: TaskItem.self)` to the window group. That one modifier creates the on-device store for `TaskItem`, builds the schema from the model, and injects a model context into the environment for every view to use:

```
import SwiftUI
import SwiftData
@main
struct TaskManagerApp: App {
  var body: some Scene {
    WindowGroup {
      ContentView()
    }
    .modelContainer(for: TaskItem.self)
  }
}
```

3. Create `TaskRowView.swift` in a `Views` folder to show one task in a row: a circle that toggles the done state, the title, any priority markers, and a chevron that expands the row to reveal when the task was added. The expansion is purely visual, so it lives in a `@State` property on the view, not on the model. Both controls use `.borderless`. In a `List` row, a standard-styled button takes the whole row as its tap target, so two of them would fire on every tap; `.borderless` scopes each button to its own frame, keeping the circle and the chevron independently tappable:

```
import SwiftUI
struct TaskRowView: View {
  let task: TaskItem
  @State private var isExpanded = false
  var body: some View {
    VStack(alignment: .leading, spacing: 4) {
      HStack {
        Button {
          task.isDone.toggle()
        } label: {
          Image(systemName: task.isDone ? "checkmark.circl
            .foregroundStyle(task.isDone ? .green : .secon
        }
        .buttonStyle(.borderless)
        Text(task.title)
          .strikethrough(task.isDone)
        Spacer()
        if task.priority > 0 {
          HStack(spacing: 1) {
            ForEach(0..<task.priority, id: \.self) { _ in
              Image(systemName: "exclamationmark")
                .font(.caption2)
                .foregroundStyle(.orange)
            }
          }
        }
        Button {
          isExpanded.toggle()
```

```
        } label: {
          Image(systemName: isExpanded ? "chevron.up" : "c
            .foregroundStyle(.secondary)
        }
        .buttonStyle(.borderless)
      }
      if isExpanded {
        let date = task.createdAt.formatted(date: .abbrevi
                                            time: .shorten
        Text("Added \(date)")
          .font(.caption)
          .foregroundStyle(.secondary)
      }
    }
  }
}
#Preview {
  TaskRowView(task: TaskItem(title: "Buy groceries", categ
                             priority: 2))
}
```

In the preview canvas, tap the circle to toggle the checkmark and the chevron to reveal the date.



The row's expanded state is a `@State` property, not a property on `TaskItem`. It is tempting to mark such throwaway state `@Transient` so SwiftData skips persisting it, but `@Transient` also opts a property out of change tracking: the `@Model` macro wires observation only for stored, persisted properties, so toggling a `@Transient` value never refreshes the view. Keep ephemeral interface state, like whether a row is expanded,

> in `@State`, and reach for `@Transient` only for non-observed runtime values.

4. Create `AddTaskView.swift` for the form to add new tasks. The form collects a title, a category, and a priority, then inserts a `TaskItem` into the model context when the user taps the checkmark button to save. We never call `save()`: the model context that `.modelContainer` sets up has autosave enabled, so SwiftData writes the new task to the store for us. The context and a `dismiss` action both come from the environment:

```
import SwiftUI
import SwiftData
struct AddTaskView: View {
  @Environment(\.modelContext) private var context
  @Environment(\.dismiss) private var dismiss
  @State private var title = ""
  @State private var category = "Work"
  @State private var priority = 0
  private let categories = ["Work", "Home", "Errands"]
  var body: some View {
    NavigationStack {
      Form {
        Section("Task") {
          TextField("Title", text: $title)
          Picker("Category", selection: $category) {
            ForEach(categories, id: \.self) { category in
              Text(category)
            }
          }
        }
        Section("Priority") {
          Picker("Priority", selection: $priority) {
            Text("None").tag(0)
            Text("Low").tag(1)
            Text("Medium").tag(2)
            Text("High").tag(3)
```

```
            }
            .pickerStyle(.segmented)
          }
        }
        .navigationTitle("New Task")
        .toolbar {
          ToolbarItem(placement: .cancellationAction) {
            Button("Cancel", systemImage: "xmark") { dismiss
          }
          ToolbarItem(placement: .confirmationAction) {
            Button("Save", systemImage: "checkmark") {
              let task = TaskItem(title: title, category: ca
                                  priority: priority)
              context.insert(task)
              dismiss()
            }
            .disabled(title.isEmpty)
          }
        }
      }
    }
  }
  #Preview {
    AddTaskView()
      .modelContainer(for: TaskItem.self, inMemory: true)
  }
```

Its preview attaches an in-memory container, so tapping the checkmark in the preview canvas inserts into a throwaway store rather than the app's real one.

5. Now, the main screen, and the heart of the recipe. Open `ContentView.swift` and replace the template. The view uses the new iOS 27 sectioned `@Query`, sorting the tasks by title and grouping them by category. Declaring the property as `SectionedResults<TaskItem, String>` rather than `[TaskItem]` tells the query to hand back the grouped results directly: each

`ResultsSection` carries its category string as its `title` and the tasks in that group, so the `List` renders one `Section` per category. Swipe-to-delete maps the row offsets back to the tasks in that section before deleting them through the model context:

```
import SwiftUI
import SwiftData
struct ContentView: View {
  @Query(sort: \TaskItem.title, sectionBy: \TaskItem.categ
  private var sections: SectionedResults<TaskItem, String>
  @Environment(\.modelContext) private var context
  var body: some View {
    NavigationStack {
      List {
        ForEach(sections) { section in
          Section(section.title) {
            ForEach(section) { task in
              TaskRowView(task: task)
            }
            .onDelete { offsets in
              let items = Array(section)
              for index in offsets {
                context.delete(items[index])
              }
            }
          }
        }
      }
      .navigationTitle("Tasks")
    }
  }
}
#Preview {
  ContentView()
    .modelContainer(for: TaskItem.self, inMemory: true)
}
```

The preview canvas shows only an empty list for now, since its in-memory container starts with no tasks. Step 7 gives it something to show.



Because `sectionBy:` groups by a `String` key path, `category` is a plain `String` rather than an enum. SwiftData can persist a `Codable` enum, but you cannot section directly by an enum-typed property, so a value you intend to group, sort, or filter on is usually best stored as a primitive type.

6. Give `ContentView` its finishing touches. A `@State` flag, declared with the other properties above `body`, drives the add-task sheet. A toolbar button presents it, and an overlay shows a `ContentUnavailableView` while there are no tasks yet, all three attached to the `List` after `navigationTitle`:

```
struct ContentView: View {
  // ... the sections query and the context property
  @State private var showingAdd = false
  var body: some View {
    NavigationStack {
      List {
        // ... the sectioned ForEach
      }
      .navigationTitle("Tasks")
      .toolbar {
        ToolbarItem(placement: .topBarTrailing) {
          Button("Add Task", systemImage: "plus") {
            showingAdd = true
          }
        }
```

```
        }
        .overlay {
          if sections.isEmpty {
            ContentUnavailableView("No Tasks", systemImage:
              description: Text("Tap + to add your first tas
          }
        }
        .sheet(isPresented: $showingAdd) {
          AddTaskView()
        }
      }
    }
  }
```

7. Give the preview canvas something to show. A `@Query` reads from whatever model container the environment provides, and the preview we already have hands it an empty in-memory store, so the preview canvas shows only the empty state. Rather than seed a container inside one preview, we build it once in a `PreviewModifier`, the reusable mock-data type from the *Previews with Mock Data* section of *Chapter 3*, so any preview in the project can borrow the same sample tasks. Add `SampleTasks` below `ContentView`. Its `makeSharedContext()` creates the container with `ModelConfiguration(isStoredInMemoryOnly: true)`, so the sample data never touches the real on-device database, and inserting each task into the container's `mainContext` is all the `@Query` needs to find them. The `body(content:context:)` method attaches that container to whatever the preview shows, and the `PreviewTrait` extension gives the modifier a name we can pass to a second preview:

```
struct SampleTasks: PreviewModifier {
  static func makeSharedContext() throws -> ModelContainer
```

```
    let config = ModelConfiguration(isStoredInMemoryOnly:
    let container = try ModelContainer(for: TaskItem.self,
                                       configurations: con
    let context = container.mainContext
    let done = TaskItem(title: "Pick up parcel", category:
    done.isDone = true
    let samples = [
      TaskItem(title: "Email the client", category: "Work"
      TaskItem(title: "Prepare slides", category: "Work",
      TaskItem(title: "Buy groceries", category: "Errands"
      done,
      TaskItem(title: "Water the plants", category: "Home"
    ]
    for task in samples { context.insert(task) }
    return container
  }
  func body(content: Content, context: ModelContainer) ->
    content
      .modelContainer(context)
  }
}
extension PreviewTrait where T == Preview.ViewTraits {
  @MainActor static var sampleTasks: Self = .modifier(Samp
}
#Preview("Sample data", traits: .sampleTasks) {
  ContentView()
}
```

8. Run the app in Device Hub, since checking persistence means quitting and relaunching. Add a few tasks in different categories and watch them fall into sections. Tap a circle to complete a task, tap a chevron to see when it was added, and swipe a row to delete it. Quit the app from the app switcher and launch it again: every task is still there, grouped exactly as before, because SwiftData persisted them for us.

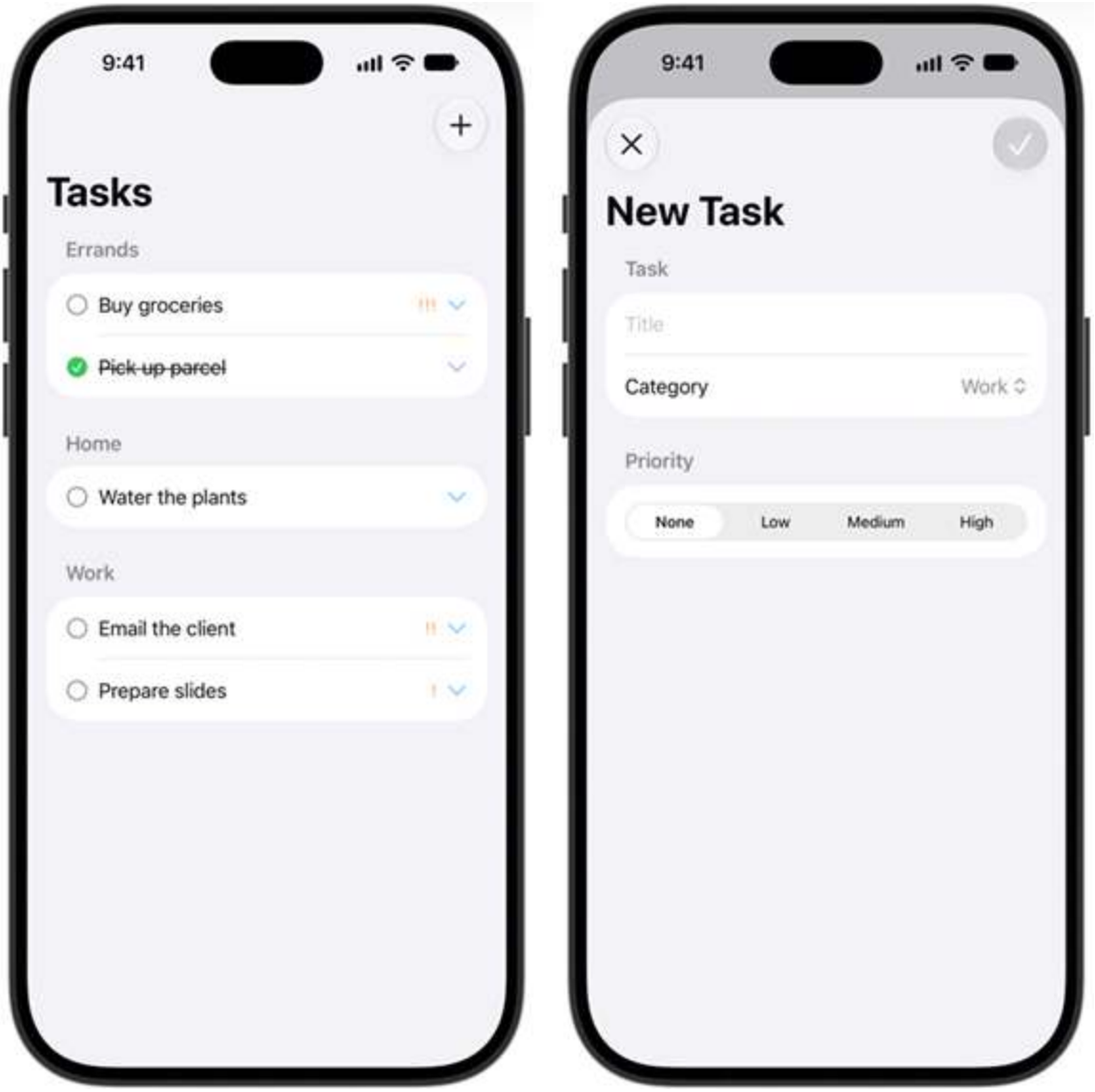


Figure 7.3: The gr ouped task list and the New Task form

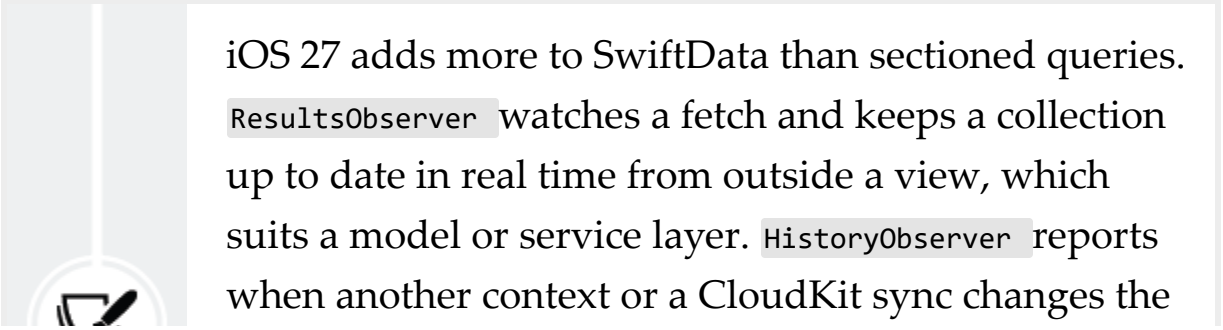
How it works...

Every task is a n instance of `TaskItem`, and the `@Model` macro is what makes it persistent. It rewrites the class to conform to SwiftData's `PersistentModel` and to the Observation framework, so the same object SwiftData saves is also the one SwiftUI watches for changes. The store comes from `.modelContainer(for: TaskItem.self)` on the app's

window group, which builds the schema from the model and places a `ModelContext` in the environment. That context is the workspace: `AddTaskView` reads it to `insert` a new task and `ContentView` reads it to `delete` one, and in both cases SwiftData autosaves, so we never call `save()` by hand.

`@Query` is why the task list stays current without a fetch method in our view. It reads `TaskItem` objects from the model context already in the environment, applies the `SortDescriptor`, and runs again as the data changes. Because `AddTaskView` inserts into that same context, the new task appears as soon as the sheet dismisses, with no separate reload step. In the preview canvas, the `SampleTasks` modifier stands in for the app's `.modelContainer(for:)`, attaching an in-memory container that the query reads exactly as it would the on-device store.

Grouping the list is where iOS 27 comes in. A sectioned `@Query` changes the shape of the query's value: declared as `SectionedResults<TaskItem, String>`, the property is a collection of `ResultsSection` values rather than a flat `[TaskItem]`. Each section carries its category string as `title` and is itself a collection of the tasks in that group, so `ForEach(sections)` with a `Section(section.title)` renders one titled group per category, and `sections.isEmpty` tells the overlay when there is nothing to show. Deleting maps the row offsets back to tasks inside the section before calling `context.delete`.



iOS 27 adds more to SwiftData than sectioned queries. `ResultsObserver` watches a fetch and keeps a collection up to date in real time from outside a view, which suits a model or service layer. `HistoryObserver` reports when another context or a CloudKit sync changes the



store. And the new `codable` option for `@Attribute` lets a model store a property of any `Codable` type, including one you do not control, such as a type from another framework or package. Each is worth reaching for as your data layer grows.

SwiftData fits structured, queryable data with relationships and history. For a handful of preferences, `UserDefaults` is lighter, as we saw in the *UserDefaults* section, and for content you read and write as files yourself, the *L ocal Files* section covers the approach.

See also

- `SwiftData` documentation: https://x.swiftuicookbook.com/4amxp
- `@Model` documentation: https://x.swiftuicookbook.com/yeziq
- `@Query` documentation: https://x.swiftuicookbook.com/z5dmq
- `ModelContainer` documentation: https://x.swiftuicookbook.com/rx30e
- `ModelContext` documentation: https://x.swiftuicookbook.com/ksp04
- Meet SwiftData (WWDC 202 3): https://x.swiftuicookbook.com/kogew

Local Files

Between a handful of settings and a full relational database sits a common middle ground: a moderate amount of structured data that belongs to the app and changes as the user works. I reach for a plain file in the file system for this more often than people expect. `UserDefaults` is built for small preferences, and a database is more machinery than a short list of records needs, so when the data is naturally one collection, encoding it ourselves and writing it to a file is the most direct option we have.

In this recipe, we build a small journaling app. Each entry has a mood, a title, and a body, and the whole collection lives in a single JSON file in the app's Documents directory. An `@Observable` store reads that file when the app starts and rewrites it after every change, so on the next launch of the app, the journal is exactly where the user left it. Along the way we locate the file with `URL.documentsDirectory`, turn entries into `Data` with `JSONEncoder` and back with `JSONDecoder`, write atomically with `Data.write(to:options:)`, and read the file's size from its URL. A header on the main screen shows that size, so the storage is something we can watch grow rather than an abstraction.

What makes files different from the previous two recipes is that nothing is hidden: we choose the directory, name the file, decide when to read and write, and handle the errors that file access can raise. By the end, you should be comfortable reading and writing `Codable` data as files, picking the right directory for a given kind of data, and deciding when the file system is a b etter fit than `UserDefaults` or SwiftData.

Getting ready

Create a new SwiftUI iOS app named **SimpleJournal** .

How to do it...

We build the app from the bottom up, starting with the data we save and ending with the interface that displays it. First, the model for a single journal entry, then the store that reads and writes the JSON file, and only once that persistence is working do we add the views that list and open the entries. Getting data correctly into the file system and back is the part that matters most, so we keep it in front of us first. These are the steps:

1. Start with the shape of one entry. Create `JournalEntry.swift` in a `Models` folder. An entry carries a mood, a title, and a body, along with a creation date and a stable identifier. It conforms to `Codable` so the system can turn it into JSON, and to `Identifiable` so SwiftUI can track it in a list:

```
import Foundation
struct JournalEntry: Codable, Identifiable {
  var id = UUID()
  var title: String
  var content: String
  var mood: String
  var createdAt: Date
}
```

2. Now, the heart of the recipe. Create `JournalStore.swift` in the `Models` folder, an `@Observable` class that owns the entries and knows how to read and write them. It builds the file URL by appending a file name to the Documents directory. `load()` runs in `init` , returning early when the file does not exist yet and

otherwise decoding the JSON into our entries; `save()` encodes the entries and writes them back, returning whether the write succeeded. Both wrap their work in `do` / `catch`, because file access can fail, recording any problem in an `errorMessage` the views can show:

```
import SwiftUI
@Observable
class JournalStore {
  var entries: [JournalEntry] = []
  var errorMessage: String?
  private var fileURL: URL {
    URL.documentsDirectory.appending(path: "journal.json")
  }
  init() {
    load()
  }
  func load() {
    guard let data = try? Data(contentsOf: fileURL) else {
      return
    }
    do {
      let decoder = JSONDecoder()
      decoder.dateDecodingStrategy = .iso8601
      entries = try decoder.decode([JournalEntry].self, fr
    } catch {
      errorMessage = "Could not load entries: \(error.loca
    }
  }
  @discardableResult
  func save() -> Bool {
    do {
      let encoder = JSONEncoder()
      encoder.outputFormatting = .prettyPrinted
      encoder.dateEncodingStrategy = .iso8601
      let data = try encoder.encode(entries)
      try data.write(to: fileURL, options: [.atomic])
      return true
    } catch {
```

```
            errorMessage = "Could not save entries: \(error.loca
            return false
        }
    }
}
```



We locate the file with `URL.documentsDirectory`, a static property added in iOS 16. Before it, you reached the same folder through `FileManager`, the general file-system class, with `urls(for:in:)`. `FileManager` is not obsolete: it is still the right tool for directory-level work such as creating, listing, moving, and deleting files. For reading and writing a single file, though, the URL-based APIs we use throughout this recipe are the more direct choice.

3. With reading and writing in place, add the methods that change the journal. Each one snapshots `entries`, makes its change, and calls `save()`, restoring the snapshot if the write fails so the list never shows an entry that was never saved. `addEntry` inserts the newest entry at the top, and `deleteEntries` removes the rows the user swipes away:

```
func addEntry(title: String, content: String, mood: String
  let backup = entries
  let entry = JournalEntry(title: title, content: content,
                           mood: mood, createdAt: Date())
  entries.insert(entry, at: 0)
  if !save() {
    entries = backup
```

```
  }
}
func deleteEntries(at offsets: IndexSet) {
  let backup = entries
  entries.remove(atOffsets: offsets)
  if !save() {
    entries = backup
  }
}
```

4. Add one more property to make the storage visible. `fileSize` reads the size from the file URL's resource values and formats the byte count for display, returning a friendly placeholder before the file exists. We will surface this in the interface, so the file is something we can watch rather than just trust:

```
var fileSize: String {
  guard let size = try? fileURL.resourceValues(
    forKeys: [.fileSizeKey]).fileSize else {
    return "No file yet"
  }
  return Int64(size).formatted(.byteCount(style: .file))
}
```

5. Now, the views, starting with a single row. Create `EntryRowView.swift` in a `Views` folder. It pairs the entry's mood symbol with its title and creation date, giving the list a compact, scannable layout:

```
import SwiftUI
struct EntryRowView: View {
  let entry: JournalEntry
  var body: some View {
    HStack {
      Image(systemName: entry.mood)
```

```
          .foregroundStyle(.tint)
          .font(.title3)
        VStack(alignment: .leading, spacing: 2) {
          Text(entry.title)
            .font(.headline)
          Text(entry.createdAt, format: .dateTime.month().da
            .font(.caption)
            .foregroundStyle(.secondary)
        }
      }
    }
  }
  #Preview {
    EntryRowView(entry: JournalEntry(title: "A good day",
      content: "Went for a long walk.", mood: "face.smiling"
      createdAt: Date()))
  }
```

Check the row in the preview canvas, where the sample entry gives it a mood, a title, and today's date.

6. Create `NewEntryView.swift` in the `Views` folder, the sheet for composing an entry. A segmented picker chooses the mood, a text field holds the title, and a `TextEditor` takes the body. The checkmark button hands the new entry to the store and dismisses the sheet, and it stays disabled until there is at least a title:

```
import SwiftUI
struct NewEntryView: View {
  let store: JournalStore
  @Environment(\.dismiss) private var dismiss
  @State private var title = ""
  @State private var content = ""
  @State private var mood = "face.smiling"
  private let moods = ["face.smiling", "face.dashed",
                       "cloud.rain", "star.fill", "moon.zz
```

```
  var body: some View {
    NavigationStack {
      Form {
        Section("Mood") {
          Picker("Mood", selection: $mood) {
            ForEach(moods, id: \.self) { mood in
              Image(systemName: mood).tag(mood)
            }
          }
          .pickerStyle(.segmented)
        }
        Section("Entry") {
          TextField("Title", text: $title)
          TextEditor(text: $content)
            .frame(minHeight: 150)
        }
      }
      .navigationTitle("New Entry")
      .toolbar {
        ToolbarItem(placement: .cancellationAction) {
          Button("Cancel", systemImage: "xmark") { dismiss
        }
        ToolbarItem(placement: .confirmationAction) {
          Button("Save", systemImage: "checkmark") {
            store.addEntry(title: title, content: content,
            dismiss()
          }
          .disabled(title.isEmpty)
        }
      }
    }
  }
}
#Preview {
  NewEntryView(store: JournalStore())
}
```

Its preview builds a real `JournalStore`, so an entry saved from the preview canvas is written to the Documents directory of

the preview's own copy of the app, not the one you run in Device Hub.

7. Create `EntryDetailView.swift` in the `Views` folder to read a single entry in full. It shows the mood and title together, a long-form date, and the body text inside a scroll view, so a longer entry stays readable, while `.scrollBounceBehavior(.basedOnSize)` keeps a short one from bouncing when there is nothing to scroll:

```
import SwiftUI
struct EntryDetailView: View {
  let entry: JournalEntry
  var body: some View {
    ScrollView {
      VStack(alignment: .leading, spacing: 16) {
        HStack {
          Image(systemName: entry.mood)
            .font(.largeTitle)
            .foregroundStyle(.tint)
          Text(entry.title)
            .font(.title)
            .fontWeight(.bold)
        }
        Text(entry.createdAt, format: .dateTime.weekday(.w
          .month().day().year().hour().minute())
          .font(.subheadline)
          .foregroundStyle(.secondary)
        Divider()
        Text(entry.content)
          .font(.body)
      }
      .padding()
    }
    .scrollBounceBehavior(.basedOnSize)
    .navigationTitle("Entry")
    .navigationBarTitleDisplayMode(.inline)
  }
}
#Preview {
```

```
    EntryDetailView(entry: JournalEntry(title: "A good day",
      content: "Went for a long walk and felt great.",
      mood: "face.smiling", createdAt: Date()))
}
```

In the preview canvas the sample entry fits on one screen, and with the bounce modifier in place it stays put when dragged.

8. With the views ready, assemble the main screen, starting with the list that displays the journal. Open `ContentView.swift` and replace the template. The view creates the store in a `@State` property, and its `List` holds two sections: a **Storage** section showing the file size and entry count, and an **Entries** section listing each entry as a `NavigationLink` to its detail view, with swipe-to-delete wired to the store:

```
import SwiftUI
struct ContentView: View {
  @State private var store = JournalStore()
  var body: some View {
    NavigationStack {
      List {
        Section("Storage") {
          LabeledContent("File size", value: store.fileSiz
          LabeledContent("Entries", value: "\(store.entrie
        }
        Section("Entries") {
          ForEach(store.entries) { entry in
            NavigationLink {
              EntryDetailView(entry: entry)
            } label: {
              EntryRowView(entry: entry)
            }
          }
          .onDelete(perform: store.deleteEntries)
        }
      }
```

```
      .navigationTitle("Journal")
    }
  }
}
#Preview {
  ContentView()
}
```

Check the preview canvas: with nothing saved yet, the **Storage** section reads **No file yet** , and the **Entries** section stays empty until the next step adds a way to create entries.

9. Now, make the journal editable, and give the empty journal a welcoming first screen. A new `@State` property below the store tracks whether the compose sheet is showing, a toolbar button raises it, and the sheet presents `NewEntryView` . When there are no entries yet, a `ContentUnavailableView` overlay fills the screen with an invitation to create the first one. The toolbar, the overlay, and the sheet all attach to the `List` after `navigationTitle` :

```
struct ContentView: View {
  @State private var store = JournalStore()
  @State private var showingNewEntry = false
  var body: some View {
    NavigationStack {
      List {
        // ... the Storage and Entries sections
      }
      .navigationTitle("Journal")
      .toolbar {
        ToolbarItem(placement: .topBarTrailing) {
          Button("New Entry", systemImage: "square.and.per
            showingNewEntry = true
          }
        }
      }
```

```
        .overlay {
          if store.entries.isEmpty {
            ContentUnavailableView("No Entries", systemImage
              description: Text("Tap the compose button to s
          }
        }
        .sheet(isPresented: $showingNewEntry) {
          NewEntryView(store: store)
        }
      }
    }
  }
```

10. Finish the screen by surfacing failures. Because every file operation can throw, the store records any problem in `errorMessage`. We present it with the iOS 27 item-driven alert, binding the modifier straight to that optional so the alert appears whenever a message is waiting and clears itself on dismissal:

```
// ... the List, toolbar, and overlay
.sheet(isPresented: $showingNewEntry) {
  NewEntryView(store: store)
}
.alert("Error", item: $store.errorMessage) { _ in
  Button("OK") { }
} message: { message in
  Text(message)
}
```

11. Run the app in Device Hub, since the point is to quit and relaunch. The **Storage** header starts at **No file yet** with zero entries. Tap the compose button, pick a mood, write a title and a few lines, and tap the checkmark button: the entry appears at the top and the file size jumps to a real byte count. Open an

entry to read it in full, and swipe a row away to delete it. Quit the app from the app switcher and launch it again, and every entry is still there, because it was written to a file the moment you saved.

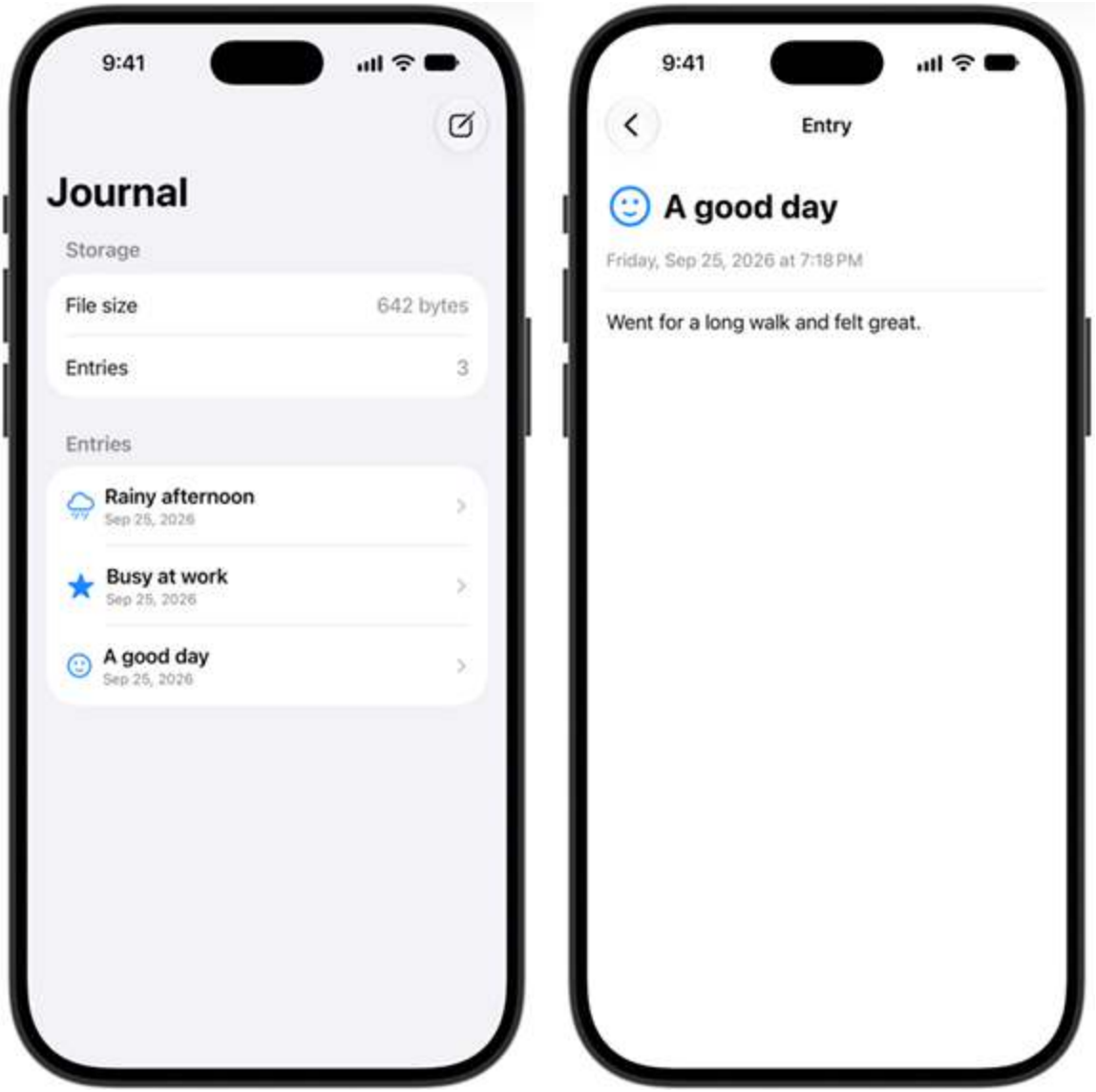


Figure 7.4: The journal list with its storage header and a single entry open for reading

12. Our store already handles a failed save, but it is worth watching that path actually run. Let's force a failure on purpose. In `save()`

, temporarily point the write at a path the app is not allowed to touch:

```
// ... inside save(), in place of the write to fileURL:
// Temporary: an unwritable path makes the write throw.
try data.write(to: URL(filePath: "/journal.json"), options
```

Run the app in Device Hub again and add an entry. The write throws, `save()` returns `false`, and two things happen together: the **Error** alert appears with the message, and the new entry is rolled back out of the list, so nothing that was never written is left behind. Dismiss the alert and restore the original `fileURL` destination afterward. The `Steps/Step12.swift` file reproduces the same result on its own, using a small `JournalStore` subclass whose saves can be switched to fail, so you can run it without editing the real store.



Figure 7.5: The error alert on a failed save, and the unchanged list after dismissing it

How it works...

Every byte the app persists moves through `JournalStore`. It builds one `URL` by appending `journal.json` to `URL.documentsDirectory`, the modern, `FileManager`-free way to reach the app's Documents folder, and that single URL is the address for both reading and writing. Because the class is `@Observable`, the views hold it as `@State` and

simply read `entries`; when `addEntry` or `deleteEntries` changes the array, the list refreshes on its own.

The file APIs give us bytes, not `JournalEntry` values: `Data.write(to:)` and `Data(contentsOf:)` store and return raw bytes, so our `[JournalEntry]` array cannot be written as it stands. Flattening that typed value into a sequence of bytes is serialization, and rebuilding the value from those bytes is deserialization. `save()` uses a `JSONEncoder` to serialize the array into JSON-formatted `Data`, using `.iso8601` so `createdAt` stays portable, then writes with `.atomic` so a failed write does not leave a half-written `journal.json`. `load()` runs the path in reverse: it returns early when the file does not exist, keeping a fresh install as an empty journal, and otherwise uses a `JSONDecoder` to rebuild the entries from the saved bytes. This serialize/deserialize step is the price of owning a file directly, and `Codable` keeps that price low by making `JournalEntry` serializable for us.

We reached for JSON rather than a property list, and the choice is worth a word. A `JSONEncoder` writes text we can open and read, which keeps the stored file easy to inspect and portable to any other tool or platform. When size and speed matter more than readability, a property list saved in its binary format is the alternative: it packs the same `Codable` values into a smaller file that encodes and decodes faster, in exchange for being Apple-specific and not human-readable. That is the format we leaned on in the *UserDefaults* section, because `UserDefaults` is itself backed by a property list; for a standalone file, JSON is the friendlier default.

> To look at `journal.json` yourself, print the directory once from the app, for example with `.task {`



`print(URL.documentsDirectory.path(percentEncoded: false))` `}` on the `ContentView()` inside `SimpleJournalApp`. Copy the path from the Xcode console into Finder's **Go** > **Go to Folder** command, or *Cmd* + *Shift* + *G* , and the file opens in any text editor, because the simulator's Documents directory is an ordinary folder on your Mac. Remove the line when you are done.

Unlike `@AppStorage` or SwiftData, file access makes its failure modes explicit: every read and write is a throwing call. We wrap each one in `do` / `catch` and store any failure in `errorMessage` , which the main view turns into an alert with the iOS 27 item-driven `alert(_:item:)` overload from the *Alerts and Confirmation Dialogs* section of *Chapter 4* : it binds straight to the optional, presenting the alert while `errorMessage` is non- `nil` and resetting it on dismissal, so no separate `Bool` can drift out of sync. In everyday use the Documents directory rarely refuses a write, but treating file access as something that can fail is the habit that keeps file code robust, and it is the clearest difference between owning a file yourself and letting a higher-level store take care of it.

Surfacing the failure is only half the job; the other half is not trusting a change that did not happen. Because each mutation updates `entries` before it knows whether the write will succeed, a failed save would otherwise leave a phantom entry in the list, or hide one that is still on disk. `addEntry` and `deleteEntries` guard against that by taking a `backup` snapshot of the array, applying the change, and assigning the snapshot back when `save()` returns `false` , so the in-memory list and the file never disagree. Updating first and rolling back only on

failure keeps the interface immediate on the common success path, while still treating the file as the final authority on what the journal contains. `save()` returns that `Bool` so its callers can make this decision, and we mark it `@discardableResult` so the same method can also be called purely to persist, without the compiler flagging an ignored result.



The Documents directory is one of several the sandbox gives every app, and the right one depends on the data. Documents is for user-created content we want backed up and kept; the Caches directory is for data we can recreate, such as downloaded thumbnails, and the system may purge it when storage runs low; Application Support holds private support files the user never sees; and the temporary directory is for short-lived working files that may vanish at any time. `URL` exposes each one as a static property, so swapping `URL.documentsDirectory` for `URL.cachesDirectory` is all it takes to move a file.

A file in the file system sits between the two stores from earlier in the chapter. When the data is a handful of preferences, the *UserDefaults* section is simpler; when it is large or relational and needs querying, the *SwiftData* section does the heavy lifting. Encoding a `Codable` collection to a single JSON file is the right reach in between, when the data is too structured for `UserDefa ults` but does not need a database built for complex queries.

See also

- `URL` documentation: https://x.swiftuicookbook.com/jmor9
- `documentsDirectory` documentation: https://x.swiftuicookbook.com/dtspj
- `Data` documentation: https://x.swiftuicookbook.com/0rukw
- `JSONEncoder` documentation: https://x.swiftuicookbook.com/w6ktt
- `JSONDecoder` documentation: https://x.swiftuicookbook.com/p85wu
- `URLResourceValues` documentation: https://x.swiftuicookbook.com/isqf5

8

Networking and Remote Data

I have yet to ship an app that didn't need remote data. Sooner or later a screen needs something from a server: a search result, a product feed, a photo. When I started in iOS, that meant completion handlers nested three levels deep, JSON picked apart by hand with `JSONSerialization`, and an image cache someone on the team wrote on a Friday. Every piece of that stack has since been replaced by something better, and this chapter tours the modern replacements.

It is also the chapter where the data-flow tools we have been using in the second part of the book earn their keep. The observable models, the bindings, and the task lifecycle we practiced on local data all reappear here, now fed by live HTTP responses instead of sample arrays.

We start with the bridge between a JSON-encoded `String` and Swift: a product catalog built on `Codable`, where a `CodingKeys` enum maps the API's snake_case keys onto camelCase properties, `JSONDecoder` parses the feed, and `JSONEncoder` serializes the models back into formatted JSON text. Then, we connect to the iTunes Search API using two different approaches. The async/await approach fetches data with `URLSession`, funnels failures into a typed `NetworkError`, and debounces search-as-you-type with a cancellable `.task(id:)` on a

`@MainActor` view model. The Combine approach keeps the same view but expresses the search as a pipeline, where `dataTaskPublisher(for:)` feeds `debounce` and `switchToLatest`, published to the UI through an `ObservableObject`. We finish with image downloads. A photo browser renders remote photos with `AsyncImage` and gives every `AsyncImagePhase` its own presentation. On top of it we adopt the three iOS 27 caching additions: automatic HTTP caching, the `AsyncImage(request:)` initializers with per-request cache policies, and the `asyncImageURLSession()` modifier with its own sized cache.

By the end of the chapter, you should be comfortable taking a raw JSON `String` from an HTTP API response payload to a finished SwiftUI view, have a clear opinion on when async/await or Combine fits better, and know what image caching iOS 27 performs automatically and how to take control of it when the defaults are not enough.

We are going to cover the following recipes in this chapter:

- Codable Models
- Async/await with Remote APIs
- Combine with Remote Data
- AsyncImage and Caching

Technical requirements

The code in this chapter is based on Xcode 27.0 and iOS 27.0. You can download and install the latest version of Xcode from the App Store. All the code examples for this chapter can be found on GitHub at https://x.swiftuicookbook.com/ch08.

Codable Models

Almost every app I build eventually has to turn a block of JSON from a server into Swift values it can actually use. Before `Codable`, that meant reaching for `JSONSerialization`, casting `Any` out of nested dictionaries one key at a time, and writing a small mountain of glue code that broke the moment the payload changed. `Codable` replaced all of it with a single idea: describe the shape of the data as ordinary Swift types, and let the compiler generate the reading and writing for us.

In this recipe, we build an app that loads a product feed and shows it in a filterable list. The feed is a JSON file shaped like an HTTP response, and decoding it exercises the parts of `Codable` that come up in real projects: a `CodingKeys` enum to bridge the API's snake_case keys to camelCase properties, a `dateDecodingStrategy` to parse calendar-string dates, an optional property for a value that is sometimes `null`, and an enum that is `Codable` all on its own. We also add a button that encodes the models back to JSON with `JSONEncoder`, so we can see both decoding and encoding in one app. The models we design here are exactly what the networking recipes later in this chapter decode from live servers, so this is a good place to get comfortable with `Codable` before any network code is involved.

Getting ready

Create a new SwiftUI iOS app named **ProductCatalog**.

How to do it...

We will start with the data. First, we add the JSON our app will decode, then the Swift types that mirror it, and a store that turns one into the other. With the data layer in place we build the catalog screen, and we finish by encoding our models back into JSON. These are the steps:

1. Our app decodes a JSON feed shaped like a typical HTTP response: an array of products wrapped in an object with paging fields. Download the `products.json` file from the `ch08/assets/` folder in the book's GitHub repository, then drag it into the Xcode Project Navigator so it is bundled with the app. The file holds five products; the first two are shown here, since they cover the shapes that matter:

```
{
  "products": [
    {
      "id": 1,
      "name": "Wireless Headphones",
      "description": "Noise-cancelling headphones with 30-
      "price": 249.99,
      "category": "electronics",
      "image_url": "https://picsum.photos/seed/headphones/
      "rating": 4.7,
      "review_count": 1284,
      "release_date": "2024-03-15",
      "is_available": true,
      "tags": ["wireless", "audio", "noise-cancelling"]
    },
    {
      "id": 2,
      "name": "SwiftUI Cookbook 3rd Edition",
      "description": "SwiftUI for iOS development.",
      "price": 44.99,
      "category": "books",
      "image_url": null,
```

```
      "rating": 4.7,
      "review_count": 46,
      "release_date": "2023-12-26",
      "is_available": true,
      "tags": ["programming", "Swift", "SwiftUI", "iOS"]
    }
    // 3 more objects
  ],
  "total_count": 5,
  "page": 1
}
```

Notice the three things that will not map straight onto Swift properties: the keys are snake_case, one `image_url` is `null`, and the dates are plain `yyyy-MM-dd` strings.

2. Model the product category first, since the other types depend on it. Create `ProductCategory.swift` in a `Models` folder. A `String` raw value makes the enum `Codable` automatically, and a `symbolName` property maps each case to an SF Symbol for the row label:

```
import Foundation
enum ProductCategory: String, Codable, CaseIterable {
  case electronics
  case clothing
  case books
  case food
  var symbolName: String {
    switch self {
    case .electronics: "headphones"
    case .clothing: "tshirt"
    case .books: "book"
    case .food: "cup.and.saucer"
    }
  }
}
```

3. Create `Product.swift` in the `Models` folder. Each property matches a field in the JSON, and the `CodingKeys` enum maps the snake_case keys onto our camelCase names. `imageURL` is optional because its value can be `null`:

```
import Foundation
struct Product: Codable, Identifiable {
  let id: Int
  let name: String
  let description: String
  let price: Double
  let category: ProductCategory
  let imageURL: String?
  let rating: Double
  let reviewCount: Int
  let releaseDate: Date
  let isAvailable: Bool
  let tags: [String]
  enum CodingKeys: String, CodingKey {
    case id, name, description, price, category
    case imageURL = "image_url"
    case rating
    case reviewCount = "review_count"
    case releaseDate = "release_date"
    case isAvailable = "is_available"
    case tags
  }
}
extension Product {
  static let preview = Product(
    id: 1,
    name: "Wireless Headphones",
    description: "Noise-cancelling headphones with 30-hour
    price: 249.99,
    category: .electronics,
    imageURL: "https://picsum.photos/seed/headphones/400/4
    rating: 4.7,
    reviewCount: 1284,
    releaseDate: .now,
    isAvailable: true,
```

```
    tags: ["wireless", "audio", "noise-cancelling"]
  )
}
```

`Product` is `Identifiable` so it can drive a `List` directly, using its `id` field for identity. The `preview` sample at the bottom gives us a ready-made `Product` for the previews we build next.

4. The top level of the feed is an object, not a bare array, so we mirror it with a wrapper type. Create `APIResponse.swift` in the `Models` folder, mapping `total_count` using a `CodingKeys` enum:

```
import Foundation
struct APIResponse: Codable {
  let products: [Product]
  let totalCount: Int
  let page: Int
  enum CodingKeys: String, CodingKey {
    case products
    case totalCount = "total_count"
    case page
  }
}
```

5. Now, the piece that does the decoding. Create `ProductStore.swift` in the `Models` folder, an `@Observable` class that reads the bundled file and turns it into `[Product]`. We give `JSONDecoder` a date strategy that matches our `yyyy-MM-dd` strings, and we catch the most common decoding failures by name so a mismatch surfaces as a readable message rather than an empty screen:

```
import Foundation
@Observable
class ProductStore {
```

```
  var products: [Product] = []
  var selectedCategory: ProductCategory?
  var errorMessage: String?
  var filteredProducts: [Product] {
    guard let selectedCategory else { return products }
    return products.filter { $0.category == selectedCatego
  }
  init() {
    loadProducts()
  }
  func loadProducts() {
    guard let url = Bundle.main.url(forResource: "products
                                    withExtension: "json")
      errorMessage = "Could not find products.json in the
      return
    }
    let decoder = JSONDecoder()
    let formatter = DateFormatter()
    formatter.dateFormat = "yyyy-MM-dd"
    formatter.locale = Locale(identifier: "en_US_POSIX")
    decoder.dateDecodingStrategy = .formatted(formatter)
    do {
      let data = try Data(contentsOf: url)
      let response = try decoder.decode(APIResponse.self,
      products = response.products
      errorMessage = nil
    } catch let DecodingError.keyNotFound(key, context) {
      let description = context.debugDescription
      errorMessage = "Missing key '\(key.stringValue)': \(
    } catch let DecodingError.typeMismatch(type, context)
      let description = context.debugDescription
      errorMessage = "Type mismatch for \(type): \(descrip
    } catch {
      errorMessage = "Failed to load products: \(error.loc
    }
  }
}
```

Because `init()` calls `loadProducts()`, the store is populated the moment it is created, so the view always has data to show on

its first render.

6. With the data layer done, we switch to the user interface, starting with the smallest visible piece. Create `ProductRowView.swift` in a `Views` folder. For now the row shows only the name, price, and description, laid out with a `header` we can grow later:

```
import SwiftUI
struct ProductRowView: View {
  let product: Product
  var body: some View {
    VStack(alignment: .leading, spacing: 8) {
      header
      Text(product.description)
        .font(.caption)
        .foregroundStyle(.secondary)
        .lineLimit(2)
    }
    .padding(.vertical, 4)
  }
  private var header: some View {
    HStack {
      Text(product.name)
        .font(.headline)
      Spacer()
      Text(product.price, format: .currency(code: "USD"))
        .font(.subheadline)
        .fontWeight(.semibold)
        .foregroundStyle(product.isAvailable ? .primary :
    }
  }
}
#Preview {
  List {
    ProductRowView(product: .preview)
  }
  .listStyle(.plain)
}
```

The preview uses the `Product.preview` sample from step 3, so the row renders in the preview canvas on its own, without loading the JSON file.

7. Fill in the rest of the row. Add three computed properties, a `detailRow` with the category, rating, and release date, a `tagList`, and an `availability` label, then place them in `body`. The two `@ViewBuilder` properties return nothing when there is no content to show:

```
struct ProductRowView: View {
  let product: Product
  var body: some View {
    VStack(alignment: .leading, spacing: 8) {
      header
      // ... Text with product description
      detailRow
      tagList
      availability
    }
    .padding(.vertical, 4)
  }
  private var header: some View {
    // ... existing header code
  }
  private var detailRow: some View {
    HStack {
      Label(product.category.rawValue.capitalized,
            systemImage: product.category.symbolName)
        .font(.caption2)
        .padding(.horizontal, 8)
        .padding(.vertical, 4)
        .background(.blue.opacity(0.1))
        .clipShape(Capsule())
      Spacer()
      HStack(spacing: 2) {
        Image(systemName: "star.fill")
```

```
            .foregroundStyle(.yellow)
          Text(product.rating, format: .number.precision(.fr
          Text("(\(product.reviewCount))")
            .foregroundStyle(.secondary)
        }
        .font(.caption)
        Text(product.releaseDate, format: .dateTime.month(.a
          .font(.caption2)
          .foregroundStyle(.secondary)
      }
    }
    @ViewBuilder
    private var tagList: some View {
      if !product.tags.isEmpty {
        HStack {
          ForEach(product.tags, id: \.self) { tag in
            Text(tag)
              .font(.caption2)
              .padding(.horizontal, 6)
              .padding(.vertical, 2)
              .background(.gray.opacity(0.15))
              .clipShape(Capsule())
          }
        }
      }
    }
    @ViewBuilder
    private var availability: some View {
      if !product.isAvailable {
        Text("Out of Stock")
          .font(.caption)
          .fontWeight(.medium)
          .foregroundStyle(.red)
      }
    }
}
```

The `releaseDate` and `price` both use format styles, so the `Date` and `Double` we decoded render as a localized month and currency without any manual string work.

8. Replace the `ContentView` struct with a first version that shows the decoded products in a `List` and displays `errorMessage` if the decoding failed:

```
import SwiftUI
struct ContentView: View {
  @State private var store = ProductStore()
  var body: some View {
    NavigationStack {
      VStack(spacing: 0) {
        if let errorMessage = store.errorMessage {
          Text(errorMessage)
            .font(.caption)
            .foregroundStyle(.red)
            .padding()
        }
        List(store.products) { product in
          ProductRowView(product: product)
        }
        .listStyle(.plain)
      }
      .navigationTitle("Product Catalog")
    }
  }
}
#Preview {
  ContentView()
}
```

   Preview `ContentView` now. The list shows all five products, which confirms the whole file decoded correctly.

9. Create `CategoryChipView.swift` in the `Views` folder, a small selectable capsule we will use to filter by category:

```
import SwiftUI
struct CategoryChipView: View {
  let title: String
```

```
  let isSelected: Bool
  let action: () -> Void
  var body: some View {
    Button(action: action) {
      Text(title)
        .font(.caption)
        .fontWeight(isSelected ? .semibold : .regular)
        .padding(.horizontal, 12)
        .padding(.vertical, 6)
        .foregroundStyle(isSelected ? .white : .primary)
        .background(isSelected ? Color.blue : Color.gray.o
        .clipShape(Capsule())
    }
    .buttonStyle(.plain)
  }
}
#Preview {
  HStack {
    CategoryChipView(title: "All", isSelected: true) {}
    CategoryChipView(title: "Books", isSelected: false) {}
  }
  .padding()
}
```

10. Back in `ContentView`, add a horizontal row of chips above the list and switch the list to `filteredProducts`. Because `ProductCategory` is `CaseIterable`, a single `ForEach` over `allCases` builds one chip per category:

```
struct ContentView: View {
  @State private var store = ProductStore()
  var body: some View {
    NavigationStack {
      VStack(spacing: 0) {
        categoryFilter
        // ... error message
        List(store.filteredProducts) { product in
          ProductRowView(product: product)
        }
```

```
            .listStyle(.plain)
        }
        .navigationTitle("Product Catalog")
      }
    }
    private var categoryFilter: some View {
      ScrollView(.horizontal, showsIndicators: false) {
        HStack(spacing: 8) {
          CategoryChipView(title: "All",
                           isSelected: store.selectedCategor
            store.selectedCategory = nil
          }
          ForEach(ProductCategory.allCases, id: \.self) { ca
            CategoryChipView(
              title: category.rawValue.capitalized,
              isSelected: store.selectedCategory == category
            ) {
              store.selectedCategory = category
            }
          }
        }
        .padding(.horizontal)
      }
      .padding(.vertical, 8)
    }
}
```

Tapping **Electronics** in the live preview now narrows the list to the headphones and the MacBook Air, since both decoded into the same `ProductCategory` case.

11. Now, we add encoding. Give `ProductStore` an `encodedJSON()` method, after `loadProducts()`, that serializes `filteredProducts`, the array the list is showing, into JSON text with `JSONEncoder`:

```
@Observable
class ProductStore {
  // ... the stored properties, filteredProducts, init(),
```

```
  func encodedJSON() -> String {
    let encoder = JSONEncoder()
    encoder.outputFormatting = [
      .prettyPrinted, .sortedKeys, .withoutEscapingSlashes
    ]
    let formatter = DateFormatter()
    formatter.dateFormat = "yyyy-MM-dd"
    formatter.locale = Locale(identifier: "en_US_POSIX")
    encoder.dateEncodingStrategy = .formatted(formatter)
    do {
      let data = try encoder.encode(filteredProducts)
      return String(data: data, encoding: .utf8) ?? "Encod
    } catch {
      return "Encoding failed: \(error.localizedDescriptio
    }
  }
}
```

The same `CodingKeys` that mapped `image_url` when decoding produce `image_url` when encoding, so the encoded feed matches the shape we started from.

12. Create `EncodedJSONView.swift` in the `Views` folder, a sheet that shows the encoded text in a scrollable, selectable monospaced block and dismisses itself through a standard close button:

```
import SwiftUI
struct EncodedJSONView: View {
  let json: String
  @Environment(\.dismiss) private var dismiss
  var body: some View {
    NavigationStack {
      ScrollView {
        Text(json)
          .font(.system(.caption, design: .monospaced))
          .frame(maxWidth: .infinity, alignment: .leading)
          .textSelection(.enabled)
          .padding()
```

```
      }
      .navigationTitle("Encoded JSON")
      .navigationBarTitleDisplayMode(.inline)
      .toolbar {
        ToolbarItem(placement: .cancellationAction) {
          Button(role: .close) { dismiss() }
        }
      }
    }
  }
}
#Preview {
  EncodedJSONView(json: """
  [
    {
      "id" : 1,
      "name" : "Wireless Headphones",
      "price" : 249.99
    }
  ]
  """)
}
```

13. Finish `ContentView` by adding a `showingEncodedJSON` state property below `store` and wiring a curly braces toolbar button to present the sheet:

```
struct ContentView: View {
  @State private var store = ProductStore()
  @State private var showingEncodedJSON = false
  var body: some View {
    NavigationStack {
      VStack(spacing: 0) {
        // ... category filter, error message, and product
      }
      .navigationTitle("Product Catalog")
      .toolbar {
        ToolbarItem(placement: .topBarTrailing) {
          Button("View JSON", systemImage: "curlybraces")
```

```
                showingEncodedJSON = true
              }
            }
          }
          .sheet(isPresented: $showingEncodedJSON) {
            EncodedJSONView(json: store.encodedJSON())
          }
        }
      }
      // ... the categoryFilter property
}
```

With the preview canvas in live mode, tap the curly braces button. The sheet shows the products re-encoded to JSON, the same data we started with, now produced by our own model. Close it, select **Electronics** , and open the sheet again: only the headphones and the MacBook Air are encoded, because the sheet follows the filter.

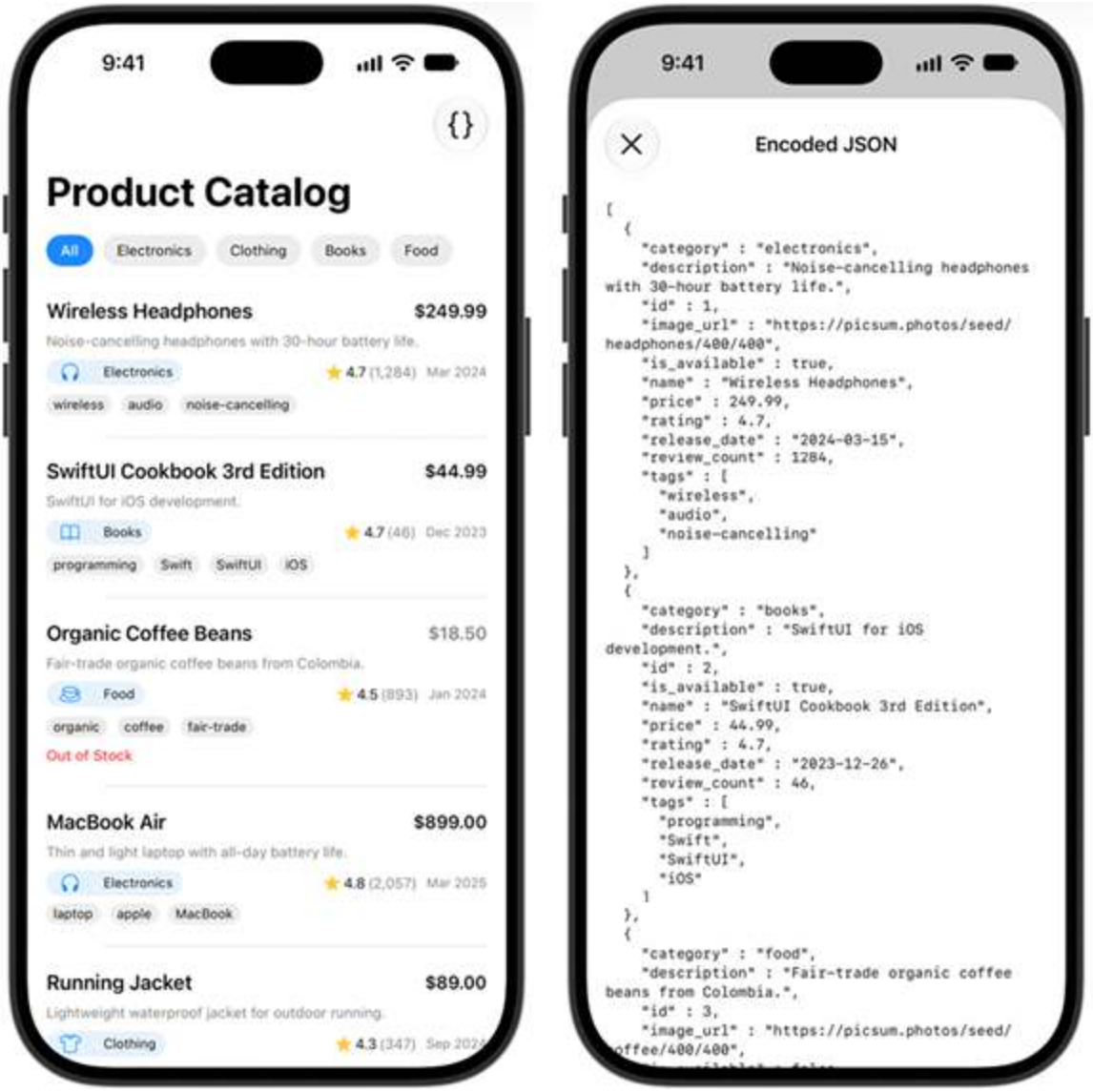


Figure 8.1: The decoded product catalog and the re-encoded J SON output

How it works...

Our five products started life as text in `products.json` and ended up as an array of `Product` values we could filter and display. The bridge is `Codable`, the type alias that combines the `Decodable` and `Encodable` protocols. The compiler synthesizes the conformance to these protocols because every stored property of `Product` is itself `Codable`:

the `Int`, `String`, `Double`, and `Bool` fields, the array of tags, the optional string, the `Date`, and our own `ProductCategory`. `JSONDecoder` walks the data and builds each value, and because `APIResponse` mirrors the object that wraps the array, one `decode(APIResponse.self, from:)` call turns the entire payload into typed Swift. Our store is an `@Observable` class, the observation model we cover in the *Observation Framework* section of *Chapter 5*.

The JSON uses snake_case keys like `image_url` and `review_count`, while our properties are camelCase, and the `CodingKeys` enum is where we spell out that mapping: each case name matches a property, and its string raw value matches the JSON key. Cases whose names already match, such as `id` and `name`, still have to be listed, because declaring the enum opts every property in, and a property left out is excluded from decoding entirely. When the only difference is snake_case versus camelCase, there is a shorter route: set `decoder.keyDecodingStrategy = .convertFromSnakeCase` and delete the enum. We kept explicit keys for one practical reason: `image_url` would convert to `imageUrl`, not the `imageURL` we want, so a property with an uppercase acronym needs a hand-written key regardless. The rule of thumb is to reach for the strategy when names differ only in case, and fall back to `CodingKeys` the moment a single key breaks the pattern.

Dates are the other place raw JSON and Swift disagree. Our `release_date` values are plain calendar strings like `2024-03-15`, which no decoder parses on its own, so we hand `JSONDecoder` a `dateDecodingStrategy` built from a `DateFormatter` whose format is `yyyy-MM-dd`. Setting the formatter's locale to `en_US_POSIX` is not optional: it pins parsing to a fixed calendar so a user's regional settings cannot

change how the string is read. When an API speaks ISO 8601 instead, `.iso8601` handles it directly, and `.secondsSince1970` covers Unix timestamps. The strategy exists so the model stays clean while the decoder absorbs the format differences.

Decoding can fail, and the way it fails is often the fastest clue to a mismatched model. We catch three `DecodingError` cases: `keyNotFound` fires when the JSON is missing a key our model requires, `typeMismatch` when a value has the wrong type, such as a string where we expected a number, and a final `catch` covers everything else. Each error carries a coding path and a debug description, so surfacing it in `errorMessage` turns a silent empty screen into a message that points at the offending field. There is one gentler failure worth noting: `imageURL` is an optional `String?`, so the `null` in the **SwiftUI Cookbook** entry decodes to `nil` rather than throwing. Optionals are how `Codable` expresses a value that may be absent.

Encoding runs the same mapping in reverse. The curly braces button calls `encodedJSON()`, where a `JSONEncoder` turns the filtered `[Product]` array back into text, and the same `CodingKeys` that read `image_url` now write it. Setting `outputFormatting` to `.prettyPrinted`, `.sortedKeys`, and `.withoutEscapingSlashes` gives stable, readable output with unescaped URLs, which is what makes the sheet worth looking at: seeing the encoded result next to the original file is the clearest way to confirm a model is faithful in both directions. For encoding a `Date`, `dateEncodingStrategy` mirrors the decoding side, so the strings we write back out match the `yyyy-MM-dd` format we read in. The presentation around it uses two APIs we lean on lightly here: toolbars are covered in depth in the *Toolbars and the Liquid Glass*

Design section of *Chapter 4*, and sheet presentation in the *Modal Sheets and Full-Screen Covers* section of *Chapter 11*.

See also

- `Codable` documentation: https://x.swiftuicookbook.com/sdzlw
- `JSONDecoder` documentation: https://x.swiftuicookbook.com/p85wu
- `JSONEncoder` documentation: https://x.swiftuicookbook.com/w6ktt
- `CodingKey` documentation: https://x.swiftuicookbook.com/mkouz
- *Encoding and decoding custom types* article: https://x.swiftuicookbo ok.com/8u6i4

Async/await with Remote APIs

For years, calling a web API from an iOS app meant completion handlers: start the request, handle the result in a nested closure, remember to hop back to the main queue before touching the UI, and make sure every code path eventually called the completion exactly once. I have lost real debugging hours to handlers that ran twice or never ran at all. Swift's async/await removes that whole class of problems: the code suspends where we write `await`, resumes with a value or throws an error, and reads top to bottom like any other function.

In this recipe, we build a live music search: the user types in a search bar, and matching songs appear as the typing pauses, complete with artwork, artist, and album. The backend is Apple's public iTunes Search API, so everything works with nothing more than an internet connection: no account, no API key. Along the way we assemble the pieces every networking layer needs: `URLComponents` to build the request URL safely, the async `data(from:)` method of `URLSession` to fetch the response, a `NetworkError` enum that turns failures into readable messages, and a `@MainActor` view model whose `@Observable` state drives the list. The search field comes from `.searchable(text:)`, and a cancellable `.task(id:)` paired with `Task.sleep(for:)` debounces the requests. In the next recipe we rebuild this same screen with Combine, so we can compare the two appro aches directly.

Getting ready

Create a new SwiftUI iOS app nam ed **MusicSearch** .

How to do it...

As in the previous recipe, we will start with the data: the structs that mirror the response, an error type, the service that performs the request, and the view model that connects it all to SwiftUI. With the data layer in place, we build the row view and assemble the search screen on top. These are the steps:

1. Before modeling anything, let's look at what the server sends back. The iTunes Search API answers a query like `itunes.apple.com/search?term=hey+jude&media=music` with a wrapper object much like the product feed from the previous recipe: a

count and an array of results. Each result carries around thirty fields, and the trimmed sample here shows the five our app needs:

```
{
  "resultCount": 25,
  "results": [
    {
      "trackId": 1441164589,
      "trackName": "Here Comes the Sun",
      "artistName": "The Beatles",
      "collectionName": "Abbey Road (Remastered)",
      "artworkUrl100": "https://is1-ssl.mzstatic.com/.../1
      // about 25 more fields per result
    }
    // 24 more results
  ]
}
```

2. Create `MusicTrack.swift` in a `Models` folder, with one struct for the wrapper and one for the track. The `CodingKeys` enum renames the awkward `artworkUrl100` key to `artworkURL` and decodes it directly as an optional `URL`, and a `preview` sample gives us a ready-made track for the previews we build next:

```
import Foundation
struct SearchResponse: Decodable {
  let resultCount: Int
  let results: [MusicTrack]
}
struct MusicTrack: Decodable, Identifiable {
  let trackId: Int
  let trackName: String
  let artistName: String
  let collectionName: String?
  let artworkURL: URL?
```

```
  var id: Int { trackId }
  enum CodingKeys: String, CodingKey {
    case trackId, trackName, artistName, collectionName
    case artworkURL = "artworkUrl100"
  }
}
extension MusicTrack {
  static let preview = MusicTrack(
    trackId: 1441164589,
    trackName: "Here Comes the Sun",
    artistName: "The Beatles",
    collectionName: "Abbey Road (Remastered)",
    artworkURL: URL(string: "https://is1-ssl.mzstatic.com/
      "Music112/v4/df/db/61/dfdb615d-47f8-06e9-9533-b96dac
      "18UMGIM31076.rgb.jpg/100x100bb.jpg")
  )
}
```

Both structs adopt `Decodable` rather than `Codable` because this app only ever reads from the API.

3. Networking fails in more ways than decoding alone, so we give each failure a name. Create `NetworkError.swift` in the `Models` folder, an enum whose cases cover a malformed URL, a non-HTTP response, a bad status code, and a decoding problem. Conforming to `LocalizedError` lets us hand any of them to the UI as a plain message:

```
import Foundation
enum NetworkError: LocalizedError {
  case invalidURL
  case invalidResponse
  case httpError(statusCode: Int)
  case decodingFailed(Error)
  var errorDescription: String? {
    switch self {
    case .invalidURL:
```

```
        return "The request URL is invalid."
      case .invalidResponse:
        return "The server response was not valid HTTP."
      case .httpError(let statusCode):
        return "The server returned status code \(statusCode
      case .decodingFailed(let error):
        return "Decoding failed: \(error.localizedDescriptio
      }
    }
}
```

4. Now, the network call itself. Create `MusicAPIService.swift` in the `Models` folder. The `searchTracks(matching:)` method builds the query URL piece by piece with `URLComponents`, fetches it with `URLSession`'s async `data(from:)` method, validates the HTTP status, and decodes the payload into our models:

```
import Foundation
struct MusicAPIService {
  func searchTracks(matching term: String) async throws ->
    var components = URLComponents()
    components.scheme = "https"
    components.host = "itunes.apple.com"
    components.path = "/search"
    components.queryItems = [
      URLQueryItem(name: "term", value: term),
      URLQueryItem(name: "media", value: "music"),
      URLQueryItem(name: "limit", value: "25")
    ]
    guard let url = components.url else {
      throw NetworkError.invalidURL
    }
    let (data, response) = try await URLSession.shared.dat
    guard let httpResponse = response as? HTTPURLResponse
      throw NetworkError.invalidResponse
    }
    guard (200...299).contains(httpResponse.statusCode) el
      throw NetworkError.httpError(statusCode: httpRespons
```

```
    }
    do {
      let searchResponse = try JSONDecoder().decode(Search
                                                    from:
      return searchResponse.results
    } catch {
      throw NetworkError.decodingFailed(error)
    }
  }
}
```

5. To connect the service to SwiftUI we need observable state. Create `MusicSearchViewModel.swift` in the `Models` folder. We mark the class `@MainActor` so every property change happens on the main thread, and `@Observable` so the views reading `tracks` re-render when it changes:

```
import Foundation
@MainActor
@Observable
class MusicSearchViewModel {
  var tracks: [MusicTrack] = []
  var isSearching = false
  var errorMessage: String?
  private let service = MusicAPIService()
  func search(for term: String) async {
    let query = term.trimmingCharacters(in: .whitespaces)
    errorMessage = nil
    guard !query.isEmpty else {
      tracks = []
      return
    }
    isSearching = true
    defer { isSearching = false }
    do {
      tracks = try await service.searchTracks(matching: qu
    } catch {
      if !Task.isCancelled {
```

```
            errorMessage = error.localizedDescription
        }
      }
    }
  }
```

The `defer` guarantees `isSearching` resets on every exit path, and the `Task.isCancelled` check keeps the error from a canceled request off the screen.

6. With the data layer complete, we switch to the user interface, starting with the view that displays a single track in the results list. Create `TrackRowView.swift` in a `Views` folder. A basic `AsyncImage` loads the artwork with a placeholder for the loading state, and a hairline border keeps plain-white album covers from vanishing into the list background. A single `lineLimit(1)` on the `VStack` caps all three `Text` views at one line, so every row keeps a uniform height:

```
import SwiftUI
struct TrackRowView: View {
  let track: MusicTrack
  var body: some View {
    HStack(spacing: 12) {
      artwork
      VStack(alignment: .leading, spacing: 4) {
        Text(track.trackName)
          .font(.headline)
        Text(track.artistName)
          .font(.subheadline)
          .foregroundStyle(.secondary)
        if let album = track.collectionName {
          Text(album)
            .font(.caption)
            .foregroundStyle(.tertiary)
        }
```

```
        }
        .lineLimit(1)
      }
      .padding(.vertical, 4)
    }
    private var artwork: some View {
      AsyncImage(url: track.artworkURL) { image in
        image
          .resizable()
          .scaledToFill()
      } placeholder: {
        RoundedRectangle(cornerRadius: 8)
          .fill(.gray.opacity(0.15))
          .overlay {
            Image(systemName: "music.note")
              .foregroundStyle(.secondary)
          }
      }
      .frame(width: 56, height: 56)
      .clipShape(RoundedRectangle(cornerRadius: 8))
      .overlay {
        RoundedRectangle(cornerRadius: 8)
          .strokeBorder(.quaternary)
      }
    }
}
#Preview {
    TrackRowView(track: .preview)
      .padding()
}
```

The preview renders a single row from the `MusicTrack.preview` sample, so we can check the layout before we create the search view. `AsyncImage` fetches each artwork once and does not retry a failed load, so an occasional row can keep its gray placeholder. We will make remote image loading more reliable with caching in the *AsyncImage and Caching* section.

7. Replace the `ContentView` struct with a first working version of the search screen: a `List` of rows, a search field from `.searchable(text:)`, and a `.task(id: query)` that runs the search when the view appears and again every time the query changes:

```
import SwiftUI
struct ContentView: View {
  @State private var viewModel = MusicSearchViewModel()
  @State private var query = "The Beatles"
  var body: some View {
    NavigationStack {
      List(viewModel.tracks) { track in
        TrackRowView(track: track)
      }
      .listStyle(.plain)
      .navigationTitle("Music Search")
      .searchable(text: $query, prompt: "Songs and artists
      .task(id: query) {
        await viewModel.search(for: query)
      }
    }
  }
}
#Preview {
  ContentView()
}
```

   Let's put the preview canvas in live mode now. After a moment, the list fills with songs by The Beatles. Try the search field: results already update as we type, as every single keystroke fires its own request.

8. Firing a request per keystroke is wasteful. Someone typing `imagine` triggers seven searches, and six of the responses are stale before they arrive. Update the `.task(id:)` modifier to wait briefly

before searching. Because SwiftUI cancels the running task whenever `id` changes, the sleep doubles as a debounce. A canceled sleep throws, and we simply return:

```
.task(id: query) {
  do {
    try await Task.sleep(for: .milliseconds(300))
  } catch {
    return
  }
  await viewModel.search(for: query)
}
```

9. The screen still shows a blank list while searching, or when something goes wrong. Insert an `.overlay` between `.listStyle(.plain)` and `.navigationTitle` that gives each empty or busy state its own presentation: a spinner while a search is in flight, the error message if the request failed, a prompt when the search field is empty, and the standard no-results view otherwise:

```
.overlay {
  if viewModel.isSearching {
    ProgressView("Searching...")
  } else if let errorMessage = viewModel.errorMessage {
    ContentUnavailableView {
      Label("Search Failed", systemImage: "exclamationmark
    } description: {
      Text(errorMessage)
    }
  } else if viewModel.tracks.isEmpty && query.isEmpty {
    ContentUnavailableView(
      "Search for Music",
      systemImage: "music.note",
      description: Text("Type a song or artist name.")
    )
```

```
    } else if viewModel.tracks.isEmpty {
      ContentUnavailableView.search(text: query)
    }
  }
```

10. We have finally finished our app. Run it and play with the search: results replace each other as the typing pauses, clearing the field brings up the prompt, and a nonsense word like `zzqq` lands on the no-results state:

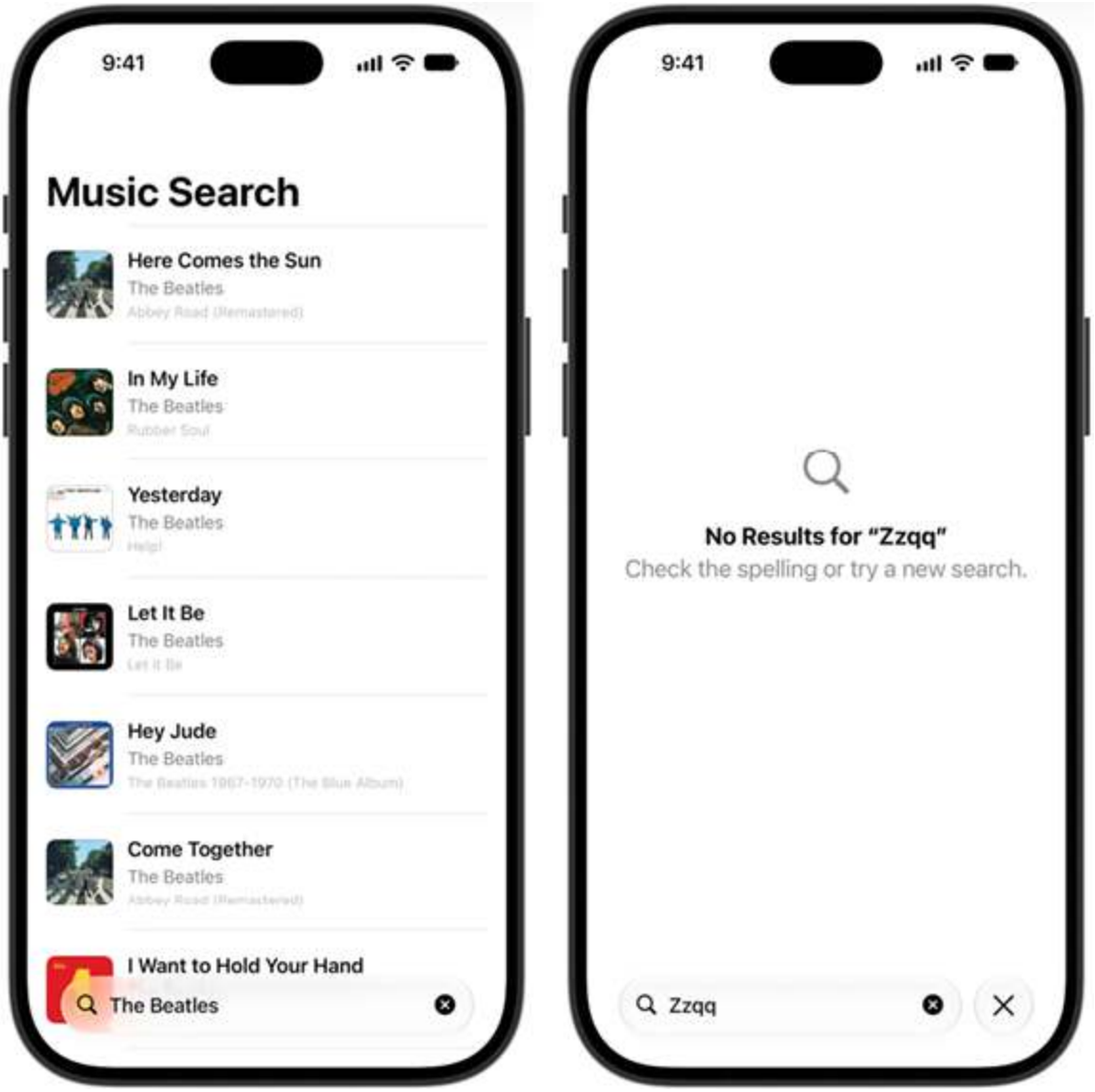


Figure 8.2: Search results for T he Beatles and the no-results state

How it works…

Every search in our app follows the same path: the pause in typing triggers `search(for:)`, the view model awaits the service, the service awaits `URLSession`, and a decoded `[MusicTrack]` array flows back up into the list. The `await` keyword marks where each function suspends. No thread is blocked while the network round trip is in flight, and execution resumes with a value once the response arrives. Errors travel the same road in reverse: `data(from:)` and `decode(_:from:)` both throw, `searchTracks(matching:)` forwards their failures as typed `NetworkError` cases, and the single `do-catch` in the view model turns whatever arrives into a message for the overlay. Compared with the completion-handler version of this screen, there are no nested closures, no `[weak self]`, and no way to forget a completion path.

Building the URL with `URLComponents` instead of string interpolation matters the moment the user types something like `AC/DC` or `Sigur Rós`: each `URLQueryItem` is percent-encoded automatically, so characters that would corrupt a hand-assembled string reach the server intact. The two `guard` statements after the fetch are cheap insurance. Casting to `HTTPURLResponse` exposes the status code, and checking it against `200...299` catches the failures that are not Swift errors at all, such as a `404` or a `500`, where the request succeeded but the payload is not what we want to decode. Only five of the roughly thirty fields in each result appear in `MusicTrack`, because `JSONDecoder` ignores JSON keys that have no matching property. We model just what the screen needs, using the same `CodingKeys` renaming we practiced in the *Codable Models* section.

The two annotations on `MusicSearchViewModel` solve two distinct problems. SwiftUI requires that state changes driving views happen on the main thread, and `@MainActor` guarantees that every property assignment in the view model does exactly that, no matter where the network response was delivered. `@Observable` handles the other direction: when `tracks` , `isSearching` , or `errorMessage` changes, the views reading those properties re-render. We cover the observable model in the *Observation Framework* section of *Chapter 5* . One small detail earns its place: trimming the term rejects whitespace-only queries before they ever reach the network.

The debounce is the part I find most satisfying, because we get it almost for free from structured concurrency. `.task(id: query)` starts its work when the view appears and, whenever `query` changes, cancels the running task before starting a new one with the fresh value. Cancellation lands in two places. If the task is still inside `Task.sleep(for:)` , the sleep throws `CancellationError` , our `catch` returns, and the network is never touched. This is what collapses a burst of keystrokes into a single request 300 milliseconds after the typing stops. If cancellation arrives later, while `URLSession` is waiting on the server, the request itself is torn down and throws, and the view model's `Task.isCancelled` check recognizes that expected failure. Either way, results from an old query can never overwrite a newer one, and because the same modifier cancels its task when the view disappears, navigating away never leaks an in-flight request.

The rest of the screen is presentation. `.searchable(text:)` gives us the platform search field, covered in depth in the *Search and Filtering* section of *Chapter 2* . The `.overlay` reads like a truth table for the screen: the order of the `if` branches sets their priority, so an in-flight

search always shows the spinner and an error outranks the empty states. The two `ContentUnavailableView` forms split the remaining cases between our custom prompt, built from a title, a symbol, and a description, and the standard `.search(text:)` layout, the same one Apple's own apps show when a query comes back empty.

See also

- `URLSession` documentation: https://x.swiftuicookbook.com/cxsx1
- `data(from:delegate:)` documentation: https://x.swiftuicookbook.com/ivjlg
- `URLComponents` documentation: https://x.swiftuicookbook.com/nmt6r
- `Task` documentation: https://x.swiftuicookbook.com/tv5oq
- `task(id:name:priority:file:line:_:)` documentation: https://x.swiftuicookbook.com/vfejh
- *Concurrency* art icle: https://x.swiftuicookbook.com/1h0ds

Combine with Remote Data

Before Swift concurrency arrived, Combine was how reactive SwiftUI apps talked to the network, and a large amount of that code is still running in production today. Beyond maintenance work, Combine models something async/await does not express as naturally: a stream of values arriving over time, like the keystrokes

of a search field, flowing through a chain of operators that transform, filter, and time them.

In this recipe, we rebuild the music search app from the previous recipe with Combine, keeping the models, the row view, and the behavior identical so the two networking styles can be compared line by line. The network call becomes a publisher built from `dataTaskPublisher(for:)`, validated with `tryMap`, and decoded with `decode(type:decoder:)`. The debounce we hand-rolled with `Task.sleep(for:)` becomes a declarative pipeline: a `@Published` query property flows through `debounce`, `removeDuplicates`, and `switchToLatest`, with `receive(on:)` and `sink` delivering results to an `ObservableObject` view model. The view observes it through `@StateObject`, and a set of `AnyCancellable` values keeps the whole subscription alive.

If you have not read the *Async/await with Remote APIs* section yet, start there: it is the modern approach we recommend for new code, and this recipe constantly refers back to it. For the fundamentals of publishers, subscribers, and `@Published`, see the *C ombine with Local Data* section of *Chapter 5*.

Getting ready

Since the data models and the track row view are identical to the ones we built in the previous recipe, we have already provided them in a starter project so you can focus on the Combine networking code. Download or clone the code for this chapter from GitHub as explained in the *Technical requirements* section, and open the **MusicSearch** projec t located in the **Starter** folder of this recipe.

How to do it...

All the user interface pieces are waiting in the starter project. We will build the networking layer as a Combine pipeline, one operator at a time: first the service that turns a search term into a publisher, then the view model that reacts to typing, and finally the view that binds them together. These are the steps:

1. Open the starter project and look at what it already contains. The `Models` folder has `MusicTrack.swift`, with the `SearchResponse` and `MusicTrack` structs that mirror the iTunes Search API payload, and `NetworkError.swift`, with the error cases we surface in the UI. The `Views` folder has `TrackRowView.swift`, the artwork-and-titles row for the results list. All three files are identical to the ones we wrote in the *Async/await with Remote APIs* section, which explains how they work.
2. Create `MusicAPIService.swift` in the `Models` folder. Note the second import: Combine is its own framework. The `searchPublisher(for:)` method builds the same query URL as before with `URLComponents`, but instead of awaiting data it returns a publisher that emits the decoded tracks:

```
import Foundation
import Combine
struct MusicAPIService {
  func searchPublisher(for term: String) -> AnyPublisher<[
    var components = URLComponents()
    components.scheme = "https"
    components.host = "itunes.apple.com"
    components.path = "/search"
    components.queryItems = [
      URLQueryItem(name: "term", value: term),
      URLQueryItem(name: "media", value: "music"),
```

```
        URLQueryItem(name: "limit", value: "25")
      ]
      guard let url = components.url else {
        return Fail(error: NetworkError.invalidURL).eraseToA
      }
      return URLSession.shared.dataTaskPublisher(for: url)
        .tryMap { data, response in
          guard let httpResponse = response as? HTTPURLRespo
            throw NetworkError.invalidResponse
          }
          guard (200...299).contains(httpResponse.statusCode
            throw NetworkError.httpError(statusCode: httpRes
          }
          return data
        }
        .decode(type: SearchResponse.self, decoder: JSONDeco
        .map(\.results)
        .mapError { error in
          error is DecodingError ? NetworkError.decodingFail
        }
        .eraseToAnyPublisher()
    }
}
```

`Fail` is the publisher equivalent of a `throw` statement, so a malformed URL still surfaces as `NetworkError.invalidURL`.

3. Create `MusicSearchViewModel.swift` in the `Models` folder. This time the class conforms to `ObservableObject` and marks its state with `@Published`, the property wrapper that gives every property a publisher of its own. The search text lives here too, as `query`, because the pipeline we build next needs to observe it:

```
import Foundation
import Combine
class MusicSearchViewModel: ObservableObject {
  @Published var query = "The Beatles"
  @Published var tracks: [MusicTrack] = []
```

```
  @Published var isSearching = false
  @Published var errorMessage: String?
  private let service = MusicAPIService()
  private var cancellables = Set<AnyCancellable>()
}
```

4. A Combine subscription terminates the moment its publisher fails, and a terminated pipeline would leave the search field dead. To keep ours alive across bad networks, add a private helper below the properties that converts success and failure alike into ordinary `Result` values, using `catch` to absorb the error:

```
private func resultPublisher(
  for term: String
) -> AnyPublisher<Result<[MusicTrack], Error>, Never> {
  guard !term.isEmpty else {
    return Just(.success([])).eraseToAnyPublisher()
  }
  return service.searchPublisher(for: term)
    .map { Result.success($0) }
    .catch { Just(.failure($0)) }
    .eraseToAnyPublisher()
}
```

   An empty term short-circuits to an empty success, which is how clearing the search field clears the list.

5. Now, the centerpiece. Add an `init()` to `MusicSearchViewModel`, after the stored properties and before the `resultPublisher(for:)` helper. The initializer subscribes to `$query` and turns typing into network requests:

```
```

```
init() {
  $query
    .map { $0.trimmingCharacters(in: .whitespaces) }
    .debounce(for: .milliseconds(300), scheduler: Dispatch
    .removeDuplicates()
    .handleEvents(receiveOutput: { [weak self] term in
      self?.isSearching = !term.isEmpty
      self?.errorMessage = nil
    })
    .map { [weak self] term in
      self?.resultPublisher(for: term)
        ?? Just(.success([])).eraseToAnyPublisher()
    }
    .switchToLatest()
    .receive(on: DispatchQueue.main)
    .sink { [weak self] result in
      self?.isSearching = false
      switch result {
      case .success(let tracks):
        self?.tracks = tracks
      case .failure(let error):
        self?.errorMessage = error.localizedDescription
      }
    }
    .store(in: &cancellables)
}
```

Read the pipeline top to bottom, one operator per line: `map` trims the term, `debounce` waits out the typing, `removeDuplicates` skips repeats, `handleEvents` flips the spinner on, `switchToLatest` switches to the newest request, `receive(on:)` hops to the main thread, and `sink` delivers the result.

6. Replace the `ContentView` struct with the search screen. Compared with the previous recipe there is no `.task(id:)` anywhere: `.searchable` binds directly to `$viewModel.query`, and every keystroke flows straight into the pipeline:

```
import SwiftUI
struct ContentView: View {
  @StateObject private var viewModel = MusicSearchViewMode
  var body: some View {
    NavigationStack {
      List(viewModel.tracks) { track in
        TrackRowView(track: track)
      }
      .listStyle(.plain)
      .overlay {
        if viewModel.isSearching {
          ProgressView("Searching...")
        } else if let errorMessage = viewModel.errorMessag
          ContentUnavailableView {
            Label("Search Failed", systemImage: "exclamati
          } description: {
            Text(errorMessage)
          }
        } else if viewModel.tracks.isEmpty && viewModel.qu
          ContentUnavailableView(
            "Search for Music",
            systemImage: "music.note",
            description: Text("Type a song or artist name.
          )
        } else if viewModel.tracks.isEmpty {
          ContentUnavailableView.search(text: viewModel.qu
        }
      }
      .navigationTitle("Music Search")
      .searchable(text: $viewModel.query, prompt: "Songs a
    }
  }
}
#Preview {
  ContentView()
}
```

The overlay states are unchanged from the previous recipe, they just read `@Published` properties now.

7. That completes the second version of the app. Run it and compare it with the async/await version: launch results, debounced typing, the empty and no-results states, everything behaves the same. To see the error state, turn off your Mac's Wi-Fi (or the device's network) and type a new search:

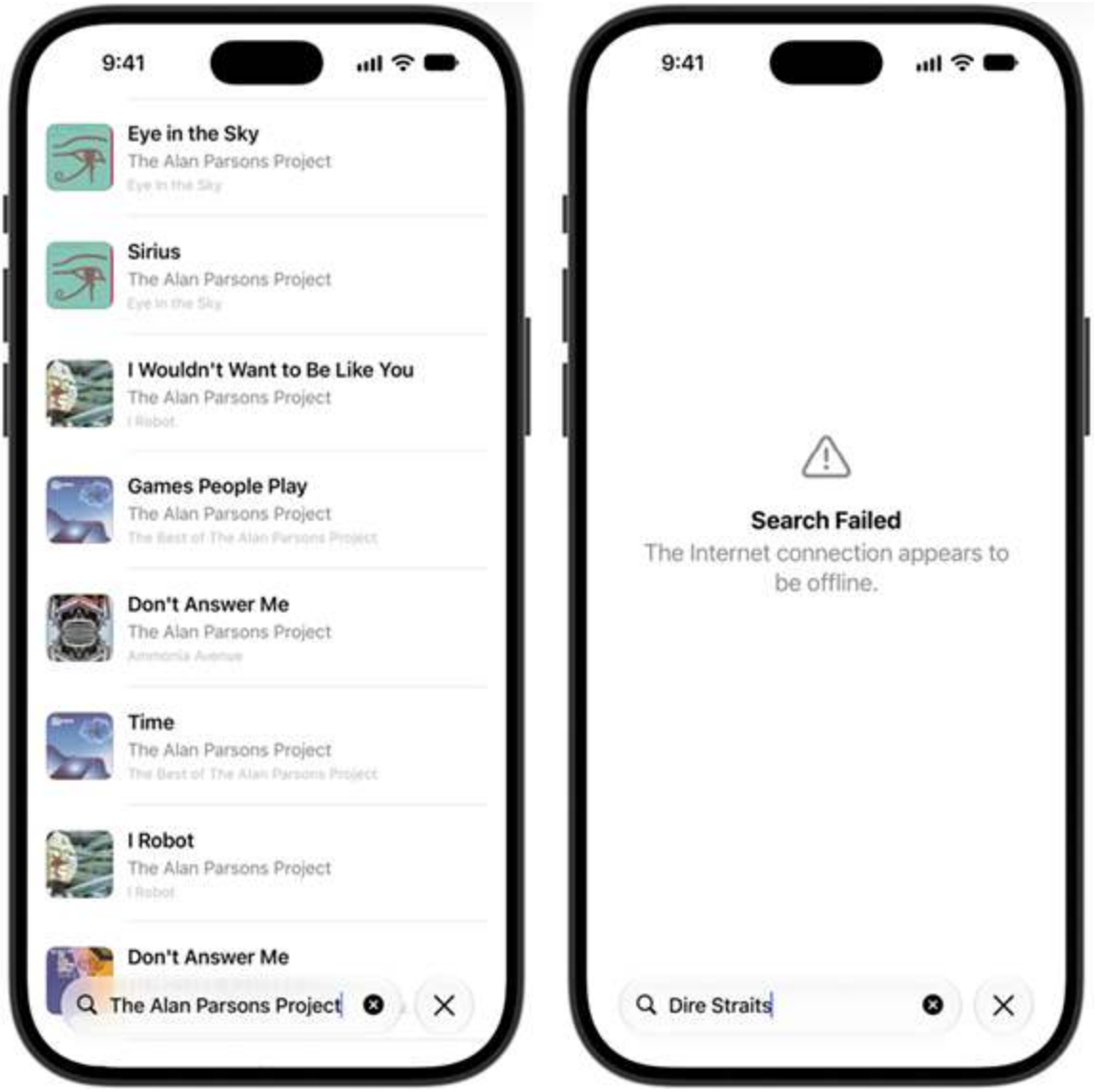


Figure 8.3: The Combine-driven sear ch results and the error state with the network turned off

How it works...

The biggest difference from the previous recipe is where the logic lives. The async/await version runs a fresh `search(for:)` every time the query changes, a sequence of steps we wrote imperatively. The Combine version declares the whole flow once, in `init()`, and then lets values run through it: `@Published` gives `query` a publisher, `$query`, and everything downstream reacts to what it emits. This pairing is also why the view model went back to `ObservableObject`: the Observation framework does not expose publishers, while `@Published` properties are publishers, exactly the glue Combine needs. We covered this style of view model in the *Combine with Local Data* section of *Chapter 5*.

Inside `searchPublisher(for:)`, each operator transforms the output of the one above it. `dataTaskPublisher(for:)` emits a single `(data, response)` tuple, `tryMap` runs the same `HTTPURLResponse` status checks we wrote in the async/await service and throws the same `NetworkError` cases, `decode` feeds the bytes to a `JSONDecoder` for us, and `map(\.results)` unwraps the track array out of `SearchResponse`. The `mapError` at the end wraps decoding problems in `NetworkError.decodingFailed`, matching the error vocabulary of the previous recipe. `eraseToAnyPublisher()` deserves a note: without it, the return type would spell out every operator in the chain, so we erase it to `AnyPublisher<[MusicTrack], Error>` and keep the signature readable.

The three operators in the middle of the pipeline are the reason Combine earns its place in this chapter. `debounce` holds each value back for 300 milliseconds and discards it if a newer one arrives, which is the behavior we built by hand with `Task.sleep(for:)`. `removeDuplicates` drops a term identical to the previous one, so

tapping in and out of the search field does not repeat a search we already ran, a refinement the async/await version does not have. And `switchToLatest` does what its name says: each incoming term becomes a new inner publisher, and subscribing to the newest one automatically cancels the request still in flight, the same guarantee `.task(id:)` gave us.

Error handling needs more care here than it did with `do-catch`, because failure is terminal for a Combine subscription. That is why `resultPublisher(for:)` never fails: its `catch` converts an error into an ordinary `Result.failure` value, the pipeline keeps running, and the `sink` decides whether to fill `tracks` or `errorMessage`. The remaining pieces are bookkeeping. `receive(on: DispatchQueue.main)` plays the role `@MainActor` played before, `[weak self]` keeps the closures from retaining the view model, and the `AnyCancellable` returned by `sink` lands in `cancellables` via `store(in:)`, keeping the subscription alive exactly as long as the view model lives.

So which style should you reach for? For a one-shot request, async/await wins on clarity, which is why it opens this chapter and why we recommend it for new code. Combine pays off when the input is a stream: search-as-you-type, live form validation, anything where operators like `debounce` and `switchToLatest` replace state you would otherwise manage by hand. In an existing codebase you will often meet both, and after building the sa me app twice you can now translate between them line by line.

See also

- `Combine` documentation: https://x.swiftuicookbook.com/tb5gd
- `URLSession.DataTaskPublisher` documentation: https://x.swiftuicookbook.com/jv7zf
- *Processing URL Session Data Task Results with Combine* article: https://x.swiftuicookbook.com/wy34d
- `debounce(for:scheduler:options:)` documentation: https://x.swiftuicookbook.com/12h3v
- `Publishers.SwitchToLatest` documentation: https://x.swiftuicookbook.com/4i9zu
- `AnyC ancellable` documentation: https://x.swiftuicookbook.com/lb0z1

AsyncImage and Caching

Most of the bytes a typical app moves over the network are not JSON, they are images. Before `AsyncImage`, showing a remote image meant downloading data with `URLSession`, decoding it into a `UIImage`, hopping back to the main thread, and finding somewhere sensible to cache it, or pulling in a third-party library to do all of that for you. `AsyncImage` folds that pipeline into a single SwiftUI view, and in iOS 27 it gains the piece that kept those old third-party libraries popular: real caching.

In this recipe, we build a photo browser backed by the free Dog API at `dog.ceo`, which requires no key. A segmented control picks a dog breed, a three-column grid shows its photos, and tapping a photo opens a full-size view. The list fetch uses patterns we already know:

a `Decodable` response, an `@Observable` view model, and a `.task(id:)` that reloads when the breed changes. The new material is all about displaying the images. We will show them with the closure-based variant of `AsyncImage` , handle every loading state with the `AsyncImagePhase` variant and an animated `Transaction` , and then adopt the three iOS 27 additions: automatic HTTP caching, the `AsyncImage(request:)` initializers with per-request cache policies, and the `asyncImageURLSession()` mod ifier that gives a whole view subtree its own sized image cache .

Getting ready

Create a new SwiftUI iOS app named **PhotoBrowser** .

How to do it...

We will build the data layer first, then the grid with thumbnails, then the full-size detail view, and once the whole app works we will layer the caching control on top, one iOS 27 API at a time. These are the steps:

1. Let's look at the data first. The Dog API answers a query like `dog.ceo/api/breed/husky/images` with the simplest payload in this chapter: a status string and an array of image URLs, nothing else:

```
{
  "message": [
    "https://images.dog.ceo/breeds/husky/n02110185_10047.j
    "https://images.dog.ceo/breeds/husky/n02110185_10116.j
    // ... more image URLs
```

```
  ],
  "status": "success"
}
```

2. Model the breeds first. Create `Breed.swift` in a `Models` folder, an enum whose raw value is the path segment the API expects and whose capitalized name feeds the UI. `CaseIterable` will drive the segmented picker, the same pattern our product categories used in the *Codable Models* section:

```
import Foundation
enum Breed: String, CaseIterable, Identifiable {
  case husky
  case pug
  case corgi
  case beagle
  case labrador
  var id: Self { self }
  var displayName: String { rawValue.capitalized }
}
```

3. Create `DogPhoto.swift` in the `Models` folder. `BreedImagesResponse` mirrors the payload, and `DogPhoto` wraps one parsed `URL` so the grid has `Identifiable` items to iterate. The `preview` sample gives us a ready-made photo, so we can see how each view looks in the preview canvas as we build it:

```
import Foundation
struct BreedImagesResponse: Decodable {
  let message: [String]
  let status: String
}
struct DogPhoto: Identifiable {
  let url: URL
```

```
  var id: URL { url }
}
extension DogPhoto {
  static var preview: DogPhoto? {
    if let url = URL(string:
      "https://images.dog.ceo/breeds/husky/n02110185_12498
      DogPhoto(url: url)
    } else {
      nil
    }
  }
}
```

4. Create `GalleryViewModel.swift` in the `Models` folder. The fetch is the async/await pattern from the *Async/await with Remote APIs* section, and the last two lines of the `do` block turn the raw strings into `[DogPhoto]`, keeping the first thirty and dropping anything that fails to parse as a `URL`:

```
import Foundation
@MainActor
@Observable
class GalleryViewModel {
  var photos: [DogPhoto] = []
  var isLoading = false
  var errorMessage: String?
  func loadPhotos(breed: Breed) async {
    let listURL = "https://dog.ceo/api/breed/\(breed.rawVa
    guard let url = URL(string: listURL) else {
      errorMessage = "The photo list URL is invalid."
      return
    }
    isLoading = true
    errorMessage = nil
    defer { isLoading = false }
    do {
      let (data, _) = try await URLSession.shared.data(fro
      let response = try JSONDecoder().decode(BreedImagesR
```

```
                                                    from: data)
      let urls = response.message.prefix(30).compactMap {
      photos = urls.map { DogPhoto(url: $0) }
    } catch {
      errorMessage = error.localizedDescription
    }
  }
}
```

5. Now, the first `AsyncImage`. Create `PhotoThumbnailView.swift` in a `Views` folder, using the closure-based variant: the content closure receives the loaded `Image`, which is where `resizable()` and `scaledToFill()` must go, and the placeholder closure fills the cell until the download finishes. A quiet gray rectangle works better here than a spinner, because thirty flickering spinners read as a problem while a soft gray flash reads as loading:

```
import SwiftUI
struct PhotoThumbnailView: View {
  let photo: DogPhoto
  var body: some View {
    AsyncImage(url: photo.url) { image in
      image
        .resizable()
        .scaledToFill()
    } placeholder: {
      Rectangle()
        .fill(.gray.opacity(0.1))
    }
    .frame(minWidth: 0, maxWidth: .infinity, minHeight: 12
    .clipped()
  }
}
#Preview {
  if let photo = DogPhoto.preview {
    PhotoThumbnailView(photo: photo)
      .frame(width: 130)
```

```
  }
}
```

6. Assemble the browsing screen. Replace the `ContentView` struct with a breed picker, the loading and error states, and a `LazyVGrid` of thumbnails. The lazy grid matters with images: cells load their photos only as they appear, instead of firing thirty downloads for rows nobody has scrolled to:

```
import SwiftUI
struct ContentView: View {
  @State private var viewModel = GalleryViewModel()
  @State private var selectedBreed = Breed.husky
  private let columns = [
    GridItem(.flexible(), spacing: 2),
    GridItem(.flexible(), spacing: 2),
    GridItem(.flexible(), spacing: 2)
  ]
  var body: some View {
    NavigationStack {
      VStack(spacing: 0) {
        breedPicker
        if viewModel.isLoading {
          Spacer()
          ProgressView("Loading photos...")
          Spacer()
        } else if let errorMessage = viewModel.errorMessag
          ContentUnavailableView {
            Label("Loading Failed", systemImage: "exclamat
          } description: {
            Text(errorMessage)
          } actions: {
            Button("Retry") {
              Task { await viewModel.loadPhotos(breed: sel
            }
          }
        } else {
          photoGrid
```

```
        }
      }
      .navigationTitle("Photo Browser")
      .task(id: selectedBreed) {
        await viewModel.loadPhotos(breed: selectedBreed)
      }
    }
  }
  private var breedPicker: some View {
    Picker("Breed", selection: $selectedBreed) {
      ForEach(Breed.allCases) { breed in
        Text(breed.displayName).tag(breed)
      }
    }
    .pickerStyle(.segmented)
    .padding(.horizontal)
    .padding(.vertical, 8)
  }
  private var photoGrid: some View {
    ScrollView {
      LazyVGrid(columns: columns, spacing: 2) {
        ForEach(viewModel.photos) { photo in
          PhotoThumbnailView(photo: photo)
        }
      }
    }
  }
}
#Preview {
  ContentView()
}
```

Preview `ContentView` now. The grid fills with huskies, and switching the picker swaps in another breed.

7. The grid shows small crops, so let's give each photo a full-size view. Create `PhotoDetailView.swift` in the `Views` folder, this time using the phase variant of `AsyncImage`: one closure receives an `AsyncImagePhase` value, and a `switch` gives each loading state its

own presentation. The `Transaction` animates the change between phases, so the photo fades in over the skeleton instead of popping:

```
import SwiftUI
struct PhotoDetailView: View {
  let photo: DogPhoto
  let breed: Breed
  @Environment(\.dismiss) private var dismiss
  var body: some View {
    NavigationStack {
      VStack(spacing: 16) {
        AsyncImage(
          url: photo.url,
          transaction: Transaction(animation: .easeInOut(d
        ) { phase in
          switch phase {
          case .empty:
            Rectangle()
              .fill(.gray.opacity(0.1))
              .aspectRatio(3 / 2, contentMode: .fit)
              .overlay {
                ProgressView("Loading full image...")
              }
          case .success(let image):
            image
              .resizable()
              .scaledToFit()
              .transition(.opacity)
          case .failure:
            ContentUnavailableView(
              "Image Unavailable",
              systemImage: "photo",
              description: Text("Could not load the full i
            )
          @unknown default:
            EmptyView()
          }
        }
        .clipShape(RoundedRectangle(cornerRadius: 16))
```

```
          .padding(.horizontal)
          Spacer()
        }
        .navigationTitle(breed.displayName)
        .navigationBarTitleDisplayMode(.inline)
        .toolbar {
          ToolbarItem(placement: .cancellationAction) {
            Button(role: .close) { dismiss() }
          }
        }
      }
    }
  }
  #Preview {
    if let photo = DogPhoto.preview {
      PhotoDetailView(photo: photo, breed: .husky)
    }
  }
```

8. Wire the detail view to the grid. Back in `ContentView`, add a `selectedPhoto` state property, present the sheet from it, and wrap each thumbnail in a `Button` that sets it:

```
@State private var viewModel = GalleryViewModel()
@State private var selectedBreed = Breed.husky
@State private var selectedPhoto: DogPhoto?
// ... the columns property
var body: some View {
  NavigationStack {
    VStack(spacing: 0) {
      // ... breed picker and state branches
    }
    .navigationTitle("Photo Browser")
    // ... existing task modifier
    .sheet(item: $selectedPhoto) { photo in
      PhotoDetailView(photo: photo, breed: selectedBreed)
    }
  }
}
```

```
// ... the breedPicker property
private var photoGrid: some View {
  ScrollView {
    LazyVGrid(columns: columns, spacing: 2) {
      ForEach(viewModel.photos) { photo in
        Button {
          selectedPhoto = photo
        } label: {
          PhotoThumbnailView(photo: photo)
        }
      }
    }
  }
}
```

The app is fully browsable now: pick a breed, scroll the grid, tap a photo, close it with the close button.

9. With the preview canvas in live mode, pay attention to what does not happen: switch to **Pug** , then back to **Husky** , and the grid returns without re-downloading a single image. We wrote no caching code for this. Before iOS 27, we would have had to implement that caching ourselves.



New in iOS 27. Every `AsyncImage` now caches its downloads automatically, honoring the server's HTTP cache headers, with no API to enable. The Dog API marks its images as immutable and lets clients keep them for 30 days, so once a photo lands on the device, the framework has no reason to download it again for a month.

Existing `AsyncImage(url:)` code gains this behavior without modification.

10. Automatic caching follows the server's rules, but we can also set our own per image. Update `PhotoThumbnailView` to build a `URLRequest` with an explicit cache policy and hand it to the new request-based initializer:

```
struct PhotoThumbnailView: View {
  let photo: DogPhoto
  private var imageRequest: URLRequest {
    URLRequest(url: photo.url, cachePolicy: .returnCacheDa
  }
  var body: some View {
    AsyncImage(request: imageRequest) { image in
      // ... existing image modifiers
    } placeholder: {
      // ... existing placeholder
    }
    .frame(minWidth: 0, maxWidth: .infinity, minHeight: 12
    .clipped()
  }
}
```



New in iOS 27. The `AsyncImage(request:)` initializers accept a `URLRequest` instead of a `URL`, so any request property is ours to set, with the cache policy being the practical one. In our thumbnail, `.returnCacheDataElseLoad` serves whatever is cached without asking the server first, and

> only goes to the network when the cache has nothing.

11. Give the full-size view the same policy. In `PhotoDetailView`, add the same `imageRequest` property below the `dismiss` environment property and pass it to the phase-based initializer:

```
private var imageRequest: URLRequest {
  URLRequest(url: photo.url, cachePolicy: .returnCacheData
}
var body: some View {
  NavigationStack {
    VStack(spacing: 16) {
      AsyncImage(
        request: imageRequest,
        transaction: Transaction(animation: .easeInOut(dur
      ) { phase in
        // ... existing phase switch
      }
      // ... existing clip shape and padding
      Spacer()
    }
    // ... existing navigation and toolbar modifiers
  }
}
```

    Because the grid and the sheet request the same URL, a photo you tap is usually already in the cache, so the full-size view skips the download, and the photo fades in over the skeleton almost at once.

12. One piece remains: deciding how much space those images may occupy. Add a static `URLSession` with its own sized `URLCache` to

`ContentView`, and attach it to the view tree with the `asyncImageURLSession()` modifier:

```
private static let imageSession: URLSession = {
  let megabyte = 1024 * 1024
  let configuration = URLSessionConfiguration.default
  configuration.urlCache = URLCache(memoryCapacity: 64 * m
                                    diskCapacity: 256 * me
  return URLSession(configuration: configuration)
}()
// ... the columns property
var body: some View {
  NavigationStack {
    VStack(spacing: 0) {
      // ... breed picker and state branches
    }
    // ... existing navigationTitle, task, and sheet modif
    .asyncImageURLSession(Self.imageSession)
  }
}
```



New in iOS 27. The `asyncImageURLSession()` modifier sets the `URLSession` that every `AsyncImage` in the subtree uses to load images, and the session's `URLCache` decides how much memory and disk those images may take. To keep an image-heavy feature's files apart from the rest of the app's cached responses, give that cache a `directory` of its own; without one, it shares the app's default on-disk store.

13. We have finally finished our app. Run it and browse: the first visit to each breed streams its photos in, revisits are instant, and a tapped photo opens its full-size view s traight from the cache:



Figure 8.4: The husky photo grid and the full-size photo view

How it works...

Every image in our app passes through the same lifecycle. When a thumbnail cell appears, its `AsyncImage` starts in the placeholder state,

the download or cache lookup runs, and the view swaps to the loaded image. The two variants we used are the same machinery with different levels of control: the content-and-placeholder variant covers the happy path in the grid, while the phase variant in the full-size view unpacks the whole `AsyncImagePhase` enum, giving `.empty`, `.success`, and `.failure` each their own presentation, with `@unknown default` future-proofing the `switch`. The `Transaction` we passed alongside it animates the phase changes, which is why the photo fades in over the skeleton. We explore transactions and transitions in depth in the *Animations, Transitions, and Matched Geometry* section of *Chapter 12*. That explicit `.failure` case also settles something we noted back in the *Async/await with Remote APIs* section: where a plain `AsyncImage` quietly leaves a placeholder when a load fails, the phase variant lets us show the user an honest error instead.

The `resizable()` and `scaledToFill()` calls live inside the content closure for a reason: they are `Image` modifiers, and `AsyncImage` itself is not an `Image`, just a view that will eventually contain one. The frame and `clipped()` on the outside belong to the cell, so the placeholder and the loaded photo occupy identical space and the grid never jumps as downloads finish. The zero minimum width is what keeps a wide photo inside its column: without it, the frame would grow to the photo's width, and `clipped()` would clip too late.

Caching is the layer iOS 27 takes off our hands. The automatic HTTP cache stores each image response in the app's standard URL cache and honors the response headers, so what it does for your app depends on your server: the Dog API marks images immutable for 30 days, which makes revisits free, while its JSON list endpoint allows a cached copy for only five minutes, and a server that sends

no cache headers promises nothing. The `AsyncImage(request:)` initializers add the second layer: a `URLRequest` carries a cache policy, and `.returnCacheDataElseLoad` means any cached copy is good enough, skip the server entirely. That policy keeps our detail view instant even after the server's expiry has passed, and it is also why the app needs no separate thumbnail size: the grid already paid for the full image, and the sheet reuses it. The third layer, `asyncImageURLSession()`, controls where all those bytes live. The `URLCache` we attached gives our photos their own 64 MB memory cache and a 256 MB disk capacity. Because we created it without a `directory`, it writes to the app's default cache folder, the same on-disk store `URLCache.shared` uses; passing a folder of its own keeps the two apart completely. And because the modifier works through the environment, the sheet inherits the session from the view that presents it.



Cache policies are not just for images. Our breed list request uses the default policy, and the Dog API lets a client reuse its JSON list for only five minutes, so once those minutes pass, switching breeds without a network connection shows the error state even though every photo of that breed is already cached. Handing the list request the same `.returnCacheDataElseLoad` policy would let every visited breed open offline, because that policy accepts a cached copy of any age. When you control the server, explicit cache headers are what make offline behavior dependable.

Before iOS 27, apps that needed this behavior built it by hand, typically an `NSCache` of decoded images behind a custom view that checked the cache before creating a `URLSession` task. That pattern is not dead, it just moved up a level: the automatic cache stores compressed image data, so a custom `NSCache` still earns its place when decoding itself is the bottleneck, such as scrolling hundreds of large images. For everything short of that, the rule of thumb after this recipe is: let the automatic cache work for you, reach for a request policy when a specific image needs different rules, and scope a sized session to any image-heavy feature.

See also

- `AsyncImage` documentation: https://x.swiftuicookbook.com/ucs2q
- `AsyncImagePhase` documentation: https://x.swiftuicookbook.com/orss4
- `init(request:scale:transaction:content:)` documentation: https://x.swiftuicookbook.com/0zvly
- `init(request:scale:content:placeholder:)` documentation: https://x.swiftuicookbook.com/at10c
- `asyncImageURLSession(_:)` documentation: https://x.swiftuicookbook.com/d35g8
- `URLCache` documentation: https://x.swiftuicookbook.com/yzp0p

9

Debugging, Testing, and Performance

I spent my first years in iOS with a mental model of debugging that SwiftUI quietly took away. In UIKit, when something went wrong, I knew which method had run and in what order, and a breakpoint in `viewDidLoad` or `layoutSubviews` usually told the rest of the story. In SwiftUI, the framework decides when a `body` runs, a slow view rarely points at the line that is slow, and the interface we would test by hand is rebuilt from state we never touch directly. The questions are the same ones we asked in UIKit: why did this update, where did the time go, and does the app still do what we specified? The tools for answering them are different, and this chapter is about those tools.

The second part of the book is about moving data through an app: observable models, bindings, tasks, persistence, and remote data. Every one of those recipes ended with a working app, and every one of them could have ended with a puzzling update, a stutter in a list, or a regression that slipped through a manual check. The three recipes here take the apps we know how to build and add the discipline of finding out what they are actually doing.

We start with a shopping list that reports its own updates. A `print()` inside `body` through the `let _ =` trick, `Self._printChanges()` in every view, `debugPrint()` and `dump()` in the actions, and an `.onChange()` log show which view updated and why. A debug-only modifier tints a row every time its

`body` runs. Then we pause the app at a breakpoint, read state with a few LLDB commands, and open Xcode's view hierarchy debugger. Next, an expense ledger with three thousand transactions is written the slow way on purpose, so we can measure it. `OSSignposter` intervals mark the data layer, and the SwiftUI template in Instruments records the SwiftUI instrument's long-update lanes. We follow those updates into the Time Profiler's call tree, flame graph, and top-functions views, which point at the hot closure, and read our own intervals back in the os_signpost track. After the fix, a second profile confirms the improvement, and we look at what the Hangs and Hitches instruments, Run Comparison, and the Organizer's metrics add once the app ships. Finally, a small country picker becomes a test subject. We mark its views with `.accessibilityIdentifier()`, write a `XCTestCase` that launches the app through an `XCUIApplication` proxy, find its `XCUIElement` values, drive them with taps, and assert on their labels with `XCTAssertTrue` and `XCTAssertEqual`; then we let Xcode record an interaction for us, replay a failing run in the Automation Explorer, and run the suite on a physical iPhone through Device Hub.

By the end of the chapter, you should be able to tell why a SwiftUI view updated, measure what an update costs on a device before and after a fix, and hand the repetitive checks of an app over to a UI test that fails when the app stops behaving as specified.

We will cover these topics in the following recipes:

- Debugging Views and State
- Instruments and Performance
- Automated UI Tests

Technical requirements

The code in this chapter is based on Xcode 27.0 and iOS 27.0. You can download and install the latest version of Xcode from the App Store. All the code examples for this chapter can be found on GitHub at https://x.swiftuicookbook.com/ch09 .

Debugging Views and State

The first time I tried to debug a SwiftUI view the way I had debugged UIKit for years, I put a `print()` in the middle of `body` and the compiler rejected it. There is no `viewDidLoad` to put a breakpoint in and no `layoutSubviews` to step through, and SwiftUI offers no obvious place to ask the question that matters most: why did this view update? The framework decides when to call `body` , and when a view flickers or the app feels sluggish, that decision is exactly what we need to see.

In this recipe, we build a small shopping list and then make it explain itself. We split the app into small views on purpose: an entry row for new items, a filter bar, and individual rows we can mark as bought. Then we add the debugging layer: a `print()` inside `body` using the `let _ =` trick, `Self._printChanges()` in every view to report which property triggered each update, `debugPrint()` and `dump()` in the actions to inspect values, `.onChange()` to log a state transition, and a debug-only view modifier that tints a row every time its `body` runs. We finish with a breakpoint, a few LLDB commands, and Xcode's view hierarchy debugger.

The logging output in this recipe is read from Xcode's console with the app running in Device Hub. The preview canvas re-evaluates views for its own reasons, so its console output would not tell us what the app is doing. Once we know which views update and why, the *Instruments and Performance* section shows how to measure what each update costs .

Getting ready

Create a new SwiftUI iOS app named **ShoppingList** .

How to do it...

We will build the app first and instrument it second. The most useful debugging output in SwiftUI is which view re-evaluated its `body` , and with each view responsible for one job, that output points at a plausible cause. These are the steps:

1. Start with the data. Create `ShoppingItem.swift` in a `Models` folder with an `Aisle` enum and the item struct. `Aisle` is `Comparable` , and because the enum has no raw values, Swift synthesizes the comparison from the declaration order, which is the order we walk the store. The sample data is listed out of aisle order on purpose, so sorting has something visible to do:

```
import Foundation
enum Aisle: CaseIterable, Comparable {
  case produce, dairy, bakery, pantry
  var label: String {
    switch self {
    case .produce: "Produce"
    case .dairy: "Dairy"
    case .bakery: "Bakery"
    case .pantry: "Pantry"
    }
  }
}
struct ShoppingItem: Identifiable {
  let id = UUID()
  var name: String
  var aisle: Aisle
  var isBought = false
}
extension ShoppingItem {
  static let sample = [
    ShoppingItem(name: "Milk", aisle: .dairy),
```

```
    // ... the remaining 5 items
  ]
}
```

The full `sample` array has six items, two of them already bought; it is in the book's GitHub repository.

2. Create `ItemRowView.swift` in a `Views` folder. The row takes a plain `ShoppingItem` value rather than a binding, a choice that will matter once we start watching updates. The `.contentShape(.rect)` modifier makes the whole row respond to taps, because the row will become the label of a button and the `Spacer` on its own has no hit area:

```
import SwiftUI
struct ItemRowView: View {
  let item: ShoppingItem
  var body: some View {
    HStack {
      Image(systemName: item.isBought ? "checkmark.circle.fill"
        .foregroundStyle(item.isBought ? Color.green : Color.sec
      VStack(alignment: .leading) {
        Text(item.name)
          .strikethrough(item.isBought)
        Text(item.aisle.label)
          .font(.caption)
          .foregroundStyle(.secondary)
      }
      Spacer()
    }
    .contentShape(.rect)
  }
}
#Preview {
  List {
    ItemRowView(item: ShoppingItem.sample[0])
    ItemRowView(item: ShoppingItem.sample[1])
  }
}
```

3. Create `AddItemView.swift` in the `Views` folder. The text field and the aisle picker keep their own `@State`, and the view appends to the list through a binding to the array, so the parent never has to know when the text changes:

```
import SwiftUI
struct AddItemView: View {
  @Binding var items: [ShoppingItem]
  @State private var name = ""
  @State private var aisle = Aisle.produce
  var body: some View {
    HStack {
      TextField("New item", text: $name)
        .textFieldStyle(.roundedBorder)
        .onSubmit(addItem)
      Picker("Aisle", selection: $aisle) {
        ForEach(Aisle.allCases, id: \.self) { aisle in
          Text(aisle.label)
        }
      }
      .labelsHidden()
      Button("Add", systemImage: "plus", action: addItem)
        .buttonStyle(.borderedProminent)
        .disabled(name.isEmpty)
    }
    .padding(.horizontal)
  }
  private func addItem() {
    let trimmedName = name.trimmingCharacters(in: .whitespaces)
    guard !trimmedName.isEmpty else { return }
    items.append(ShoppingItem(name: trimmedName, aisle: aisle))
    name = ""
  }
}
#Preview {
  @Previewable @State var items = ShoppingItem.sample
  AddItemView(items: $items)
}
```

4. Create `FilterBarView.swift` in the `Views` folder with two button-style toggles bound to flags owned by the parent:

```
import SwiftUI
struct FilterBarView: View {
  @Binding var hideBought: Bool
  @Binding var sortByAisle: Bool
  var body: some View {
    HStack {
      Toggle("Hide bought", systemImage: "eye.slash", isOn: $hic
      Toggle("Sort by aisle", systemImage: "arrow.up.arrow.down"
             isOn: $sortByAisle)
    }
    .toggleStyle(.button)
    .padding(.horizontal)
  }
}
#Preview {
  @Previewable @State var hideBought = false
  @Previewable @State var sortByAisle = true
  FilterBarView(hideBought: $hideBought, sortByAisle: $sortByAis
}
```

5. Replace the `ContentView` struct with the following, where the three `@State` properties own the list and the two flags, `visibleItems` applies the filter and the sort, and each row is a plain-style `Button` that marks its item as bought:

```
struct ContentView: View {
  @State private var items = ShoppingItem.sample
  @State private var hideBought = false
  @State private var sortByAisle = false
  private var visibleItems: [ShoppingItem] {
    let filtered = hideBought ? items.filter { !$0.isBought } :
    guard sortByAisle else { return filtered }
    return filtered.sorted { ($0.aisle, $0.name) < ($1.aisle, $1
  }
  var body: some View {
    NavigationStack {
      VStack {
        AddItemView(items: $items)
        FilterBarView(hideBought: $hideBought, sortByAisle: $sor
        List {
          ForEach(visibleItems) { item in
```

```
            Button {
              toggleBought(item)
            } label: {
              ItemRowView(item: item)
            }
            .buttonStyle(.plain)
          }
          .onDelete(perform: deleteItems)
        }
      }
      .navigationTitle("Shopping List")
    }
  }
  private func toggleBought(_ item: ShoppingItem) {
    guard let index = items.firstIndex(where: { $0.id == item.ic
      return
    }
    items[index].isBought.toggle()
  }
  private func deleteItems(at offsets: IndexSet) {
    let ids = offsets.map { visibleItems[$0].id }
    items.removeAll { ids.contains($0.id) }
  }
}
```

6. Run the app in Device Hub rather than in the preview canvas, because from here on we read Xcode's console. Add an item, tap a row to mark it as bought, and flip both toggles. The app works; now we make it talk.
7. A bare `print()` inside `body` does not compile, because the `@ViewBuilder` result builder that assembles `body` accepts only view expressions, and `print()` returns `Void`. Binding the call to `_` turns it into a declaration, which the builder allows between the views it collects. Add this as the first line of `body` in `ContentView`:

```
var body: some View {
  let _ = print("ContentView body ran with \(visibleItems.count)
  NavigationStack {
    // ... the rest of the body
```

```
    }
  }
```

Run the app and tap **Hide bought** . The console shows one line for the launch and one for the tap, and nothing while we scroll or type, because `body` only runs when SwiftUI decides the view needs a new value:

```
ContentView body ran with 6 items
ContentView body ran with 4 items
```

8. A count tells us that `body` ran, not why. Replace the `print()` with `Self._printChanges()` , a debug-only method every view has that prints which of its inputs changed since the last evaluation. It is a private API, hence the leading underscore, so we wrap the call in `#if DEBUG` and it never reaches a release build. Add the same three lines at the top of `body` in `AddItemView` , `FilterBarView` , and `ItemRowView` :

```
var body: some View {
  #if DEBUG
  let _ = Self._printChanges()
  #endif
  NavigationStack {
    // ... the rest of the body
  }
}
```

Run the app again. At launch, every view reports itself once. The parent lists its three `@State` properties, the entry view lists its binding and its local state, the filter bar lists its two bindings, and each of the six rows reports only `@self` , because a row has no dynamic properties of its own:

```
ContentView: @self, @identity, __items, __hideBought, __sortByAisle c
AddItemView: @self, @identity, _items, __name, __aisle changed.
FilterBarView: @self, @identity, _hideBought, _sortByAisle changed.
ItemRowView: @self changed.
ItemRowView: @self changed.
ItemRowView: @self changed.
ItemRowView: @self changed.
ItemRowView: @self changed.
ItemRowView: @self changed.
```

9. Clear the console with its trash button, so it holds only what the next taps print. *Cmd + K* clears it too, but only while Xcode is the active app, because in Device Hub the same shortcut turns on **Keyboard Capture** . Now tap **Hide bought** , then **Sort by aisle** . Each tap produces exactly two lines: the parent, because the flag is its state, and the filter bar, because the value behind its binding changed. Not one row reports anything, even though two rows disappeared and the rest reordered. SwiftUI moved the existing rows by identity; it did not rebuild them:

```
ContentView: __hideBought changed.
FilterBarView: @self, _hideBought changed.
ContentView: __sortByAisle changed.
FilterBarView: @self, _sortByAisle changed.
```

10. Tap the **Milk** row, then type `Eggs` in the text field and tap **Add** . The row tap changes `items` , so the parent runs, the entry row runs because it holds a binding to the array, and a single `ItemRowView` runs: the one whose `item` value is different. While we type, only `AddItemView` reports, one line per keystroke, and the parent stays silent because the text is not its state. Tapping **Add** adds one more `__name` line when the field clears, followed by updates from `ContentView` , from `AddItemView` , and from the new row's `ItemRowView` :

```
ContentView: __items changed.
AddItemView: @self, _items changed.
ItemRowView: @self changed.
AddItemView: __name changed.
AddItemView: __name changed.
AddItemView: __name changed.
AddItemView: __name changed.
AddItemView: __name changed.
ContentView: __items changed.
AddItemView: @self, _items changed.
ItemRowView: @self changed.
```

11. Knowing that a view updated is half the picture; the other half is the data. In `ContentView`, add a `dump()` call to `toggleBought(_:)` so each tap prints the item after the change:

```
private func toggleBought(_ item: ShoppingItem) {
  guard let index = items.firstIndex(where: { $0.id == item.id }
    return
  }
  items[index].isBought.toggle()
  dump(items[index])
}
```

Run the app again and tap the **Milk** row twice; the second tap returns `isBought` to `false` and produces the dump below. The dump appears before the update lines, because the action runs to completion before SwiftUI re-evaluates anything, and it shows every stored property as a tree, one level per nested value:

```
▿ ShoppingList.ShoppingItem
  - id: 84ECAEDB-1F72-43A5-8A5A-BADC1F715CFD
  - name: "Milk"
  - aisle: ShoppingList.Aisle.dairy
  - isBought: false
ContentView: __items changed.
AddItemView: @self, _items changed.
ItemRowView: @self changed.
```

12. `print()` hides one class of bug that `debugPrint()` exposes: text that looks right but is not. In `AddItemView`, print the raw field content at the top of `addItem()`:

```
private func addItem() {
  debugPrint(name)
  let trimmedName = name.trimmingCharacters(in: .whitespaces)
  guard !trimmedName.isEmpty else { return }
  items.append(ShoppingItem(name: trimmedName, aisle: aisle))
  name = ""
}
```

Type `Tea` followed by a space, clear the console, and tap **Add**. The first line is the output of `debugPrint()`, followed by the update lines that adding an item produces. Messages that iOS logs about itself, such as the software keyboard's messages when it is showing, may appear too. They do not come from our code, and right-clicking one and choosing an option under **Hide Similar Items** filters them out. Because `debugPrint()` prints the string's debug description, the quotes make the trailing space visible. A plain `print()` would emit the same trailing space without giving us a visible boundary:

```
"Tea "
```

13. Some questions are about transitions rather than values: what was the flag before, and what is it now? Back in `ContentView`, attach an `.onChange(of:)` modifier after the navigation title, using the two-parameter form that hands us both values:

```
.navigationTitle("Shopping List")
.onChange(of: hideBought) { oldValue, newValue in
  print("hideBought changed from \(oldValue) to \(newValue)")
}
```

Tap **Hide bought** . The transition prints after the two update lines. In this run, SwiftUI applied the new state and re-evaluated the affected views before it delivered the change callback:

```
ContentView: __hideBought changed.
FilterBarView: @self, _hideBought changed.
hideBought changed from false to true
```

14. Console output becomes hard to scan as the number of rows grows, so let's make updates visible in the app instead. Create `View+Debug.swift` in the `Views` folder with a modifier that paints a random tint behind a view. The tint is recomputed every time the `body` that applies it runs, so a color change means a re-evaluation, and the `#if DEBUG` block compiles the modifier down to `self` in release builds:

```
import SwiftUI
extension View {
  func debugBackground() -> some View {
    #if DEBUG
    background(Color(hue: .random(in: 0...1), saturation: 0.6,
                     brightness: 1, opacity: 0.35))
    #else
    self
    #endif
  }
}
```

15. Apply it to the row in `ItemRowView` , after the content shape:

```
HStack {
  // ... the image, the two Text views, and the Spacer
}
.contentShape(.rect)
.debugBackground()
```

Run the app in Device Hub. Every row wears its own tint at launch. Tap **Milk** and only that row changes color; tap **Sort by aisle** and the tints travel with their rows, which is the console output from step 9 made visible. Figure 9.1 shows the list at launch and right after tapping **Milk** :

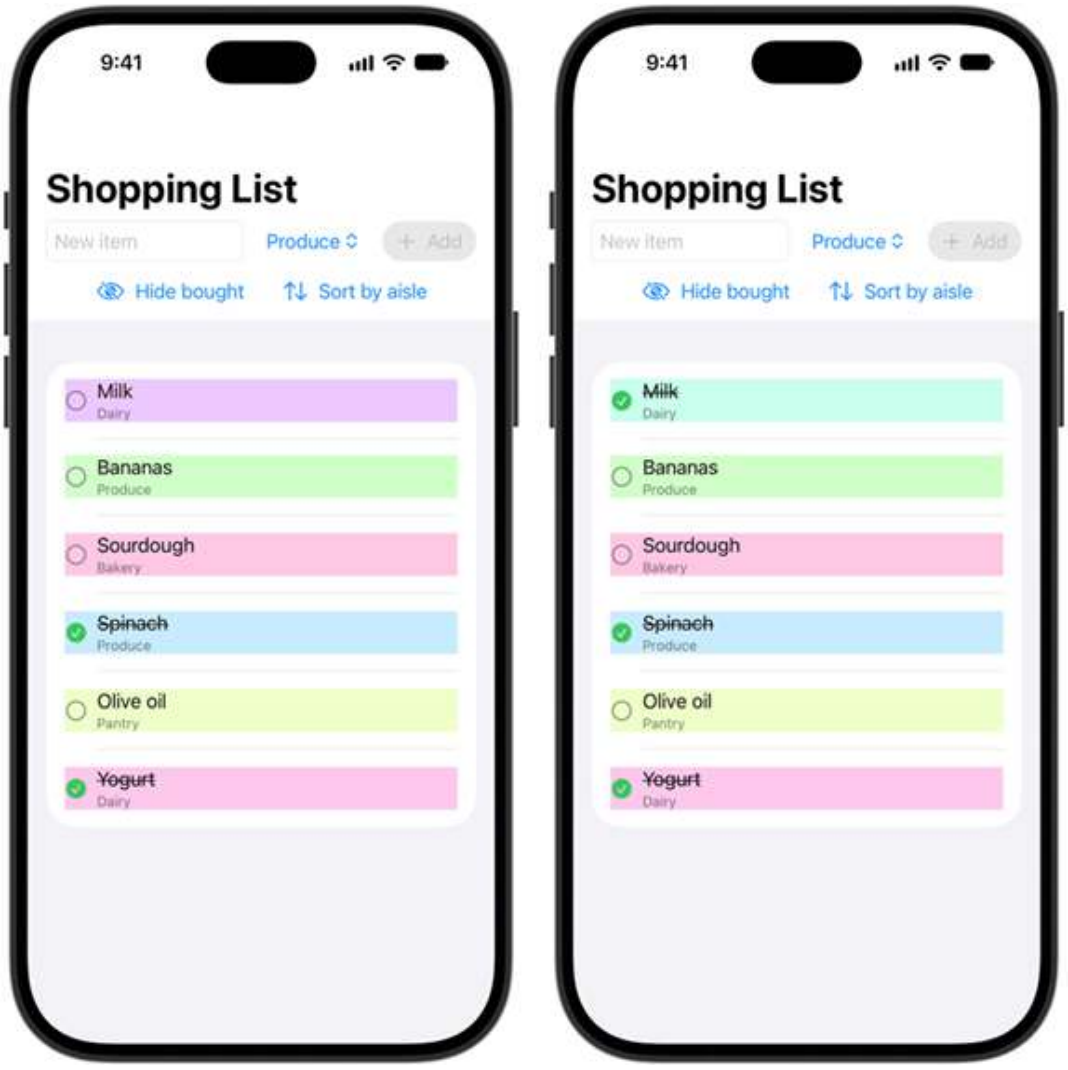


Figure 9.1: Every row is tinted at launch; tapping the Milk row repaints only that row

16. When the console is not enough, we pause the app. In `ContentView.swift`, click the line number next to `items[index].isBought.toggle()` to set a breakpoint, run the app in Device Hub, and tap a row. Xcode stops on that line with the app

frozen. The debug console at the bottom now accepts LLDB commands. `po item` prints the tapped item with all its fields, `p index` prints its position in the array, and any Swift expression works, so `po items.count` does too. Right-click the breakpoint, choose **Edit Breakpoint…** , and enter `item.name == "Milk"` in the **Condition** field: the app now pauses only for the row we are investigating. Press the continue button in the debug bar, the row of buttons just above the console, to resume:

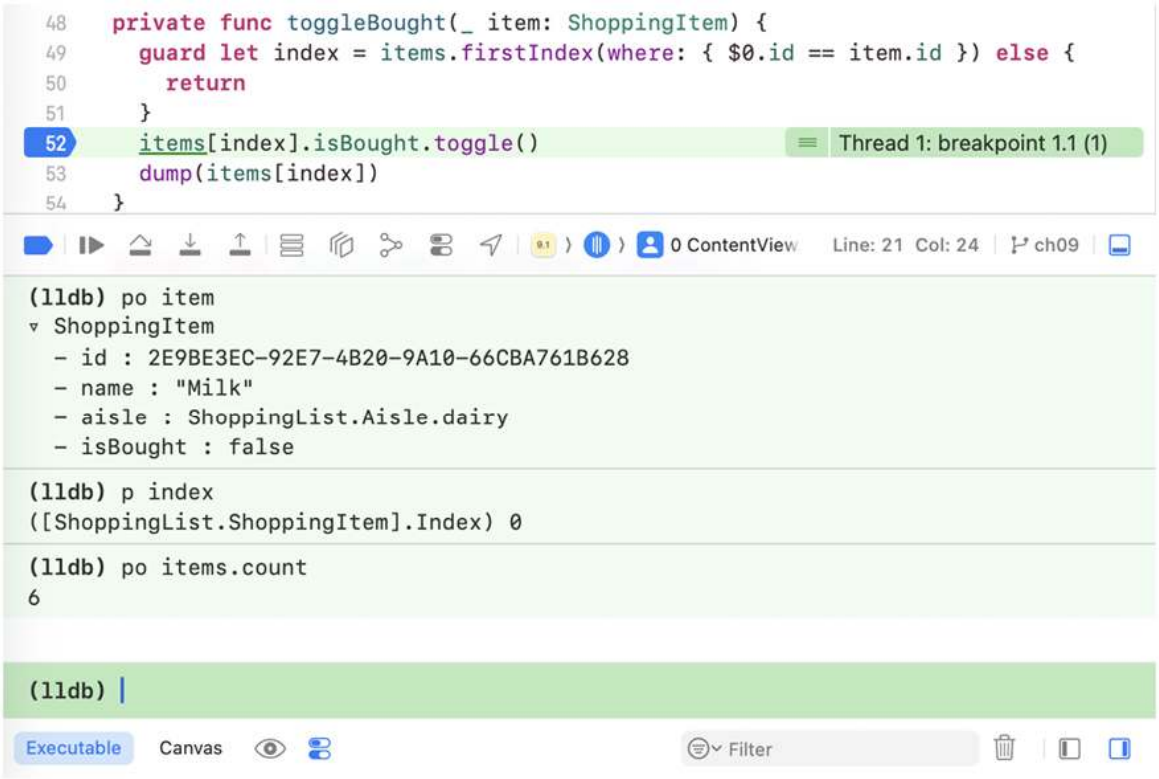


Figure 9.2: Paused at the breakpoint in toggleBought with LLDB commands in the console

17. Finally, let's look at the layout itself. With the app running, click the **Debug View Hierarchy** button in the debug bar, the icon of three stacked squares, or choose **Debug** > **View Debugging** > **Capture View Hierarchy** . Xcode pauses the app and renders every layer in an exploded 3D view we can rotate by dragging, with the whole tree listed in the navigator on the left. The tree shows the SwiftUI

structure itself: our `ContentView` sits a few levels below a hosting view, the `VStack` names `AddItemView` and `FilterBarView` inside its generic parameters, and the `List` appears as a `CollectionViewListRoot` whose `ListRepresentable` child is the collection view that renders our rows. Our short view names travel inside long wrappers such as `ModifiedContent`, which is what a chain of modifiers compiles down to. Select any element to read its frame in the inspector on the right:

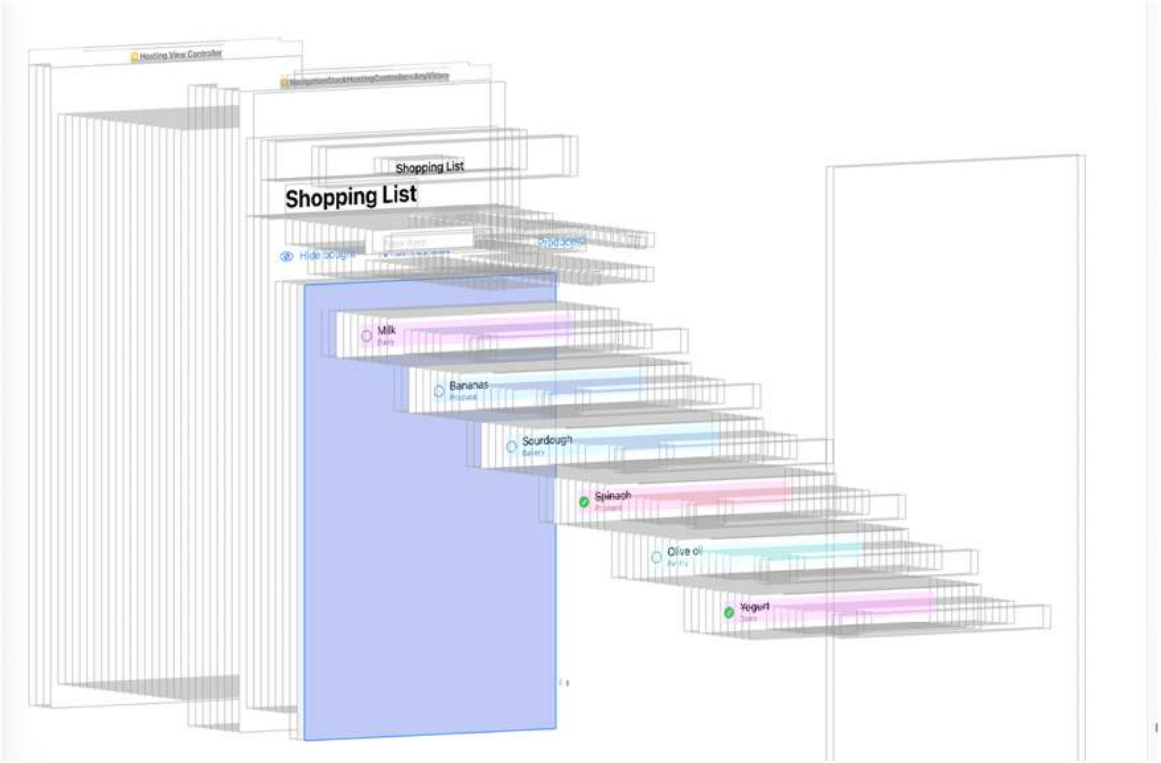


Figure 9.3: The view hierarchy debugger showing the shop ping list's layers

How it works…

The `let _ = print()` line in step 7 works because `body` is assembled by the `@ViewBuilder` result builder, which accepts declarations between the views it collects; the `_` discards the `Void` that `print()` returns and the builder never sees it. `Self._printChanges()` uses the same door, and its output is worth learning to read. Each line names the view type and then the inputs that differ from the previous evaluation: a dynamic property is listed by

the name of its storage, `@self` means the parent handed us a struct value that compares differently from the last one, and `@identity` means the view took a new place in SwiftUI's view graph, which is why in our runs it appears at launch and not afterward.

The number of underscores tells us the kind of property. Since the `@State` macro introduced with the iOS 27 SDK synthesizes its backing storage under a two-underscore name, a `@State` property prints as `__items`, while a `@Binding` keeps its single underscore and prints as `_items`. The rows report only `@self` because they have no dynamic properties: a row can change only if its parent gives it a different `item`. The underscore prefix on the method is Apple's way of marking it private and unsupported, so it belongs inside `#if DEBUG` and never in code that ships. Its sibling `Self._logChanges()` sends the same report to the unified logging system instead of the console, at the info level under the `com.apple.SwiftUI` subsystem and a `Changed Body Properties` category, which is the filter to use when we would rather read the output in the Console app. We cover the `@State` macro and the ownership rules behind these properties in the *State and Binding Basics* section of *Chapter 5*.

Read together, the console lines describe how well the app's data flows. A row re-evaluates only when its own `item` value changes, because SwiftUI compares the stored properties of a view field by field before it bothers calling `body`. Filtering and sorting produced no row output at all because our `ForEach` identifies rows by the `UUID` in `ShoppingItem`, so SwiftUI matched the old rows to the new positions and moved them. The one input that costs more than it needs to is the binding in `AddItemView`: a binding to the whole array makes that view re-evaluate whenever any item changes, even though its `body` never reads the items. The alternative I tried first was worse. Handing the view an `onAdd` closure instead made it re-evaluate on every `ContentView` update, filter taps included, because SwiftUI cannot compare closures and has to assume they changed. When

the output points at an update we want to eliminate, the fix is a narrower input, and when it points at an update we want to measure, the *Instruments and Performance* section picks up where the console leaves off.

The three printing functions differ in what they ask a value to describe. `print()` uses a value's `description`, the text meant for people, which is why it rendered our trailing space as nothing. `debugPrint()` uses `debugDescription`, the text meant for developers, so strings arrive quoted and optionals arrive wrapped, and that difference helps expose a whole category of comparison bugs. `dump()` does not ask the value for a description; it uses Swift's `Mirror` to walk the stored properties, or whatever a `CustomReflectable` type chooses to expose, and prints them as a tree, which is the fastest way to see inside a nested struct without writing a description for it. The order of the lines is informative too. The `dump()` in `toggleBought(_:)` printed before any `_printChanges()` line because an action runs to completion before SwiftUI schedules the update it caused, and the `.onChange(of:)` message printed after the update lines because the callback is delivered once the new state has been applied and the views that read it have been re-evaluated. Both orderings held in every run of this recipe; a log that differs is worth tracing back to the code that mutated the state and to the callbacks around it, to see what ran in what order.

The `debugBackground()` modifier, a `View` extension of the kind we wrote in the *Reusable Views and Modifiers* section of *Chapter 4*, is a compile-time switch, not a runtime one. `#if DEBUG` leaves exactly one implementation in each build: in a Debug build the function returns `background()` with a fresh random color, and in a Release build it returns `self` and costs nothing, not even a modifier in the view tree. That is different from a runtime conditional modifier, which switches between two view types and resets the view's identity and state every time the condition flips. The color itself carries no meaning; the change of color is the signal, because `Color(hue:saturation:brightness:opacity:)` is evaluated only when the `body`

that contains it runs. For a single view, `_printChanges()` says more, but for a list of rows a glance at which tints moved and which one changed is faster than reading the console.

Breakpoints and the view debugger complete the picture from outside the code. Once the app is paused, `po` prints an object or a struct using its debug description, `p` shows the raw value with its type, and both accept any Swift expression that is valid in the current frame, so `po items.filter { $0.isBought }.count` works as well as `po item`. A conditional breakpoint keeps the pause from firing on every tap in a busy list, and a symbolic breakpoint on a UIKit selector catches events that never pass through our code. The view hierarchy debugger shows both sides of the bridge: the SwiftUI tree, with our own types wrapped in the generic names that modifier chains compile down to, and the UIKit containers SwiftUI renders into, such as the navigation controller behind our `NavigationStack` and the collection view behind our `List`. The match with our code is loose rather than one-to-one, but frames, clipping, and layers that render offscreen are all visible there, which is precisely what the console cannot show.

See also

- `print(_:separator:terminator:)` documentation: https://x.swiftuicookbook.com/179ax
- `debugPrint(_:separator:terminator:)` documentation: https://x.swiftuicookbook.com/uu8n0
- `dump(_:name:indent:maxDepth:maxItems:)` documentation: https://x.swiftuicookbook.com/ssd8v
- *Setting breakpoints to pause your running app* article: https://x.swiftuicookbook.com/gcfev

- *Diagnosing and resolving bugs in your running app* article: https://x.swiftuicookbook.com/ivhwk
- Demystify SwiftUI (WWDC 2021): https://x.swiftuicookbook.com/r0ioe

Instruments and Performance

A slow SwiftUI view rarely announces which line is slow. I have watched a team spend an afternoon arguing about whether a list stuttered because of the data model, the formatters, or SwiftUI itself, when twenty minutes with Instruments would have settled it. The habit this recipe teaches is to measure first: record what the app does during a real scroll, read what the tools attribute the time to, change one thing, and record again.

We build an expense ledger with three thousand seeded transactions grouped by month, written the way a first draft often is: each row formats its own date and amount, and computes its share of the month by scanning every expense in that month. With `OSSignposter` intervals around the data layer, we profile the app with the SwiftUI template in Instruments, read the SwiftUI instrument's long-update lanes, the Time Profiler's call tree and flame graph, and our own signposts, then move the work out of the view layer and profile again to confirm the improvement.

Profiling belongs on a physical iPhone. The numbers in this recipe come from an iPhone Air running iOS 27. The SwiftUI instrument records nothing in the simulator, so wherever a step depends on the device, we say so and describe what the simulator can still show.

Getting ready

Create a new SwiftUI iOS app named **ExpenseLedger** . If an iPhone is available, connect it; the recipe explains what changes when only the

simulator is available. The new project targets iOS 27, so the iPhone must run iOS 27. The first run on a device also needs a development team, which Xcode offers in the target's **Signing & Capabilities** tab once we sign in with an Apple Account in its settings, and Developer Mode, which the iPhone turns on in **Settings** > **Privacy & Security** . This recipe's code on GitHub ships as two projects. The first draft we profile, as it stands after step 5, is in the **Starter** folder, and the finished app is in the **Complete** folder, so both can be recorded without retyping a line. The two Instruments recordings behind the numbers in this recipe can be downloaded as well, `v1.trace` for the first draft from https://x.swiftuicookbook.com/ihd3g_ and `v2.trace` for the finished app from https://x.swiftuicookbook.com/bg0os_, about 125 MB and 180 MB, so every figure in the recipe can be opened and explored in Ins truments without a device.

How to do it...

We build the slow version of the app on purpose, because a recipe that starts from clean code has nothing to measure. The data layer comes first; then we write the views the way they often appear in a first draft, profile that version, and apply the fix. These are the steps:

1. Start with the data. Create `Expense.swift` in a `Models` folder with a category enum and the expense struct. Each category carries a label for the picker and an SF Symbol name for the row, and the integer `id` doubles as a receipt number, which keeps the sample data readable in the debugger:

```
import Foundation
enum ExpenseCategory: CaseIterable {
  case groceries, dining, transport, utilities, entertainment, h
  var label: String {
    switch self {
```

```
    case .groceries: "Groceries"
    case .dining: "Dining"
    case .transport: "Transport"
    case .utilities: "Utilities"
    case .entertainment: "Entertainment"
    case .health: "Health"
    }
  }
  var symbolName: String {
    switch self {
    case .groceries: "cart"
    case .dining: "fork.knife"
    case .transport: "car"
    case .utilities: "bolt"
    case .entertainment: "ticket"
    case .health: "cross.case"
    }
  }
}
struct Expense: Identifiable {
  let id: Int
  let date: Date
  let merchant: String
  let category: ExpenseCategory
  let amount: Double
}
```

2. Create `LedgerGenerator.swift` in the `Models` folder. A seeded generator produces the same three thousand expenses on every launch, which matters once we compare two recordings, and it is the linear congruential generator behind the star field in the *Canvas and Custom Rendering* section of *Chapter 12*. The expenses span the twenty-four months before the end of August 2026 and come out sorted newest first:

```
import Foundation
struct SeededGenerator: RandomNumberGenerator {
  private var state: UInt64
  init(seed: UInt64) {
    state = seed
```

```
    }
    mutating func next() -> UInt64 {
      state = state &* 6364136223846793005 &+ 1442695040888963407
      return state
    }
  }
  enum LedgerGenerator {
    static let merchants: [ExpenseCategory: [String]] = [
      .groceries: ["Fresh Market", "Corner Grocer", "Farm Stand"],
      // ... the remaining 5 categories
    ]
    static func expenses(count: Int, months: Int = 24) -> [Expense
      var generator = SeededGenerator(seed: 1972)
      let calendar = Calendar.current
      let endComponents = DateComponents(year: 2026, month: 8, day
      let end = calendar.date(from: endComponents) ?? .now
      let start = calendar.date(byAdding: .month, value: -months,
      let span = end.timeIntervalSince(start)
      return (1...count).map { number in
        let categories = ExpenseCategory.allCases
        let category = categories.randomElement(using: &generator)
        let names = merchants[category] ?? []
        let merchant = names.randomElement(using: &generator) ?? "
        let offset = TimeInterval.random(in: 0..<span, using: &gen
        let amount = Double.random(in: 3...240, using: &generator)
        return Expense(id: number, date: start.addingTimeInterval(
                       merchant: merchant, category: category, amo
      }
      .sorted { $0.date > $1.date }
    }
  }
```

The full `merchants` dictionary lists three merchants for each of the six categories; it is in the book's GitHub repository.

3. Create `LedgerStore.swift` in the `Models` folder. The store owns the expenses and the selected category, and it is the first place we instrument. Add `import os` below the template's `import Foundation`, because `OSSignposter` lives in the `os` module. The `load()` method wraps the generation in a signpost interval: `beginInterval(_:)` returns

a state value that `endInterval(_:_:)` needs to close the same interval, and both calls cost almost nothing when Instruments is not recording:

```
import Foundation
import os
@MainActor
@Observable
final class LedgerStore {
  var selectedCategory: ExpenseCategory?
  private(set) var expenses: [Expense] = []
  private let signposter = OSSignposter(
    subsystem: "com.SwiftUICookbook.ExpenseLedger", category: "l
  )
  func load() {
    let generating = signposter.beginInterval("Generate expenses
    expenses = LedgerGenerator.expenses(count: 3000)
    signposter.endInterval("Generate expenses", generating)
  }
}
```

4. Create `ExpenseRowView.swift` in a `Views` folder. This is the first draft on purpose: the row receives its expense together with every expense of its month, and three computed properties build a `DateFormatter` or a `NumberFormatter` each time they are read. Nothing here is wrong in isolation, and that is exactly what makes it worth measuring:

```
import SwiftUI
struct ExpenseRowView: View {
  let expense: Expense
  let monthExpenses: [Expense]
  private var dateText: String {
    let formatter = DateFormatter()
    formatter.dateStyle = .medium
    return formatter.string(from: expense.date)
  }
  private var amountText: String {
    let formatter = NumberFormatter()
    formatter.numberStyle = .currency
    formatter.currencyCode = "USD"
    return formatter.string(from: expense.amount as NSNumber) ?
```

```
  }
  private var shareText: String {
    let total = monthExpenses.reduce(0) { $0 + $1.amount }
    let formatter = NumberFormatter()
    formatter.numberStyle = .percent
    formatter.maximumFractionDigits = 1
    return formatter.string(from: (expense.amount / total) as NS
  }
  var body: some View {
    HStack {
      Image(systemName: expense.category.symbolName)
        .frame(width: 28)
        .foregroundStyle(.tint)
      VStack(alignment: .leading) {
        Text(expense.merchant)
        Text(dateText)
          .font(.caption)
          .foregroundStyle(.secondary)
      }
      Spacer()
      VStack(alignment: .trailing) {
        Text(amountText)
          .monospacedDigit()
        Text(shareText)
          .font(.caption)
          .foregroundStyle(.secondary)
      }
    }
  }
}
#Preview {
  let expenses = LedgerGenerator.expenses(count: 20)
  List {
    ExpenseRowView(expense: expenses[0], monthExpenses: expenses
    ExpenseRowView(expense: expenses[1], monthExpenses: expenses
  }
}
```

5. Replace the `ContentView` struct with the following, where `visibleExpenses` applies the category filter, `months` collects the distinct month starts, and `expenses(in:)` filters the visible expenses down to one month. The list builds a section for each month, calling

`expenses(in:)` once to produce its rows and again for every row to supply `monthExpenses`. The toolbar `Picker` binds straight into the store's `selectedCategory`:

```
struct ContentView: View {
  @State private var store = LedgerStore()
  private var visibleExpenses: [Expense] {
    guard let category = store.selectedCategory else { return st
    return store.expenses.filter { $0.category == category }
  }
  private var months: [Date] {
    let calendar = Calendar.current
    let starts = visibleExpenses.map {
      calendar.dateInterval(of: .month, for: $0.date)?.start ??
    }
    return Array(Set(starts)).sorted(by: >)
  }
  private func expenses(in month: Date) -> [Expense] {
    let calendar = Calendar.current
    return visibleExpenses.filter {
      calendar.isDate($0.date, equalTo: month, toGranularity: .m
    }
  }
  var body: some View {
    NavigationStack {
      List(months, id: \.self) { month in
        Section {
          ForEach(expenses(in: month)) { expense in
            ExpenseRowView(expense: expense, monthExpenses: expe
          }
        } header: {
          Text(month.formatted(.dateTime.month(.wide).year()))
        }
      }
      .navigationTitle("Expense Ledger")
      .toolbar {
        ToolbarItem(placement: .topBarTrailing) {
          Picker("Category", selection: $store.selectedCategory)
            Text("All Categories").tag(ExpenseCategory?.none)
            ForEach(ExpenseCategory.allCases, id: \.self) { cate
              Text(category.label).tag(ExpenseCategory?.some(cat
            }
          }
        }
```

```
        }
        .onAppear {
          store.load()
        }
      }
    }
  }
```

6. Run the app on the iPhone, or on a simulated iPhone in Device Hub if no device is connected, because Instruments attaches to a running process and cannot record a preview. Scroll through a few months and use the category menu in the toolbar to switch to **Dining** and back to **All Categories** . On the device the list stutters while it scrolls and pauses noticeably after each category change. That is the version we are going to measure, and the project in the **Starter** folder of this recipe's code is exactly this state.
7. Choose **Product** > **Profile** , or press *Cmd + I* . Xcode builds the app with the Release configuration and hands it to Instruments, which opens its template chooser. Pick the **SwiftUI** template, which bundles the SwiftUI instrument with the Time Profiler and the Hangs and Hitches instruments. Before recording, click the **Instrument** button with the plus sign above the track list and add the **os_signpost** instrument to the document, so our own intervals land in the same recording.

   Press the record button, scroll for a while, switch the category to **Dining** and back, and stop after about thirty seconds. Figure 9.4 shows the SwiftUI track of that recording, with the long updates colored by severity:

   > The SwiftUI template refuses to record against the simulator, because its Hitches instrument is device-only, and the SwiftUI instrument records nothing

there even when added by hand. On the simulator, start from the **Blank** template and add the **Time Profiler** and **os_signpost** instruments with the same button. Steps 9 and 10 work unchanged; the timings differ, because the Mac's processor is doing the work. To see the SwiftUI instrument's data anyway, open the `v1.trace` recording from the *Getting ready* section.

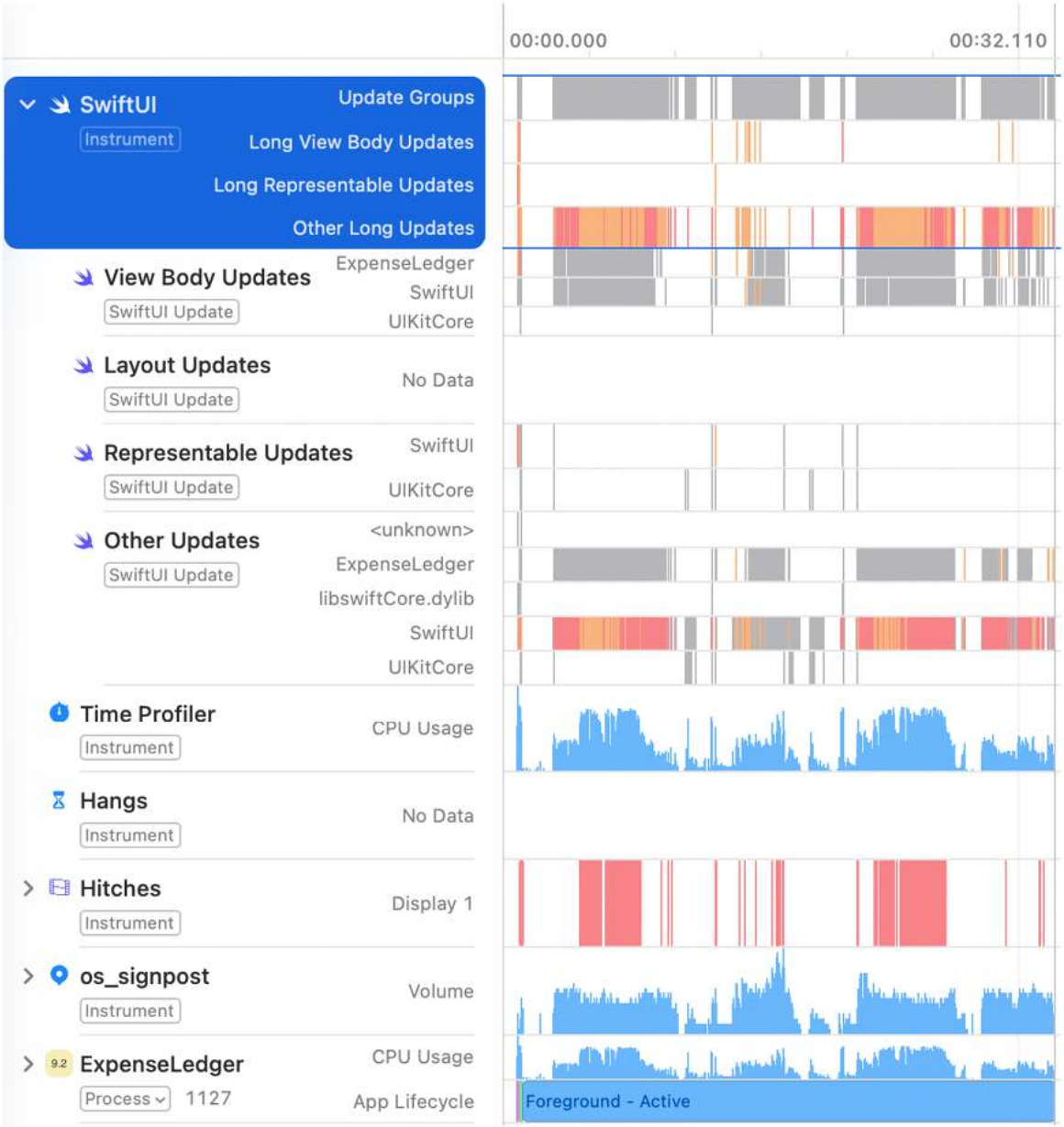


Figure 9.4: The SwiftUI instrument's lanes for the first draft, with long updates marked in orange and red

8. Select the SwiftUI track and read its lanes, which Figure 9.4 shows for our recording. The **Update Groups** lane clusters the work SwiftUI did in response to each event, and the three lanes below it single out the updates that ran long: **Long View Body Updates** , **Long Representable Updates** , and **Other Long Updates** . At first the whole recording is squeezed into the window and the markers are too thin to read, so hold *Option* while scrolling over the lanes to zoom in until the markers widen into rectangles, and hover over one to see its duration. In Figure 9.4, the **Other Long Updates** lane is almost solid orange and red, while **Long View Body Updates** shows only a few thin marks, and the detail area below the tracks puts numbers on that picture. In our recording, the worst update groups took close to 100 ms, well past the 8 ms budget of a 120 Hz frame, and the detail area lists close to 900 updates marked with high severity. The surprise is where they land. `ExpenseRowView` evaluated its `body` more than 1,600 times, never longer than a millisecond, while the longest single updates, up to 59 ms each, are attributed to the `List` itself. The reason is that our expensive code runs inside the closures the `List` and the `ForEach` call to build their content, so the instrument charges it to the container. The SwiftUI instrument tells us the cost lands in the list; to find the line, we need the Time Profiler.
9. Select the **Time Profiler** track. Its detail area opens as a call tree, one thread per top-level row, and two checkboxes behind the **Call Tree** button at the bottom make it readable: **Invert Call Tree** , which lists the functions where time was actually spent, and **Hide System Libraries** , which folds SwiftUI's own frames away. The top entry is the closure inside `ContentView.expenses(in:)` . Together with the calendar comparisons it makes, it accounts for 5.8 of the 12.7 seconds the main thread was busy during the recording, close to half of

everything it did. Below it come the `dateText` , `amountText` , and `shareText` getters and, right under them, the Foundation formatter machinery they create on every call. Click **Flame Graph** at the right end of the detail area's header to see the same recording as a shape, as in Figure 9.5. Each bar is a function, the bars under it are the functions it called, and its width is its share of the main thread's time, read against the percentage scale at the top. The bars labeled `ContentView.expenses(in:)` and its closure fill almost half of that width, scanning the month over and over, and the narrow column of bars beside them is the formatters:

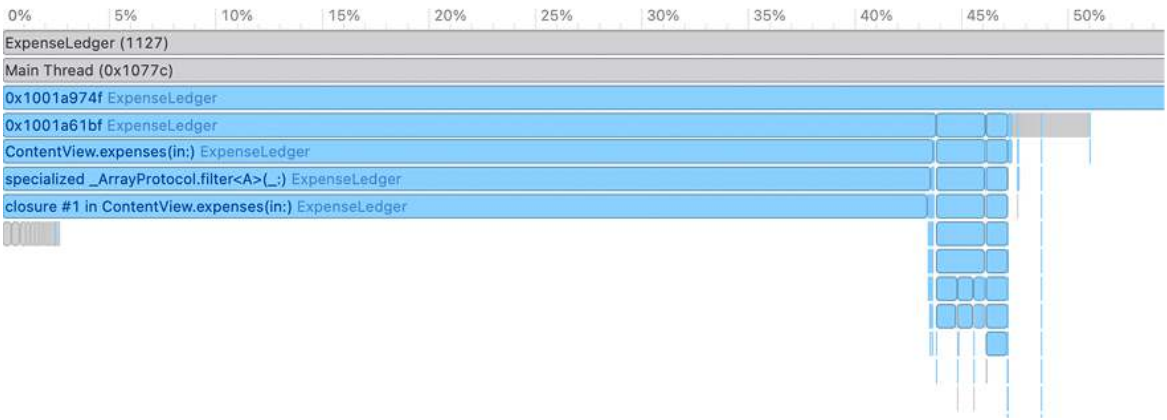


Figure 9.5: The flame graph of the first draft, dominated by expenses(in:)



New in Xcode 27. Instruments 27 gives one CPU profile three views, one button each at the right end of the detail area's header: **Call Tree** for the hierarchy we just read, **Flame Graph** (in Instruments since Xcode 16), and **Top Functions** , which ranks the functions by the time spent inside them. All three read the same recording, so we switch between them without profiling again.

10. Select the **os_signpost** track. Our `Generate expenses` interval appears once, right after launch, and lasts about two milliseconds: building three thousand expenses is not the problem, and the signpost proves it without any guessing. That is the point of instrumenting the data layer before profiling; it puts a clear boundary between the work we own and the work the views trigger.
11. The profile exposes one design flaw: the row and the list repeatedly compute values that the data layer could prepare once. Create `LedgerMonth.swift` in the `Models` folder with two value types that hold display-ready text. `LedgerRow` carries the strings the row will show, `LedgerMonth` carries a month's title, total, and rows, and both are `Equatable`, so SwiftUI can tell an unchanged row from a changed one by comparing values:

```
import Foundation
struct LedgerRow: Identifiable, Equatable {
  let id: Int
  let merchant: String
  let symbolName: String
  let dateText: String
  let amountText: String
  let shareText: String
}
struct LedgerMonth: Identifiable, Equatable {
  let id: Date
  let title: String
  let totalText: String
  let rows: [LedgerRow]
}
extension LedgerRow {
  static let sample = LedgerRow(
    id: 1,
    merchant: "Fresh Market",
    symbolName: "cart",
    dateText: "Aug 30, 2026",
    amountText: "$42.10",
    shareText: "3.2%"
  )
}
```

12. Replace the `LedgerStore` class with a version that does the grouping and formatting itself. The store now prepares `months` for the view while retaining `expenses` as its source data. Changes to `selectedCategory` call `regroup()` through `didSet`, and a second signpost interval keeps the relocated work measurable. The `Dictionary(grouping:by:)` initializer builds the month buckets in one pass, and the `formatted()` APIs replace the formatter objects:

```
@MainActor
@Observable
final class LedgerStore {
  var selectedCategory: ExpenseCategory? {
    didSet { regroup() }
  }
  private(set) var months: [LedgerMonth] = []
  private(set) var expenses: [Expense] = []
  private let signposter = OSSignposter(
    subsystem: "com.SwiftUICookbook.ExpenseLedger", category: "L
  )
  func load() {
    let generating = signposter.beginInterval("Generate expenses
    expenses = LedgerGenerator.expenses(count: 3000)
    signposter.endInterval("Generate expenses", generating)
    regroup()
  }
  private func regroup() {
    let grouping = signposter.beginInterval("Group by month")
    var visible = expenses
    if let selectedCategory {
      visible = expenses.filter { $0.category == selectedCategor
    }
    let calendar = Calendar.current
    let groups = Dictionary(grouping: visible) {
      calendar.dateInterval(of: .month, for: $0.date)?.start ??
    }
    months = groups.keys.sorted(by: >).map { monthStart in
      let monthExpenses = groups[monthStart, default: []]
      let total = monthExpenses.reduce(0) { $0 + $1.amount }
      let rows = monthExpenses.map { expense in
        let share = expense.amount / total
```

```
        return LedgerRow(
          id: expense.id,
          merchant: expense.merchant,
          symbolName: expense.category.symbolName,
          dateText: expense.date.formatted(date: .abbreviated, t
          amountText: expense.amount.formatted(.currency(code: '
          shareText: share.formatted(.percent.precision(.fractic
        )
      }
      return LedgerMonth(
        id: monthStart,
        title: monthStart.formatted(.dateTime.month(.wide).year(
        totalText: total.formatted(.currency(code: "USD")),
        rows: rows
      )
    }
    signposter.endInterval("Group by month", grouping)
  }
}
```

13. Replace the `ExpenseRowView` struct and its preview with a row that only displays what it is given. It takes a `LedgerRow` and contains no computed formatting properties, so its `body` only reads display-ready values. Xcode complains in `ContentView.swift`, because the list still constructs the old row; we fix that in the next step:

```
struct ExpenseRowView: View {
  let row: LedgerRow
  var body: some View {
    HStack {
      Image(systemName: row.symbolName)
        .frame(width: 28)
        .foregroundStyle(.tint)
      VStack(alignment: .leading) {
        Text(row.merchant)
        Text(row.dateText)
          .font(.caption)
          .foregroundStyle(.secondary)
      }
      Spacer()
      VStack(alignment: .trailing) {
```

```
        Text(row.amountText)
          .monospacedDigit()
        Text(row.shareText)
          .font(.caption)
          .foregroundStyle(.secondary)
      }
    }
  }
}
#Preview {
  List {
    ExpenseRowView(row: .sample)
  }
}
```

14. Back in `ContentView.swift`, replace the `ContentView` struct with the following. This removes the three helpers and makes the list iterate over `store.months`. Each section header now shows the month's total as well, which costs nothing because the store already formatted it. The toolbar and the `onAppear` are unchanged:

```
struct ContentView: View {
  @State private var store = LedgerStore()
  var body: some View {
    NavigationStack {
      List(store.months) { month in
        Section {
          ForEach(month.rows) { row in
            ExpenseRowView(row: row)
          }
        } header: {
          HStack {
            Text(month.title)
            Spacer()
            Text(month.totalText)
          }
        }
      }
      .navigationTitle("Expense Ledger")
      .toolbar {
        // ... the category Picker from step 5
      }
```

```
        .onAppear {
          store.load()
        }
      }
    }
  }
```

15. Profile the fixed app with *Cmd + I* , using the same template, the os_signpost instrument, and the same thirty seconds of scrolling and category switching, so the two recordings can be read against each other. The **Complete** folder contains this finished project, so we can record it without typing. On a device, compare both the Time Profiler entries and the SwiftUI long-update counts; in the simulator, compare only the Time Profiler and signpost tracks, because the SwiftUI metrics are unavailable there.

    In the Time Profiler, with **Invert Call Tree** and **Hide System Libraries** checked, the `expenses(in:)` closure and the formatter getters are gone entirely; the only entries left under our module are `regroup()` and `load()` . On the device used for this recipe, the `Group by month` signpost measures 7 ms when switching to **Dining** and 34 ms when switching back to all three thousand expenses. Across the whole recording, main-thread time spent in our code and the Foundation calls it makes falls from 6.3 seconds, half of everything the main thread did, to under 100 ms. The SwiftUI instrument's count of high-severity updates drops from close to 900 to 19, and the repeated work we targeted no longer appears in any of them. The long-update lanes that were solid orange in Figure 9.4 are now almost empty.

    > **New in Xcode 27.** Instruments 27 can compute this difference for us. Run Comparison works on two runs recorded into the same document: the second run is



added with **Next Recording** rather than from Xcode, which always opens a new document, and a compare button then appears in the detail area's header. Choosing the baseline run adds a comparison whose Call Tree, Flame Graph, and Top Functions views show each function's change in time, with regressions in red and improvements in green in the flame graph. Two separate documents cannot be compared, which is why this recipe reads the two recordings side by side instead.

16. Run the finished ledger on the same target we profiled, because smoothness is only worth judging where we measured it. The repeated stutter from scanning and formatting during row construction is gone, although a fast flick still produces a few hitches in the recording. Switching categories rebuilds the whole list in a single short pause, because the work now happens exactly once, inside `regroup()`, rather than once per row. Figure 9.6 shows the finished ledger. For the complete code, see the **Complete** project in the book's GitHub repository.

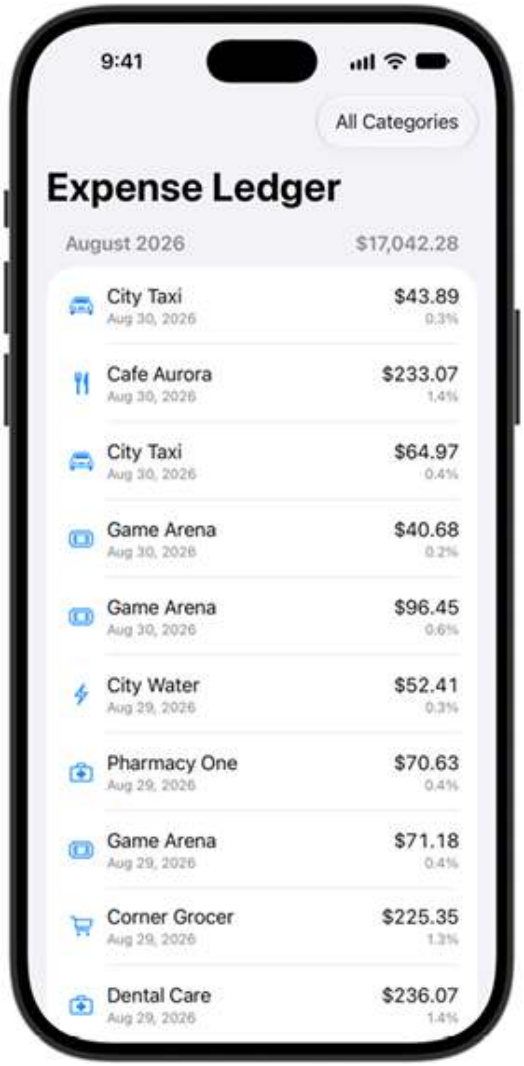


Figure 9.6: The expense ledger grouped by month, with the category picker in the toolbar

How it works…

The SwiftUI, Time Profiler, and signpost tracks answer three different questions, and the recording only made sense once we read them together. The SwiftUI instrument shows which updates ran long and what triggered them: its update groups gather every `body` evaluation that a single event caused, the long-update lanes flag the ones that threatened a frame, and selecting an update shows the state change behind it. The Time Profiler shows where the processor spent its time: it interrupts the

app about a thousand times a second, records the call stack of every running thread, and adds the samples up per function, so a function that appears in many samples is one that ran for a long time. Our signposts measure the operations whose boundaries we chose. The attribution in step 8 is the subtlety worth remembering. SwiftUI runs the closure of a `ForEach` while it builds the list's content, so any work inside that closure, such as our call to `expenses(in:)`, is charged to the `List` update rather than to the row whose data it computes. When a container shows a long update but no view of ours does, the code to look at is the code that produces the container's children, and the Time Profiler names it.

The first draft was slow for a reason that is easy to write and hard to see. Each row asked for every expense of its month, and that request filtered the full three thousand expenses with a calendar comparison, so displaying twenty rows scanned sixty thousand dates, and doing it again for every row that scrolled into view multiplied the cost by the number of rows in the month. The formatters added a smaller but steadier tax, because `DateFormatter` and `NumberFormatter` are expensive to create and were created three times per row per evaluation. The fix did not make any single scan faster; it stopped repeating it. `regroup()` filters, groups, and formats once, when the data actually changes, and stores the result in `LedgerRow` values holding identifiers and display strings, grouped into `LedgerMonth` values. The views no longer perform grouping or formatting, and because `LedgerRow` is `Equatable`, SwiftUI can compare the old and new row values and skip the `body` of every unchanged row. The rule of thumb is: the view layer should never repeat work the data layer could have done once. We cover the `@Observable` store and the `didSet` regrouping pattern in the *Observation Framework* section of *Chapter 5*.

Signposts cost almost nothing at runtime and are the ones I add first. `OSSignposter` takes a subsystem, which is conventionally the app's bundle identifier, and a category that groups related intervals, and

`beginInterval(_:)` hands back an `OSSignpostIntervalState` that `endInterval(_:_:)` uses to close that interval. Our intervals never overlap, so the default exclusive signpost ID is enough; intervals with the same name that can overlap each other need their own `OSSignpostID` from `makeSignpostID()`, or Instruments cannot tell which end belongs to which. `emitEvent(_:)` marks a single instant instead of a span, which is useful for events such as a network response arriving. When nothing is recording, the calls compile down to a check against a flag, so intervals can stay in shipping code; if we want them gone from a release build, `OSSignposter.disabled` is a signposter that does nothing. The intervals show up wherever the unified logging system is read: in the os_signpost instrument, in the Logging template, and in the `xctrace` command-line tool that exports Instruments recordings as XML, which is how the numbers in this recipe were extracted.

The Time Profiler's absolute numbers depend on where the app runs, which is why we prefer a device for representative measurements. The Profile action builds the Release configuration by default, because a Debug build disables the optimizer and adds checks that would show up in the profile as work the user never pays for. The simulator runs our code on the Mac's processor, with its cache sizes and clock speed, and the Hitches instrument is unavailable for that target, so hitches are not measurable there at all. What survives on the simulator is the shape of the profile: `expenses(in:)` still dominates the inverted call tree in a simulator recording, only with different absolute numbers.

The Hitches instrument in the SwiftUI template measures what the user actually feels: a hitch is a frame that arrived late, and the instrument reports how many there were and how long each one lasted. Our second recording still contains hitches during a fast flick, but the Time Profiler shows no code of ours inside them; they are the list preparing cells and laying out text, which is the framework's work, and the place to look next

only if those hitches matter. The Core Animation instrument goes one layer deeper, into rendering, and is the tool to add when a profile shows too little CPU time to explain a stutter, because offscreen rendering or blending is the likely cause. When smooth scrolling is a requirement rather than a hope, `XCTHitchMetric` measures the same hitches from a UI test like the ones we write in the *Automated UI Tests* section, so a regression fails the build instead of reaching users.



New in Xcode 27. The recordings behind this recipe came from an iPhone plugged into the Mac. Once the app ships, the Organizer window in Xcode 27 asks the same questions of real users' devices. Its Insights overview lists the metrics that regressed most across the user base, with the diagnostic reports behind them; the Storage pane tracks the app's footprint from release to release; and a Hitches metric reports the hitch rate during scrolling, transitions, and other continuous motion, drawing its goal as a dashed line when ours exceeds it. It is the same signal the Hitches instrume nt records locally, without the cable.

See also

- `OSSignposter` documentation: https://x.swiftuicookbook.com/yyklc
- *Understanding and improving SwiftUI performance* article: https://x.swiftuicookbook.com/ciz9u
- *Improving your app's performance* article: https://x.swiftuicookbook.com/r0p99
- *Analyzing the performance of your shipping app* article: https://x.swiftuicookbook.com/yawf4

- Optimize SwiftUI performance with Instruments (WWDC 2025): https://x.swiftuicookbook.com/hj3p2
- Optimize CPU performance with Instruments (WWDC 2025): https://x.swiftuicookbook.com/a0auj

Automated UI Tests

Every project I have shipped reached a point where checking the app by hand after each change stopped being reliable. The steps are the same every time: tap this, type that, confirm the label changed. That sameness is exactly what makes them easy to skip when a deadline is close, and that is how a regression reaches the App Store. A UI test performs those same taps for us, reads what is on screen, and fails when the result is not what we specified. Unlike us, it performs the fortieth check as carefully as the first.

In this recipe, we build a one-view app that asks for a country of origin, shows the choice in large type, and counts the taps, and then we test it the way a user would exercise it. We mark the views with `.accessibilityIdentifier()`, write four tests in a `XCTestCase` subclass that launches the app through an `XCUIApplication` proxy, find the buttons and texts as `XCUIElement` values, tap them, and assert on their labels with `XCTAssertTrue` and `XCTAssertEqual`. Along the way we run a single test, read its transcript in the console, run the whole class, let Xcode record an interaction for us, and replay a failing run in the Automation Explorer.

Apple now offers two testing frameworks, and the project template uses both. Swift Testing, with its `@Test` attribute and `#expect` macro, is the default for unit tests, while UI tests still run on XCTest, because driving an app from the outside needs the `XCUIAutomation` machinery that only XCTest provides. This recipe is about the second kind; the *How it works…* section explai ns how the two targets divide the work.

Getting ready

Create a new SwiftUI iOS app named **MyCountry** . In the options dialog, set **Testing System** to **Swift Testing with XCTest UI Tests** , as shown in Figure 9.7, so Xcode adds the two test targets this recipe relies on.

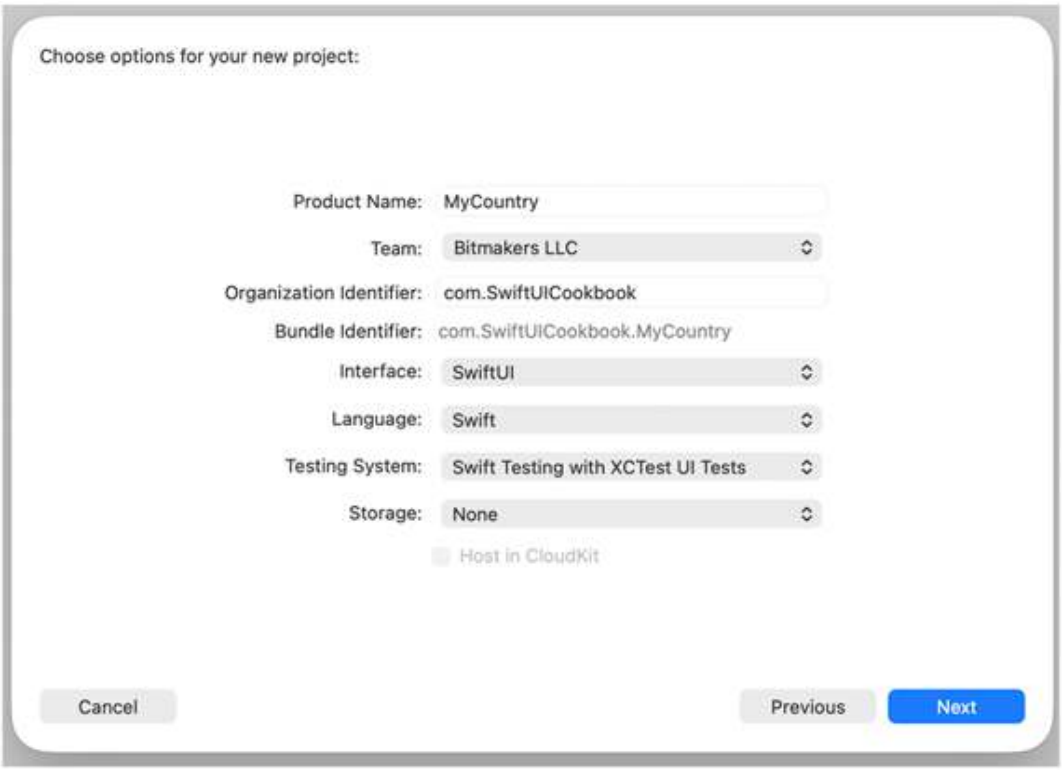


Figure 9.7: The new project options with the Testing System se t to Swift Testing with XCTest UI Tests

How to do it...

We write the app first and the tests second, because a UI test can only be as good as the app's accessibility tree, and it is easier to shape that tree while the view is small. Then each test adds one kind of check: presence, initial state, an action, and a reset. These are the steps:

1. Open `ContentView.swift` and add the list of countries and the two pieces of state, the selected country and the number of taps, right after the struct declaration:

```
static let countries = ["USA", "Spain", "France", "Italy"]
@State private var selectedCountry = Self.countries[0]
@State private var counter = 0
```

2. Replace the template `body` with the app's three `Text` views: the question at the top, the selected country in the middle, and the tap counter at the bottom. The buttons come next, so for now the layout is only text and spacers:

```
var body: some View {
  VStack {
    Text("What is your country of origin?")
      .font(.headline)
    Spacer()
    Text(selectedCountry)
      .font(.largeTitle)
      .bold()
    Spacer()
    HStack {
      Text("Number of taps: \(counter)")
      Spacer()
    }
  }
  .padding()
}
```

3. Below the question, add a row of buttons, one per country, that select the country and count the tap. The selected country is drawn in red and the others in blue, both in the prominent glass style, so the current choice is visible without reading the large text:

```
Text("What is your country of origin?")
  .font(.headline)
HStack(spacing: 12) {
  ForEach(Self.countries, id: \.self) { country in
    Button(country) {
      selectedCountry = country
      counter += 1
    }
```

```
      .buttonStyle(.glassProminent)
      .tint(selectedCountry == country ? .red : .blue)
  }
}
Spacer()
```

4. Finish the view with a **Reset** button at the end of the bottom `HStack`, which returns the app to its initial state, the first country and zero taps:

```
HStack {
  Text("Number of taps: \(counter)")
  Spacer()
  Button("Reset") {
    selectedCountry = Self.countries[0]
    counter = 0
  }
  .buttonStyle(.glass)
}
```

5. Make sure the preview canvas is in live mode and try the app. It starts with **USA** selected and zero taps, the initial state our tests will assert on. Tapping a country button highlights that button in red, updates the large text, and increments the counter, and **Reset** restores the initial state. Figure 9.8 shows the app after two taps.



Figure 9.8: The MyCountry app after tapping Spain twice

6. Now we prepare the view for testing. A UI test does not see our SwiftUI code; it sees the accessibility tree that iOS builds from it, the same tree VoiceOver reads. Every element in that tree can carry an identifier, a string of our choosing that never appears on screen, and a query that finds an element by identifier keeps finding it when its label is reworded or translated. Add an identifier to each of the three `Text` views with the `.accessibilityIdentifier()` modifier:

```
Text("What is your country of origin?")
  .font(.headline)
```

```
  .accessibilityIdentifier("Question")
// ... the HStack with the country buttons and the Spacer
Text(selectedCountry)
  .font(.largeTitle)
  .bold()
  .accessibilityIdentifier("SelectedCountry")
Spacer()
HStack {
  Text("Number of taps: \(counter)")
    .accessibilityIdentifier("NumberOfTaps")
  // ... the Spacer and the Reset button
}
```

7. Add identifiers to the five buttons as well. The country buttons take the first two letters of their name in uppercase, `US`, `SP`, `FR`, and `IT`, deliberately different from the labels so the tests show the difference between finding an element by identifier and reading its label. The **Reset** button simply uses `Reset`:

```
Button(country) {
  selectedCountry = country
  counter += 1
}
.buttonStyle(.glassProminent)
.tint(selectedCountry == country ? .red : .blue)
.accessibilityIdentifier(String(country.prefix(2)).uppercased())
// ... the selected country text and the tap counter
Button("Reset") {
  selectedCountry = Self.countries[0]
  counter = 0
}
.buttonStyle(.glass)
.accessibilityIdentifier("Reset")
```

8. The app is ready, so let's turn to the tests. The Project navigator shows two folders next to the `MyCountry` folder: `MyCountryTests` holds the unit tests, which we come back to in *How it works…*, and `MyCountryUITests` holds the UI tests. Open `MyCountryUITests.swift` in the

second folder and replace everything below the header comment with a class that owns an `XCUIApplication` proxy, the object that launches, monitors, and terminates the app under test. The `setUpWithError()` method runs before each test; it tells XCTest to stop a test at its first failure and launches a fresh copy of the app:

```
import XCTest
final class MyCountryUITests: XCTestCase {
  let app = XCUIApplication()
  override func setUpWithError() throws {
    continueAfterFailure = false
    app.launch()
  }
}
```

9. The first test checks that the five buttons exist and carry the labels we expect. A method whose name starts with `test` is a test, and Xcode marks it with a diamond in the gutter. Each test follows the same three beats: arrange the app in a known state, act on it, and assert on the result. Here the arranging is the launch in `setUpWithError()`, there is no action, and the assertions walk a dictionary from identifier to expected label. Add the method after `setUpWithError()`:

```
func testButtonLabels() throws {
  let expectedLabels = ["US": "USA", "SP": "Spain", "FR": "Franc
                        "IT": "Italy", "Reset": "Reset"]
  for (identifier, expectedLabel) in expectedLabels {
    let button = app.buttons[identifier]
    XCTAssertTrue(button.exists, "Button \(identifier) not prese
    XCTAssertEqual(button.label, expectedLabel)
  }
}
```

`app.buttons[identifier]` is a query for the button with that identifier, and `exists` resolves it. The first assertion is there so that a missing

button fails with our message, rather than with a confusing comparison against an empty label.

10. Run this one test. With an iPhone simulator selected as the run destination, hover over the diamond next to `testButtonLabels()` until it turns into a play button, and click it. Xcode builds the app and a separate test runner, runs them in Device Hub, and we can watch the app open and close again. The console records every step the test took, with a timestamp relative to the start of the test:



If the test runs but no app appears in Device Hub, the usual reason is parallel testing. Xcode runs UI tests on clones of the selected simulator rather than on the simulator itself, and the Xcode 27 release notes list those clones as not visible in Device Hub, a known issue whose workaround is to turn parallel testing off. Choose **Product** > **Scheme** > **Edit Scheme** , select the **Test** action, and under **Test Plans** click the arrow button next to **MyCountry (Autocreated)** , which opens the test plan editor. In its **Tests** tab, the `MyCountryUITests` target reads **Execute in parallel** beneath its name; click its **Options…** button and set **Parallelization** to **Disabled** .

Because the plan is autocreated, Xcode then asks whether to save it to disk. **Don't Save** keeps the change in memory for the current session and leaves the project without a test plan file, which is how the companion project ships, while **Save** writes a test plan file into the project so the setting persists. Either way, from then on the tests run on the simulator we chose, in plain view.

```
Test Suite 'MyCountryUITests' started at 2026-09-12 19:08:36.293.
Test Case '-[MyCountryUITests.MyCountryUITests testButtonLabels]'
  started.
    t =     0.00s Start Test at 2026-09-12 19:08:36.296
    t =     0.23s Set Up
    t =     0.23s     Open com.SwiftUICookbook.MyCountry
    t =     0.23s         Launch com.SwiftUICookbook.MyCountry
    t =     1.30s             Setting up automation session
    t =     3.76s             Wait for com.SwiftUICookbook.MyCountry
                              to idle
    t =     5.24s Checking existence of `"IT" Button`
    t =     5.31s Find the "IT" Button
    t =     5.31s Checking existence of `"Reset" Button`
    t =     5.32s Find the "Reset" Button
    t =     5.32s Checking existence of `"FR" Button`
    t =     5.32s Find the "FR" Button
    t =     5.33s Checking existence of `"US" Button`
    t =     5.33s Find the "US" Button
    t =     5.34s Checking existence of `"SP" Button`
    t =     5.34s Find the "SP" Button
    t =     5.35s Tear Down
Test Case '-[MyCountryUITests.MyCountryUITests testButtonLabels]'
  passed (5.361 seconds).
Test Suite 'MyCountryUITests' passed at 2026-09-12 19:08:41.658.
   Executed 1 test, with 0 failures (0 unexpected) in 5.361 (5.363)
   seconds
```

Most of the five seconds go to launching the app and waiting for it to settle; the five lookups together take a tenth of a second. The buttons are checked in an order that has nothing to do with the view, because a Swift dictionary has no defined order, which is harmless here but worth remembering when a test's steps depend on each other.

11. The second test does the same for the three `Text` views, confirming that the app starts with the question, **USA** , and zero taps. Texts are found through `staticTexts` instead of `buttons` ; everything else is the same pattern:

```
func testTextLabels() throws {
  let expectedLabels = ["Question": "What is your country of ori
                        "SelectedCountry": "USA",
                        "NumberOfTaps": "Number of taps: 0"]
  for (identifier, expectedLabel) in expectedLabels {
    let text = app.staticTexts[identifier]
    XCTAssertTrue(text.exists, "Text \(identifier) not present")
    XCTAssertEqual(text.label, expectedLabel)
  }
}
```

12. With the initial state covered, we can act on the app. The third test taps each country button in turn and, after each tap, checks that the large text shows that button's label and that the counter advanced by one. The test keeps its own count, so it knows what the app should display without reading the previous value from the screen:

```
func testCountryButtonAction() throws {
  var counter = 0
  for identifier in ["SP", "FR", "IT", "US"] {
    let button = app.buttons[identifier]
    button.tap()
    counter += 1
    XCTAssertEqual(app.staticTexts["SelectedCountry"].label, but
    XCTAssertEqual(app.staticTexts["NumberOfTaps"].label,
                   "Number of taps: \(counter)")
  }
}
```

13. The last test exercises **Reset** . It taps **Spain** twice, confirms the app is where two taps should leave it, taps **Reset** , and confirms the app is back at **USA** with zero taps. The intermediate assertions confirm that the country button changed the state before we test **Reset** ; without them, the test could pass if both actions did nothing and the app stayed at its initial state:

```
func testResetButtonAction() throws {
  let spain = app.buttons["SP"]
  spain.tap()
  spain.tap()
  XCTAssertEqual(app.staticTexts["SelectedCountry"].label, "Spai
  XCTAssertEqual(app.staticTexts["NumberOfTaps"].label, "Number
  app.buttons["Reset"].tap()
  XCTAssertEqual(app.staticTexts["SelectedCountry"].label, "USA'
  XCTAssertEqual(app.staticTexts["NumberOfTaps"].label, "Number
}
```

14. Run the four tests together by clicking the diamond next to the class name, or press *Cmd + U* to run every test target in the scheme, which also runs the placeholder test in `MyCountryTests`. The app launches four times, once per test, because `setUpWithError()` starts each one from a clean state, and each diamond turns green as its test passes. Choose the Report navigator, the last icon in the navigator bar or *Cmd + 9* , and select the test run to see the same result as a report: every test, its duration, and the list of activities it performed.
15. Writing the queries by hand is fine for five buttons, but Xcode can also write them for us. Add a test method with nothing inside it to the class, such as `func testRecording() throws {}`, and press *Return* between the braces so the cursor sits on an empty line inside the method: a small red record button appears in the gutter on that line, with the tooltip **Start recording** . Click it to launch the app. If the click does nothing, right-click the empty line and choose **Start recording** instead. As we tap **Spain** and **Reset** in Device Hub, Xcode inserts the matching `tap()` calls into the method, using our identifiers because we set them in steps 6 and 7. Stop the recording: Xcode has inserted the interaction calls, and we still need to write the assertions that check the expected result, because recording does not know what we meant to check. I use recording to discover the query for an element and then write the assertions myself. Delete the method afterward, or

keep it and add assertions; the companion project ships the four tests described earlier.

16. Let's introduce an incorrect expectation to inspect a failed test. In `testResetButtonAction()`, change the first expected counter label from `Number of taps: 2` to `Number of taps: 3` and run the test again. Its diamond turns red, and the report for the test now ends its activity list with the failed assertion, **XCTAssertEqual failed: ("Number of taps: 2") is not equal to ("Number of taps: 3")**, and the file and line it came from.
17. To watch the failure, open the Report navigator with *Cmd + 9*, select the latest test run, and double-click `testResetButtonAction()` in its list of tests. The test's report opens on its **Activities** tab, with the activity list on the left and the Automation Explorer on the right. The Automation Explorer holds a video of the whole run in the simulator: press its play button to replay it, and follow the timeline at the bottom, which marks the failure in red. By default, Xcode keeps this video only for a failed run, which is why we broke the test on purpose. Figure 9.9 shows the end of the run, where the test failed: **Spain** selected and two taps, which is what the app should show; our expectation was wrong, and the video makes that obvious without rerunning anything. Change the `3` back to `2` before moving on.

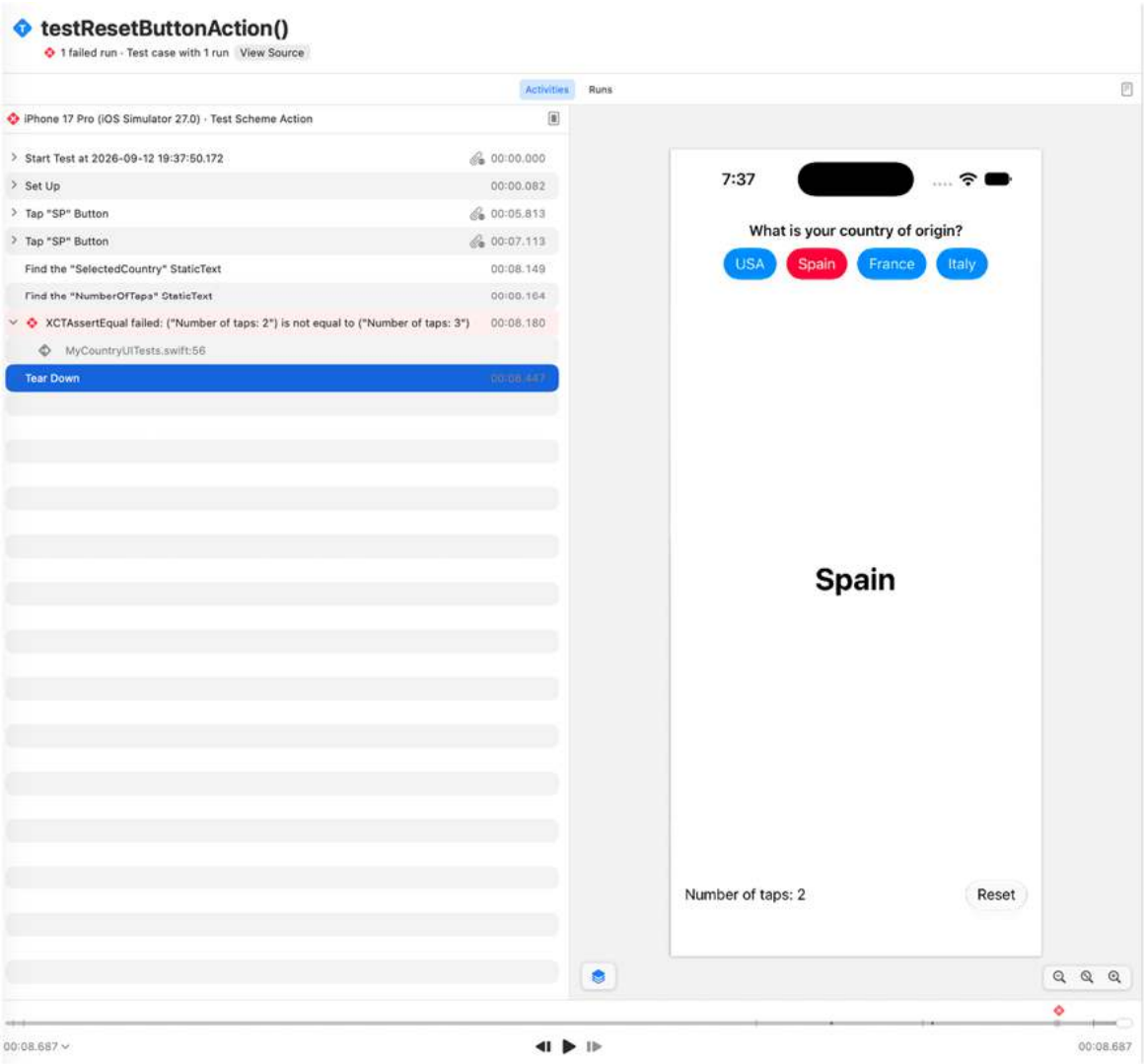


Figure 9.9: The Automation Explorer replaying the failed run of testResetButtonAction

18. The tests so far ran in the simulator. A UI test on a physical iPhone is the closest thing to a user trying the app, so run the suite once more with a connected iPhone chosen as the run destination in Xcode's toolbar, and watch the tests drive the app on the device. While the tests run, the iPhone shows **Automation Running** over the app, with a reminder that holding both volume buttons stops the session. The test code does not change; only the destination does. For the complete project, including the test target, see the book's GitHub repository.



New in Xcode 27. Xcode 27 shows the running app in Device Hub, the app that replaced the standalone Simulator app and the Devices and Simulators window, as described in the *Live Previews with #Preview* section of *Chapter 3*. To open it before a run, choose **Xcode** > **Open Developer Tool** > **Device Hub** . A connected iPhone appears in its sidebar next to the simulators, and the **View Screen** button on the iPhone's page mirrors the phone's screen on the Mac. With the screen showing, a UI test running on the device can be followed there like any other run, without picking up the phone.

How it works...

The tests never touched our SwiftUI code. They talked to the app through its accessibility tree, the structure iOS derives from the view hierarchy for VoiceOver and other assistive technologies, and everything a test can find or read is what that tree exposes. That is why the identifiers came first: `app.buttons["SP"]` asks for a button whose accessibility identifier is `SP`, while `button.label` reads the text a user sees, which is also what VoiceOver would speak. Keeping the identifier separate from the label lets our queries find the same element when its wording changes, such as when **Spain** is translated for another locale, provided the identifier itself stays fixed. Ours are computed from the strings in `countries`, so a localized version of this app would have to give each button its own fixed identifier rather than one derived from the translated name. The `NumberOfTaps` identifier likewise stays constant while its label changes with every tap, although an assertion on that label still has to spell out the exact text it

expects, as ours do. The same tree is what the *Accessibility APIs* section of *Chapter 6* shapes with labels, hints, and traits, so the work we do for VoiceOver and the work we do for testing reinforce each other.

The `XCUIApplication` proxy is the test's only handle on the app, and it lives in a different process. Xcode builds a test runner app, `MyCountryUITests-Runner`, that launches our app, sends it synthesized taps, and queries its accessibility tree over an automation session; the five seconds our first test took were mostly that session being set up. A query such as `app.staticTexts["SelectedCountry"]` is not evaluated when we write it, but when we inspect or interact with the resulting `XCUIElement`: `exists` checks for a match, `label` reads its text, and `tap()` sends a tap. We set `continueAfterFailure = false` because later checks often depend on earlier actions succeeding; if a tap fails, stopping immediately avoids a run of further failures caused by the app never reaching the expected state, and keeps the report readable.

The other folder, `MyCountryTests`, is where tests of our own types belong, and Xcode set it up with Swift Testing rather than XCTest: a test there is a function marked `@Test` that checks its results with `#expect`, and `@testable import MyCountry` gives it access to the app's internal types. Those tests instantiate our code directly and run in milliseconds, without the app launch and automation session that cost our first UI test five seconds, but they cannot tell whether the view shows the right thing, which is exactly what our four tests check. A good suite uses both: many fast unit tests for the model and the logic, and a smaller number of UI tests for the flows a user would notice breaking. Every one of them runs with *Cmd + U* and reports into the same Report navigator.

See also

- `XCUIAutomation` documentation: https://x.swiftuicookbook.com/a89s5
- `XCUIApplication` documentation: https://x.swiftuicookbook.com/ekkuc
- `accessibilityIdentifier(_:)` documentation: https://x.swiftuicookbook.com/sivj9
- *Testing your apps in Xcode* article: https://x.swiftuicookbook.com/nvcja
- Swift Testing documentation: https://x.swiftuicookbook.com/ll0rq
- Record, replay, and review: UI automation with Xcode (WWDC 2025): https://x.swiftuicookbook.com/q0z10

Part 3

Advanced APIs

In the last part of the book, you will move beyond stacks and lists to the APIs that make an app stand out. You will build precise layouts with grids, tables, and custom layout containers, model navigation as data across stacks, tabs, sheets, and deep links, and bring interfaces to life with custom drawing, animations, and Liquid Glass effects. From there, you will visualize data with Swift Charts, from basic marks to interactive and 3D charts, embed web content and edit rich text, and finish by adding Apple Intelligence to your apps with the Foundation Models and Image Playground frameworks.

This part of the book includes the following chapters:

- `Chapter 10`, *Grids and Layout*
- `Chapter 11`, *Navigation, Sheets, and Deep Linking*
- `Chapter 12`, *Drawing, Animations, and Visual Effects*
- `Chapter 13`, *Swift Charts*
- `Chapter 14`, *Web Content and Rich Text*
- `Chapter 15`, *On-Device Intelligence in SwiftUI*

10

Grids and Layout

Layout is the part of SwiftUI I find myself explaining most often, because the easy cases are so easy that the hard ones catch people off guard. A `VStack` and an `HStack` carry most of a screen without complaint, and then someone needs a table whose columns line up down the page, a row of tags that wraps to the next line, or views arranged around a circle, and the convenient containers quietly run out. In UIKit, those moments meant Auto Layout constraints or a `UICollectionView` with a custom layout, and I usually spent more time describing the geometry than the content. SwiftUI keeps the easy things easy, and when we outgrow the built-in containers it gives us precise tools, and even lets us write our own.

We begin with a medal standings board rendered by a single `Grid`, using `GridRow` for each line and `.gridCellColumns()`, `.gridColumnAlignment()`, `.gridCellAnchor()`, and `.gridCellUnsizedAxes()` to make cells span columns, align, and stay out of the way. We then line up a set of medal bars with `.alignmentGuide()` and a custom `AlignmentID`. Next, we build a gradebook around `Table`: sortable columns backed by `KeyPathComparator`, row selection with a `.contextMenu()`, editable score cells wired through `TextField` and `@FocusState`, and columns the user can show or hide through `TableColumnCustomization` persisted with `@SceneStorage`. Along the way we lift the column definitions into a

typed `@TableColumnBuilder`, and the whole table collapses to a compact layout when space runs short.

Finally, we leave the built-in containers behind and write our own, conforming to the `Layout` protocol by answering `sizeThatFits()` and `placeSubviews()`, measuring each child through its `LayoutSubview` proxy and respecting its `ViewSpacing`, and working within the `ProposedViewSize` we are handed. We then switch between two custom layouts with `AnyLayout` so the same views animate from one arrangement into the next.

By the end of this chapter, you should be comfortable reaching past `VStack` and `HStack` when a screen needs real columns, an editable data table, or an arrangement no stock container provides, and confident writing a custom layout container from scratch when the framework does not ship the one you need.

In this chapter, we will cover the following recipes:

- Grid and GridRow
- Table with Editable Cells
- Custom Layouts

Technical requirements

The code in this chapter is based on Xcode 27.0 and iOS 27.0. You can download and install the latest version of Xcode from the App Store. All the code examples for this chapter can be found on GitHub at `https://x.swiftuicookbook.com/ch10`.

Grid and GridRow

Some screens are fundamentally tabular: rows of values that need to line up into clean columns, like a scoreboard or a comparison chart. Building that by hand with a set of `HStack`s rarely holds together, because each row measures its own widths independently and the columns drift out of alignment. In UIKit, I would reach for a `UICollectionView` with a compositional layout to get dependable columns. SwiftUI gives us `Grid`, which measures every column to its widest cell and every row to its tallest, so cells line up across rows with no manual sizing at all.

In this recipe, we build a medal standings board for a fictional multi-sport event. A single `Grid` renders the whole card:

- A title that spans every column,
- A header,
- One row per country with gold, silver, and bronze counts, and a running total, and
- A summary row at the bottom.

We use `GridRow` to define each row, `.gridCellColumns()` to make a cell span several columns, `.gridColumnAlignment()` to align an entire column, `.gridCellAnchor()` to reposition a single cell within its span, and `.gridCellUnsizedAxes()` so a full-width `Divider` does not stretch the grid. We finish with a second card that lines up a row of medal bars using `.alignmentGuide()` and a custom alignment, a technique that works in any stack, not only inside a grid .

Getting ready

Create a new SwiftUI iOS app named **MedalStanding s** .

How to do it...

Our view has two cards stacked vertically: the standings grid and a medal-totals chart. We will start with the data model, then build the grid from its core outward, rows first, then the header, title, dividers, and summary, verifying each addition in the preview canvas. These are the steps:

1. Create a new Swift file named `Country.swift` for our data layer. The `Country` struct keeps just what the board needs: the three medal counts the grid lines up into columns, a computed `total` so the last column can never disagree with the medals, and a `displayName` that joins the flag emoji and name into the single label the first column shows. The `sampleData` array seeds six countries, already ordered by gold medals:

```
import Foundation
struct Country: Identifiable {
  let id = UUID()
  let flag: String
  let name: String
  let gold: Int
  let silver: Int
  let bronze: Int
  var total: Int { gold + silver + bronze }
  var displayName: String { "\(flag) \(name)" }
}
extension Country {
  static let sampleData: [Country] = [
    Country(flag: "US", name: "USA", gold: 12, silver: 9, b
    Country(flag: "ES", name: "Spain", gold: 8, silver: 10,
    Country(flag: "JP", name: "Japan", gold: 7, silver: 5, l
    Country(flag: "GB", name: "UK", gold: 6, silver: 7, bro
    Country(flag: "IN", name: "India", gold: 5, silver: 4,
    Country(flag: "AU", name: "Australia", gold: 3, silver:
  ]
}
```

2. Open `ContentView.swift` and replace the `ContentView` struct with the version below. It keeps the countries in a `let` property, since the data never changes here and does not need `@State`, and lays out one `GridRow` per country inside a `Grid`. Each row has five cells: the country's `displayName`, the three medal counts, and the total. We fix the name label to its natural width with `.fixedSize()` so a long name never wraps to a second line as the grid tightens, and the four numbers share a single `.monospacedDigit()` through a `Group`, which the row still counts as four cells. The whole grid lives in a `standingsCard` property with a rounded background:

```
struct ContentView: View {
  private let countries = Country.sampleData
  var body: some View {
    NavigationStack {
      ScrollView {
        VStack(spacing: 24) {
          standingsCard
        }
        .padding()
      }
      .navigationTitle("Medal Standings")
    }
  }
  private var standingsCard: some View {
    Grid(alignment: .center, horizontalSpacing: 12, vertica
      ForEach(countries) { country in
        GridRow {
          Text(country.displayName)
            .fixedSize(horizontal: true, vertical: false)
          Group {
            Text("\(country.gold)")
            Text("\(country.silver)")
            Text("\(country.bronze)")
            Text("\(country.total)")
```

```
                .fontWeight(.semibold)
            }
            .monospacedDigit()
          }
        }
      }
      .padding()
      .background(
        RoundedRectangle(cornerRadius: 16)
          .fill(Color(.secondarySystemBackground))
      )
    }
  }
```



A `Text` inside a grid cell wraps a long value onto a second line rather than widening its column, because the grid gives each flexible cell only as much width as the layout wants to hand it. Applying `.fixedSize(horizontal: true, vertical: false)` tells the label to insist on its natural single-line width, so every country name stays on one line as the grid tightens. To see the difference, temporarily give one country a much longer name, such as `"United States of America"`, and remove the modifier: the long name wraps onto several lines and even short names split their flag from their name. Put the modifier back and every name returns to one line, but the grid now grows past the screen edge, so `.fixedSize()` is the right tool only when the content is known to fit.

3. Add a header row above the country rows. The `.gridColumnAlignment()` modifier sets the horizontal alignment for an entire column, and we apply it to just one cell in that column. We align the country column leading and the total column trailing; the three medal columns keep the grid's default center alignment:

```
GridRow {
  Text("Country")
    .gridColumnAlignment(.leading)
  Text("🥇")
  Text("🥈")
  Text("🥉")
  Text("Total")
    .gridColumnAlignment(.trailing)
}
.font(.caption.bold())
.foregroundStyle(.secondary)
// ... the ForEach country rows
```

4. Give the card a title that runs across the full width. A cell marked with `.gridCellColumns(5)` spans all five columns, turning a single `Text` into a banner above the header:

```
GridRow {
  Text("🏅 Final Standings")
    .font(.headline)
    .gridCellColumns(5)
}
// ... header row and country rows
```

5. Separate the title, header, rows, and summary with horizontal rules. A `Divider` placed directly in a `Grid` becomes a full-width row of its own, and because a divider is flexible it would stretch

the grid to fill all the width it can get, so we mark each one with `.gridCellUnsizedAxes(.horizontal)`, which lets the divider span the width without influencing column sizing:

```
Grid(alignment: .center, horizontalSpacing: 12, verticalSpa
  GridRow {
    // ... title
  }
  Divider()
    .gridCellUnsizedAxes(.horizontal)
  GridRow {
    // ... header
  }
  .font(.caption.bold())
  .foregroundStyle(.secondary)
  Divider()
    .gridCellUnsizedAxes(.horizontal)
  ForEach(countries) { country in
    // ... country row
  }
}
```

6. Add a summary row that reports the grand total. First, add a `grandTotal` property that sums every country's total. Then, after a final divider, add a `GridRow` whose label spans the first four columns. We anchor that label to the trailing edge of its span with `.gridCellAnchor(.trailing)`, which overrides the leading alignment the column would otherwise impose, so the label sits right beside the total:

```
private var grandTotal: Int {
  countries.reduce(0) { $0 + $1.total }
}
// ... at the end of the Grid, after the country rows
Divider()
```

```
    .gridCellUnsizedAxes(.horizontal)
  GridRow {
    Text("Total medals awarded")
      .font(.caption)
      .foregroundStyle(.secondary)
      .gridCellColumns(4)
      .gridCellAnchor(.trailing)
    Text("\(grandTotal)")
      .font(.callout.bold())
      .monospacedDigit()
  }
```

7. The grid now carries a lot of inline cell content. Extract each row's cells into its own member so `standingsCard` reads as a clean list of rows. A `GridRow` expects several sibling views, so each helper is marked `@ViewBuilder` and returns the views directly. Writing `GridRow { headerContent }` expands them back into individual columns exactly as if they were inline. The country row depends on its `country`, so it becomes a function, while the header and footer are computed properties:

```
private var standingsCard: some View {
  Grid(alignment: .center, horizontalSpacing: 12, verticalS
    GridRow {
      Text("🏅 Final Standings")
        .font(.headline)
        .gridCellColumns(5)
    }
    Divider()
      .gridCellUnsizedAxes(.horizontal)
    GridRow {
      headerContent
    }
    .font(.caption.bold())
    .foregroundStyle(.secondary)
    Divider()
      .gridCellUnsizedAxes(.horizontal)
    ForEach(countries) { country in
```

```
        GridRow {
          countryContent(country)
        }
      }
      Divider()
        .gridCellUnsizedAxes(.horizontal)
      GridRow {
        footerContent
      }
    }
    .padding()
    .background(
      RoundedRectangle(cornerRadius: 16)
        .fill(Color(.secondarySystemBackground))
    )
  }
  @ViewBuilder
  private var headerContent: some View {
    Text("Country")
      .gridColumnAlignment(.leading)
    Text("🥇")
    Text("🥈")
    Text("🥉")
    Text("Total")
      .gridColumnAlignment(.trailing)
  }
  @ViewBuilder
  private func countryContent(_ country: Country) -> some Vie
    Text(country.displayName)
      .fixedSize(horizontal: true, vertical: false)
    Group {
      Text("\(country.gold)")
      Text("\(country.silver)")
      Text("\(country.bronze)")
      Text("\(country.total)")
        .fontWeight(.semibold)
    }
    .monospacedDigit()
  }
  @ViewBuilder
  private var footerContent: some View {
    Text("Total medals awarded")
      .font(.caption)
```

```
      .foregroundStyle(.secondary)
      .gridCellColumns(4)
      .gridCellAnchor(.trailing)
    Text("\(grandTotal)")
      .font(.callout.bold())
      .monospacedDigit()
}
```

8. Now, we build the second card, which draws a bar for each country. We want every bar to start on the same vertical line even though the country labels have different widths, so we define a custom horizontal alignment. Add this extension below the `ContentView` struct. An `AlignmentID` describes where a view sits on the guide by default, here at its own leading edge:

```
extension HorizontalAlignment {
  private enum BarStart: AlignmentID {
    static func defaultValue(in context: ViewDimensions) ->
      context[.leading]
    }
  }
  static let barStart = HorizontalAlignment(BarStart.self)
}
```

9. Add the totals card and wire it into `body`. Each row places a label, a `Capsule` whose width encodes the total, and a trailing count in an `HStack`. The inner `VStack` aligns its rows on our custom `.barStart` guide, and each bar reports its leading edge for that guide with `.alignmentGuide(.barStart)`, so every bar begins on the same line. Add `totalsCard` to the `VStack` in `body`, below `standingsCard`, then add the two members after the grid helpers:

```
struct ContentView: View {
  // ... the countries and grandTotal properties
  var body: some View {
    NavigationStack {
      ScrollView {
        VStack(spacing: 24) {
          standingsCard
          totalsCard
        }
        .padding()
      }
      .navigationTitle("Medal Standings")
    }
  }
  // ... standingsCard, headerContent, countryContent, and
  private var totalsCard: some View {
    VStack(alignment: .leading, spacing: 12) {
      Text("Medal totals")
        .font(.headline)
      VStack(alignment: .barStart, spacing: 10) {
        ForEach(countries) { country in
          totalsRow(for: country)
        }
      }
    }
    .frame(maxWidth: .infinity, alignment: .leading)
    .padding()
    .background(
      RoundedRectangle(cornerRadius: 16)
        .fill(Color(.secondarySystemBackground))
    )
  }
  private func totalsRow(for country: Country) -> some View
    HStack(spacing: 8) {
      Text(country.displayName)
      Capsule()
        .fill(.orange.gradient)
        .frame(width: CGFloat(country.total) * 5, height: 1
        .alignmentGuide(.barStart) { dimension in
          dimension[.leading]
        }
      Text("\(country.total)")
        .font(.caption.monospacedDigit())
```

```
            .foregroundStyle(.secondary)
        }
    }
}
```

10. Open the preview canvas to see the finished view. Every column in the standings grid lines up, the summary row totals all the medals, and in the second card the bars share a common starting line regardless of label width. For the complete code, see the book's GitHub repository.

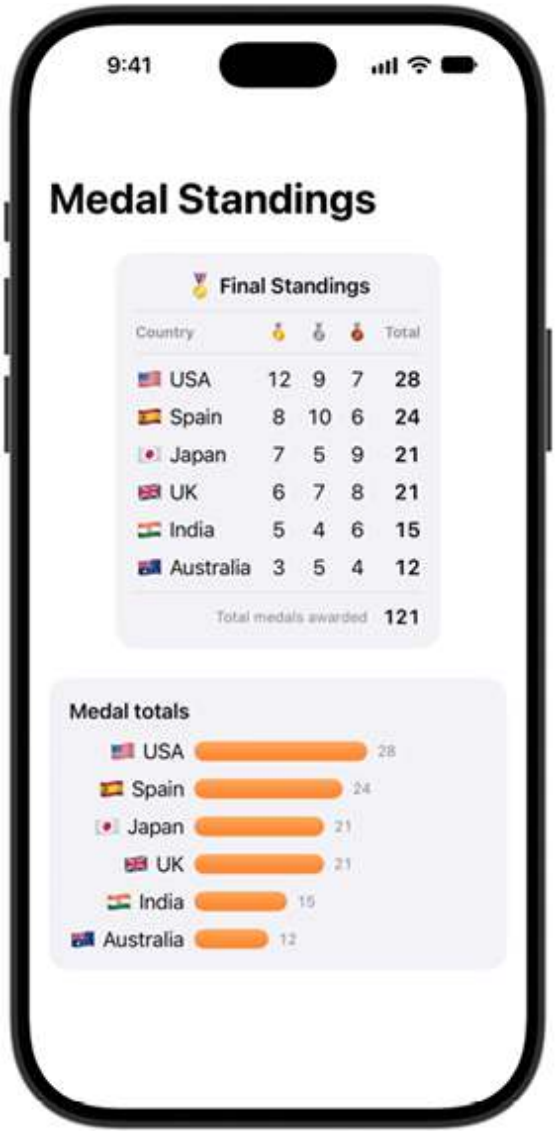


Figure 10.1: The standings grid with aligned columns and the medal-totals bars sharing a common starting line

How it works …

Our standings card lines up because `Grid` treats its content as a two-dimensional matrix: each `GridRow` is one row, every view inside it becomes a cell in the next column, and the grid measures each column to its widest cell and each row to its tallest. That is why the medal counts and totals line up across all six countries without us

specifying a single width. A `Group` inside a row is transparent to that count, which is why the four numbers we grouped under one `.monospacedDigit()` still land in four cells. A view placed directly in the `Grid` but outside any `GridRow`, like our `Divider`s, spans the full width. Unlike `LazyVGrid` and `LazyHGrid`, which render cells on demand for large scrolling collections, `Grid` renders every cell immediately, which makes it the right tool for a small, static table like this scoreboard.

Alignment in a grid works as a hierarchy, from broad to specific. The `Grid(alignment:)` initializer sets a default for every cell, which we left at `.center`. `.gridColumnAlignment()` overrides that default for an entire column; we applied it once to the country header (leading) and once to the total header (trailing), and every cell below inherits its column's alignment. Finally, `.gridCellAnchor()` overrides alignment for a single cell. Our summary label uses spanning and anchoring together: `.gridCellColumns(4)` makes it stretch across the first four columns, and `.gridCellAnchor(.trailing)` pins it to the trailing edge of that span, so it sits beside the grand total instead of following the leading alignment the first column would otherwise impose.

Both sizing modifiers in this recipe are ways of opting a view out of the grid's measuring negotiation. `.gridCellUnsizedAxes(.horizontal)` tells the grid to draw the `Divider` full width but ignore it when sizing columns, and `.fixedSize()` (explained in the note in step 2) lets a country label insist on its natural width instead of wrapping. Without them, the most flexible views in the card, not its content, would decide its shape.

The second card shows that alignment guides are not limited to grids. An alignment guide reports where a view sits along a given alignment, and a container lines its children up on that guide. We

defined a custom `HorizontalAlignment`, `barStart`, backed by an `AlignmentID` whose `defaultValue` places a view at its own leading edge. The inner `VStack(alignment: .barStart)` aligns its rows on that guide, and each `Capsule` sets `.alignmentGuide(.barStart)` to its leading edge. Every bar therefore begins on the same vertical line, while the country labels, which vary in width, fan out to its left. Custom alignments like this are the tool to reach for whenever views in different rows or containers need to line up on something other than their edges or centers.

See also

- `G rid` documentation: https://x.swiftuicookbook.com/j3tn6
- `GridRow` documentation: https://x.swiftuicookbook.com/q7i6e
- `gridCellColumns(_:)` documentation: https://x.swiftuicookbook.com/fr3hu
- `gridColumnAlignment(_:)` documentation: https://x.swiftuicookbook.com/bv2hp
- `gridCellAnchor(_:)` documentation: https://x.swiftuicookbook.com/h7qw6
- `alignmentGuide(_:computeValue:)` documentation: https://x.swiftuicookbook.com/mj73k

Table with Editable Cells

A gradebook, a price sheet, a leaderboard: some data is better presented as a table, with real columns that need to line up, sort, and

stay aligned as rows are edited. In UIKit, building that on iOS meant a `UICollectionView` with a compositional layout, or a stack of labels wrestled into columns by hand. SwiftUI's `Table` gives us honest columns with sortable headers, row selection, and editable cells, the kind of interface that used to live only on the Mac and now runs on iPad.

In this recipe, we build a table where a teacher edits each student's three subject scores and watches the average and letter grade recompute on every keystroke. We start with a read-only table and layer on capability: `KeyPathComparator` sorting wired through the `sortOrder` binding, row selection with a `.contextMenu()` for honor-roll and delete actions, editable score cells backed by `TextField` and `@FocusState`, and column show-and-hide driven by `TableColumnCustomization` persisted with `@SceneStorage`. Along the way we lift the column definitions into a typed `@TableColumnBuilder` property to keep `body` readable, and we make the whole table collapse to a compact single-column layout when the width gets tight.

`Table` lays out multiple columns only on iPad, in a regular-width window. On iPhone it shows a single column in every orientation, even on a Max-sized model in landscape, and so does a narrow iPad window, so we design for both environments: the full multi-column layout on a wide iPad and a rich report-card row everywhere else. That split matters more than it used to, because iPadOS runs apps in freely resized windows, so the same screen has to work across the whole width range.

Getting r eady

Create a new SwiftUI iOS app named **Gradebook** . Because `Table` shows its columns only in a regular-width environment, run the recipe on a simulated iPad in Device Hub, or use a resizable iPad preview so you can see both the wide and compact layouts.

How to d o it...

Our app is one `Table` that gains features step by step. We start with a read-only multi-column table on iPad, add sorting and selection, make the score cells editable, then teach the table to collapse gracefully when the width gets tight. The data model and the two small cell views each live in their own file. These are the steps:

1. For our data layer, create a new Swift file named `Student.swift` in a `Models` folder. The three subject scores are the editable heart of the app, so they are `var` properties; `isHonorRoll` is a flag the teacher toggles by hand, independent of the scores. Everything else is computed: `total` and `average` derive from the scores, `formattedAverage` rounds the average for display, and `letterGrade` turns the average into a letter with a set of grading rules. The `sampleData` array seeds eight students:

```
import Foundation
struct Student: Identifiable {
  let id = UUID()
  var name: String
  var math: Int
  var science: Int
  var english: Int
  var isHonorRoll: Bool
  var total: Int { math + science + english }
```

```
  var average: Double { Double(total) / 3.0 }
  var formattedAverage: String { String(format: "%.1f", ave
  var letterGrade: String {
    switch average {
    case 90...: "A"
    case 80..<90: "B"
    case 70..<80: "C"
    case 60..<70: "D"
    default: "F"
    }
  }
}
extension Student {
  static let sampleData: [Student] = [
    Student(name: "Alice Johnson", math: 95, science: 92,
      english: 88, isHonorRoll: true),
    Student(name: "Bob Smith", math: 82, science: 78,
      english: 91, isHonorRoll: false),
    Student(name: "Charlie Brown", math: 88, science: 94,
      english: 85, isHonorRoll: true),
    Student(name: "Diana Lee", math: 76, science: 83,
      english: 79, isHonorRoll: false),
    Student(name: "Eve Wilson", math: 98, science: 96,
      english: 94, isHonorRoll: true),
    Student(name: "Frank Garcia", math: 72, science: 68,
      english: 75, isHonorRoll: false),
    Student(name: "Grace Kim", math: 91, science: 89,
      english: 93, isHonorRoll: true),
    Student(name: "Henry Chen", math: 85, science: 87,
      english: 80, isHonorRoll: false)
  ]
}
```

2. Open `ContentView.swift` and replace the `ContentView` struct with a first, read-only table. We keep the students in a `@State` array and declare six columns. `Name` and `Grade` use the string-based `TableColumn(_:value:)` convenience initializer, which renders the property directly, while the four numeric columns take a content closure so we can format the number ourselves. Wrapping the

table in a `NavigationStack` gives us the title bar we will hang a toolbar on later:

```
struct ContentView: View {
  @State private var students = Student.sampleData
  var body: some View {
    NavigationStack {
      Table(students) {
        TableColumn("Name", value: \.name)
        TableColumn("Grade", value: \.letterGrade)
        TableColumn("Math") { student in
          Text("\(student.math)")
        }
        TableColumn("Science") { student in
          Text("\(student.science)")
        }
        TableColumn("English") { student in
          Text("\(student.english)")
        }
        TableColumn("Average") { student in
          Text(student.formattedAverage)
        }
      }
      .navigationTitle("Gradebook")
    }
  }
}
```

3. Turn the column headers into sort controls. Sorting needs two things: a `sortOrder` array of `KeyPathComparator` values bound to the `Table`, and a `value:` key path on every column we want to sort by. The two string columns already pass one, and now that the table carries a sort order, the four numeric columns can pass theirs as well. We sort the data ourselves in `.onChange(of: sortOrder)`, because the table reports the requested order but

does not reorder the rows for us. We also give the name column a minimum width, which we will explain in the note below:

```
@State private var students = Student.sampleData
@State private var sortOrder = [KeyPathComparator(\Student.
var body: some View {
  NavigationStack {
    Table(students, sortOrder: $sortOrder) {
      TableColumn("Name", value: \.name)
        .width(min: 200)
      TableColumn("Grade", value: \.letterGrade)
      TableColumn("Math", value: \.math) { student in
        Text("\(student.math)")
      }
      TableColumn("Science", value: \.science) { student in
        Text("\(student.science)")
      }
      TableColumn("English", value: \.english) { student in
        Text("\(student.english)")
      }
      TableColumn("Average", value: \.average) { student in
        Text(student.formattedAverage)
      }
    }
    .onChange(of: sortOrder) { _, newOrder in
      students.sort(using: newOrder)
    }
    .navigationTitle("Gradebook")
  }
}
```

Open the preview canvas on an iPad and tap a header to sort by that column. Tap **Math** to rank the class by math score, then tap **Name** to go back to alphabetical order.

> The first column behaves differently from the rest. Its header only becomes an active sort



control once the column is wide enough to display it, so a name column left at its natural width can look like sorting is broken. Pinning it with `.width(min: 200)` keeps the header, and its sort arrow, tappable.

4. Add row selection and a context menu. A `Set<Student.ID>`, declared with the other `@State` properties, tracks the selection, and passing it to the `Table` turns on row selection. The name column keeps its `value:` key path for sorting but gains a trailing content closure, the same shape the four numeric columns already use, so it can show a star beside an honor-roll student's name. The reserved star keeps every name aligned, using `.clear` when the student is not on the honor roll. We attach `.contextMenu(forSelectionType:)` to the `Table` itself with two actions, and add the two methods those actions call after `body`: one flips the honor-roll flag on each selected student, and the other removes the selected rows and clears them from the selection. The closure receives an empty set when the long press lands on the header or below the last row, so we return no items in that case:

```
struct ContentView: View {
  // ... the students and sortOrder properties
  @State private var selection = Set<Student.ID>()
  var body: some View {
    NavigationStack {
      Table(students, selection: $selection, sortOrder: $so
        TableColumn("Name", value: \.name) { student in
          HStack(spacing: 6) {
            Image(systemName: "star.fill")
              .font(.caption)
```

```
                .foregroundStyle(student.isHonorRoll ? .yello
              Text(student.name)
            }
          }
          .width(min: 200)
          // ... the other five columns
        }
        .contextMenu(forSelectionType: Student.ID.self) { ids
          if !ids.isEmpty {
            Button("Toggle Honor Roll", systemImage: "star")
              toggleHonorRoll(for: ids)
            }
            Button("Delete", systemImage: "trash", role: .des
              delete(ids)
            }
          }
        }
        // ... onChange(of: sortOrder) and navigationTitle
      }
    }
    private func toggleHonorRoll(for ids: Set<Student.ID>) {
      for index in students.indices where ids.contains(studer
        students[index].isHonorRoll.toggle()
      }
    }
    private func delete(_ ids: Set<Student.ID>) {
      students.removeAll { ids.contains($0.id) }
      selection.subtract(ids)
    }
}
```

5. Give the user toolbar buttons for the two most common actions: deleting the current selection and adding a new student. The `.toolbar` attaches to the `Table` after `.navigationTitle()`. The trash button stays disabled while nothing is selected, and the plus button appends a student with placeholder scores that are ready to edit, through an `addStudent()` method that joins the other two after `body`:

```
struct ContentView: View {
  // ... the properties
  var body: some View {
    NavigationStack {
      Table(students, selection: $selection, sortOrder: $sc
        // ... the six columns
      }
      // ... contextMenu and onChange(of: sortOrder)
      .navigationTitle("Gradebook")
      .toolbar {
        ToolbarItem(placement: .topBarLeading) {
          Button("Delete", systemImage: "trash", role: .des
            delete(selection)
          }
          .disabled(selection.isEmpty)
        }
        ToolbarItem(placement: .topBarTrailing) {
          Button("Add Student", systemImage: "plus") {
            addStudent()
          }
        }
      }
    }
  }
  // ... toggleHonorRoll and delete
  private func addStudent() {
    students.append(Student(name: "New Student", math: 80,
      science: 80, english: 80, isHonorRoll: false))
  }
}
```

The trash button and the context menu both act on the current selection. Tap a student's name to select the row, then delete it with the trash button or long-press for the context menu.

6. Now, we start making the scores editable, which takes the next four steps. Editing needs to know exactly which field in which row is active, so before the `ContentView` struct we add a

compound focus type: a `ScoreField` enum for the three subjects and a `ScoreFocus` value that pairs a student's ID with a field. Both are marked `nonisolated` so they can conform to `Hashable` without being tied to the main actor:

```
nonisolated enum ScoreField: Hashable {
  case math, science, english
}
nonisolated struct ScoreFocus: Hashable {
  let student: Student.ID
  let field: ScoreField
}
```

7. Create a new Swift file named `ScoreCellView.swift` in a `Views` folder for the editable score field. It binds a `TextField` to the score with the `.number` format, which parses the typed text back into an `Int`, and colors the value by how high it is. The `alignment` parameter defaults to trailing but lets each caller choose, so the same view works in the wide table and in the compact layout:

```
import SwiftUI
struct ScoreCellView: View {
  @Binding var score: Int
  var focus: FocusState<ScoreFocus?>.Binding
  let field: ScoreFocus
  var alignment: TextAlignment = .trailing
  private var color: Color {
    switch score {
    case 90...: .green
    case 80..<90: .blue
    case 70..<80: .orange
    default: .red
    }
  }
  var body: some View {
    TextField("Score", value: $score, format: .number)
```

```
        .font(.body.monospacedDigit())
        .foregroundStyle(color)
        .multilineTextAlignment(alignment)
        .frame(minWidth: 40)
        .focused(focus, equals: field)
    }
}
#Preview {
    @Previewable @State var score = 88
    @Previewable @FocusState var focus: ScoreFocus?
    ScoreCellView(
      score: $score,
      focus: $focus,
      field: ScoreFocus(student: UUID(), field: .math)
    )
    .frame(width: 120)
}
```

8. Back in `ContentView.swift`, make the three score columns editable. Add a `@FocusState` property for the active field with the other properties above `body`, and switch each score column's content to a `ScoreCellView`, produced by a `scoreCell(for:_:)` helper so the binding-and-focus wiring is written once and reused for all three subjects. That helper and a `binding(for:)` helper, which returns a student's live binding in the array, go after `body`. We track focus with `@FocusState` the same way we did for keyboard navigation in the *Forms* section of *Chapter 4*, but keyed to a compound value so it can pinpoint one field in one row:

```
struct ContentView: View {
    // ... the students, sortOrder, and selection properties
    @FocusState private var focusedField: ScoreFocus?
    var body: some View {
        NavigationStack {
            Table(students, selection: $selection, sortOrder: $sc
                // ... the Name and Grade columns
                TableColumn("Math", value: \.math) { scoreCell(for:
```

```
        TableColumn("Science", value: \.science) {
          scoreCell(for: $0, .science)
        }
        TableColumn("English", value: \.english) {
          scoreCell(for: $0, .english)
        }
        // ... the Average column
      }
      // ... contextMenu, onChange(of: sortOrder), navigati
    }
  }
  private func scoreCell(
    for student: Student,
    _ field: ScoreField
  ) -> some View {
    let score = switch field {
    case .math: binding(for: student).math
    case .science: binding(for: student).science
    case .english: binding(for: student).english
    }
    return ScoreCellView(
      score: score,
      focus: $focusedField,
      field: ScoreFocus(student: student.id, field: field),
      alignment: .center
    )
  }
  private func binding(for student: Student) -> Binding<Stu
    guard let index = students.firstIndex(where: { $0.id ==
    else { return .constant(student) }
    return $students[index]
  }
  // ... toggleHonorRoll, delete, and addStudent
}
```

9. Selecting a different row should commit whatever edit is in progress. Add an `.onChange(of: selection)` that clears the focus, which ends editing in the field that had it and writes the value back to the model:

```
// ... after the .onChange(of: sortOrder) modifier on the T
.onChange(of: selection) { _, _ in
  focusedField = nil
}
```

Edit any score in the preview canvas and watch the average and letter grade for that row update as you type.

10. The scores now sit centered in their columns, but their headers, and the grade and average cells, still hug the leading edge. Add `.alignment(.center)` to the five value columns, which aligns each column's header together with its cells, and give the `Grade` and `Average` columns cell helpers of their own, `gradeCell(for:)` and `averageCell(for:)`, added after `body`, so the two read-only values get their styling in one place:

```
struct ContentView: View {
  // ... the properties
  var body: some View {
    NavigationStack {
      Table(students, selection: $selection, sortOrder: $sc
        // ... the Name column
        TableColumn("Grade", value: \.letterGrade) { grade(
          .alignment(.center)
        TableColumn("Math", value: \.math) { scoreCell(for:
          .alignment(.center)
        TableColumn("Science", value: \.science) {
          scoreCell(for: $0, .science)
        }
        .alignment(.center)
        TableColumn("English", value: \.english) {
          scoreCell(for: $0, .english)
        }
        .alignment(.center)
        TableColumn("Average", value: \.average) { average(
          .alignment(.center)
      }
```

```
      // ... contextMenu, onChange handlers, navigationTitl
    }
  }
  private func gradeCell(for student: Student) -> some View
    Text(student.letterGrade)
      .fontWeight(.semibold)
  }
  // ... scoreCell
  private func averageCell(for student: Student) -> some Vi
    Text(student.formattedAverage)
      .font(.body.monospacedDigit())
      .foregroundStyle(.secondary)
  }
  // ... binding, toggleHonorRoll, delete, and addStudent
}
```

11. The `body` now holds the whole table: six columns, sorting, selection, the context menu, and the toolbar. Lift the column definitions into a dedicated property so `body` stays readable. A property that returns several columns is annotated with `@TableColumnBuilder`, whose two generic parameters name the row type, `Student`, and the sort-comparator type, for which we add a `StudentComparator` typealias. Inside the builder we switch to fully qualified `\Student.` key paths, and we move the name cell into a `nameCell(for:)` helper alongside the others:

```
private typealias StudentComparator = KeyPathComparator<Stu
// ... @State properties and @FocusState
var body: some View {
  NavigationStack {
    Table(students, selection: $selection, sortOrder: $sort
      columns
    }
    // ... contextMenu, onChange handlers, navigationTitle,
  }
}
@TableColumnBuilder<Student, StudentComparator>
```

```
private var columns: some TableColumnContent<Student, Stude
  TableColumn("Name", value: \Student.name) { nameCell(for:
    .width(min: 200)
  TableColumn("Grade", value: \Student.letterGrade) { grade
    .alignment(.center)
  TableColumn("Math", value: \Student.math) { scoreCell(for
    .alignment(.center)
  TableColumn("Science", value: \Student.science) {
    scoreCell(for: $0, .science)
  }
  .alignment(.center)
  TableColumn("English", value: \Student.english) {
    scoreCell(for: $0, .english)
  }
  .alignment(.center)
  TableColumn("Average", value: \Student.average) { average
    .alignment(.center)
}
private func nameCell(for student: Student) -> some View {
  HStack(spacing: 6) {
    Image(systemName: "star.fill")
      .font(.caption)
      .foregroundStyle(student.isHonorRoll ? .yellow : .cle
    Text(student.name)
  }
}
```

12. Let the user hide and show columns. We add a `TableColumnCustomization` stored in `@SceneStorage`, which persists the choices per window, with the other stored properties above `body`, and pass it to the `Table`:

```
struct ContentView: View {
  // ... the typealias, students, sortOrder, and selection
  @SceneStorage("gradebookColumns")
  private var columnCustomization: TableColumnCustomizatio
  @FocusState private var focusedField: ScoreFocus?
  var body: some View {
    NavigationStack {
```

```
        Table(
          students,
          selection: $selection,
          sortOrder: $sortOrder,
          columnCustomization: $columnCustomization
        ) {
          columns
        }
        // ... contextMenu, onChange handlers, navigationTitl
      }
    }
    // ... the columns builder and the helpers
}
```

13. Now, describe how each column participates in that customization. On every column, `.customizationID()` gives a stable key, `.width()` sets a sensible size, `.defaultVisibility(.hidden)` starts the average column hidden, and `.disabledCustomizationBehavior(.all)` opts the name column out of all three customization behaviors (visibility, reordering, and resizing), so it can never be hidden. Replace the `columns` property with this version:

```
@TableColumnBuilder<Student, StudentComparator>
private var columns: some TableColumnContent<Student, Stude
  TableColumn("Name", value: \Student.name) { nameCell(for:
    .width(min: 200)
    .customizationID("name")
    .disabledCustomizationBehavior(.all)
  TableColumn("Grade", value: \Student.letterGrade) { grade
    .width(80)
    .customizationID("grade")
    .alignment(.center)
  TableColumn("Math", value: \Student.math) { scoreCell(for
    .width(max: 90)
    .customizationID("math")
    .alignment(.center)
```

```
    TableColumn("Science", value: \Student.science) {
      scoreCell(for: $0, .science)
    }
    .width(max: 90)
    .customizationID("science")
    .alignment(.center)
    TableColumn("English", value: \Student.english) {
      scoreCell(for: $0, .english)
    }
    .width(max: 90)
    .customizationID("english")
    .alignment(.center)
    TableColumn("Average", value: \Student.average) { average
      .width(max: 90)
      .customizationID("average")
      .defaultVisibility(.hidden)
      .alignment(.center)
}
```



The order of these modifiers matters: `.width()` must come first in each chain, before `.customizationID()` and `.alignment()`, or the compiler rejects the chain. Also, on the Mac a table shows a built-in header menu for hiding, showing, and reordering columns, but iPad offers no such menu, so at this point the customization state exists yet nothing lets the user change it. We add our own control next.

14. List the columns whose visibility the user may toggle. Add a small `GradebookColumn` type before the `ContentView` struct, then a `customizableColumns` array inside `ContentView` listing the five columns, each with whether it starts hidden:

```
struct GradebookColumn: Identifiable {
  let id: String
  let title: String
  let hiddenByDefault: Bool
}
// ... add this property to ContentView
private let customizableColumns: [GradebookColumn] = [
  GradebookColumn(id: "grade", title: "Grade", hiddenByDefa
  GradebookColumn(id: "math", title: "Math", hiddenByDefaul
  GradebookColumn(id: "science", title: "Science", hiddenBy
  GradebookColumn(id: "english", title: "English", hiddenBy
  GradebookColumn(id: "average", title: "Average", hiddenBy
]
```

15. Add a menu for column visibility that drives the customization state. It goes inside the existing `.toolbar` as a third `ToolbarItem`, between the delete and add items, and its toggles read and write `columnCustomization[visibility:]` through a `columnVisible(_:)` binding, added after `body`, which resolves the `.automatic` case against each column's default so the checkmarks match what is actually on screen:

```
struct ContentView: View {
  // ... the properties
  var body: some View {
    NavigationStack {
      // ... the Table with its contextMenu and onChange ha
      .navigationTitle("Gradebook")
      .toolbar {
        // ... the Delete item
        ToolbarItem(placement: .topBarTrailing) {
          Menu("Columns", systemImage: "slider.horizontal.3
            ForEach(customizableColumns) { column in
              Toggle(column.title, isOn: columnVisible(colu
            }
          }
        }
```

```
            // ... the Add Student item
          }
        }
      }
      // ... the columns builder and the cell helpers
      private func columnVisible(_ column: GradebookColumn) ->
        Binding(
          get: {
            switch columnCustomization[visibility: column.id] {
            case .visible: true
            case .hidden: false
            default: !column.hiddenByDefault
            }
          },
          set: {
            columnCustomization[visibility: column.id] = $0 ? .
          }
        )
      }
      // ... toggleHonorRoll, delete, and addStudent
    }
```

Tap the sliders icon in the toolbar, then toggle **Average** to show the hidden column, or hide **Science** to slim the table down. The toolbar shows the menu as its icon alone, so the `Columns` title we passed is never displayed.

16. Create a new Swift file named `CompactRowView.swift` in the `Views` folder for the compact layout. When the table collapses to a single column in a compact-width environment, this report card carries the whole row: the star and name on top, the three editable score lines on the left, and the average and a large letter grade on the right. The score lines reuse `ScoreCellView`, so editing works exactly as it does in the wide table:

```
import SwiftUI
struct CompactRowView: View {
  @Binding var student: Student
  var focus: FocusState<ScoreFocus?>.Binding
  var body: some View {
    HStack(spacing: 12) {
      Image(systemName: "star.fill")
        .font(.caption)
        .foregroundStyle(student.isHonorRoll ? .yellow : .c
      VStack(alignment: .leading, spacing: 6) {
        Text(student.name)
          .font(.headline)
        HStack(alignment: .center, spacing: 16) {
          VStack(alignment: .leading, spacing: 4) {
            scoreLine("Math", .math, $student.math)
            scoreLine("Sci", .science, $student.science)
            scoreLine("Eng", .english, $student.english)
          }
          Spacer()
          Text(student.formattedAverage)
            .font(.title3.monospacedDigit())
          Text(student.letterGrade)
            .font(.system(size: 50, weight: .bold))
            .foregroundStyle(.secondary)
            .frame(minWidth: 44)
        }
      }
    }
    .padding(.vertical, 4)
  }
  private func scoreLine(
    _ label: String,
    _ field: ScoreField,
    _ score: Binding<Int>
  ) -> some View {
    HStack(spacing: 6) {
      Text(label)
        .frame(width: 40, alignment: .leading)
      ScoreCellView(
        score: score,
        focus: focus,
        field: ScoreFocus(student: student.id, field: fielc
      )
```

```
    }
  }
}
#Preview {
  @Previewable @State var student = Student.sampleData[0]
  @Previewable @FocusState var focus: ScoreFocus?
  CompactRowView(student: $student, focus: $focus)
    .padding(.horizontal)
}
```

17. Make `ContentView` adapt to its environment. Read both size classes into properties with the other properties, and add an `isCompact` convenience beside them that is true when either one is compact: a compact width covers an iPhone in portrait and a narrow iPad window, and a compact height covers every iPhone in landscape, where even a Max-sized model shows a `Table` as a single column. Then replace the `nameCell(for:)` helper from step 11 with the version below. It gains the `@ViewBuilder` attribute so it can return either branch: when `isCompact` is true the name cell becomes the full-row `CompactRowView`, and otherwise it stays the star-and-name pair:

```
struct ContentView: View {
  // ... the existing properties
  @Environment(\.horizontalSizeClass) private var sizeClass
  @Environment(\.verticalSizeClass) private var verticalSiz
  private var isCompact: Bool {
    sizeClass == .compact || verticalSizeClass == .compact
  }
  // ... customizableColumns, body, and the columns builder
  @ViewBuilder
  private func nameCell(for student: Student) -> some View
    if isCompact {
      CompactRowView(student: binding(for: student), focus:
    } else {
      HStack(spacing: 6) {
```

```
          Image(systemName: "star.fill")
            .font(.caption)
            .foregroundStyle(student.isHonorRoll ? .yellow :
          Text(student.name)
        }
      }
    }
    // ... the other helpers
}
```

18. Finally, disable the column-visibility menu when the layout is compact, since hiding and showing columns has no effect while only one column is visible. Add `.disabled(isCompact)` to the menu's toolbar item:

```
Menu("Columns", systemImage: "slider.horizontal.3") {
  // ... the Toggle rows
}
.disabled(isCompact)
```

Switch the preview canvas to resizable mode, which we met in the *Trait Variations and Device Simulations* section of *Chapter 3*, and drag the iPad preview narrower. As the width crosses into the compact size class the table collapses to a single report-card column, and dragging it wider brings the columns back:

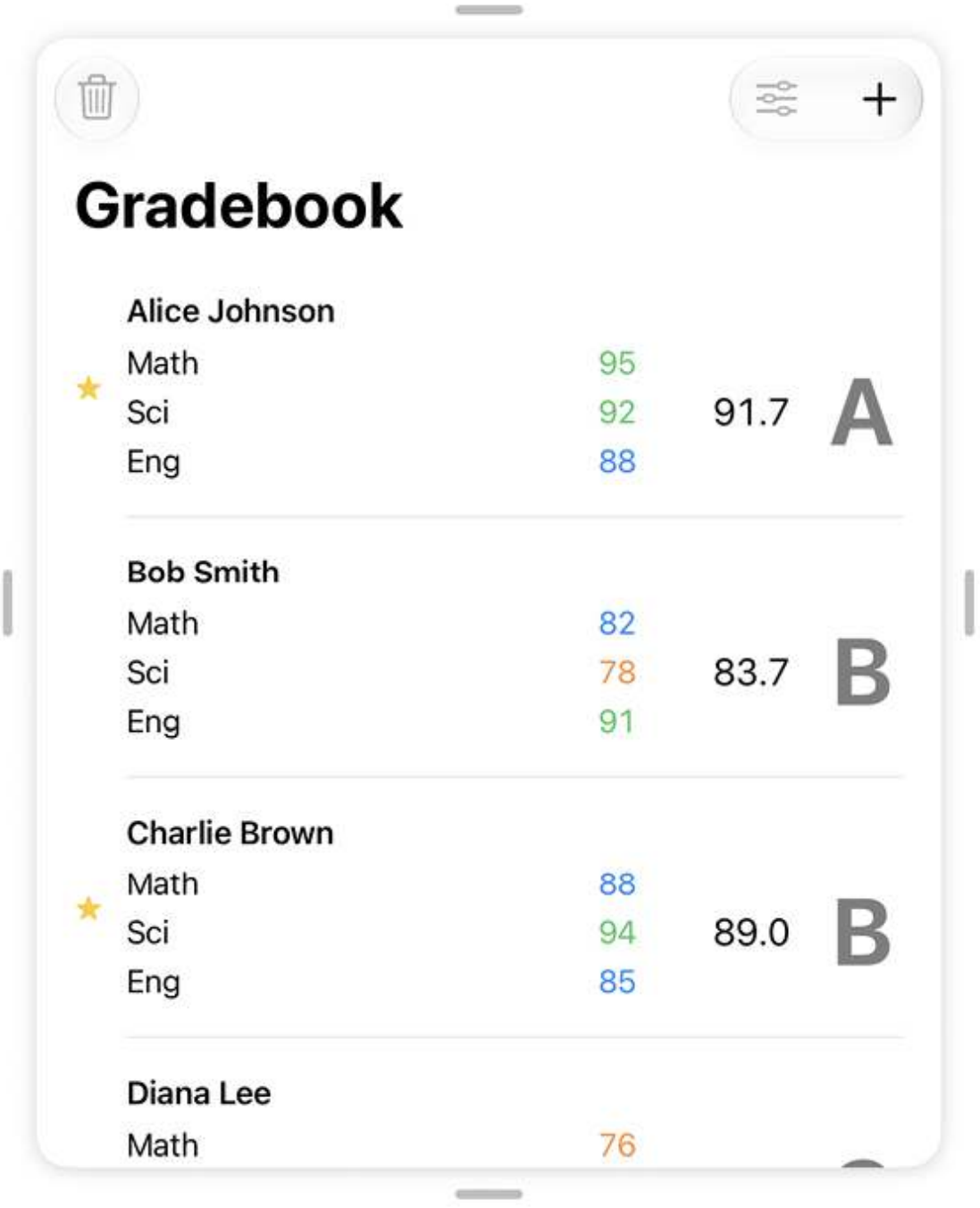


Figure 10.2: The Gradebook app at a compact width, collapsed to a single report-card column

19. We have finished the app. In the preview canvas, with the iPad preview at full width, put the wide layout through its paces: tap a header to sort, select a row and delete it with the trash button

or the context menu, edit a few scores and watch the average and grade follow, and use the sliders icon to hide and show columns. For the complete code, see the book's GitHub repository.

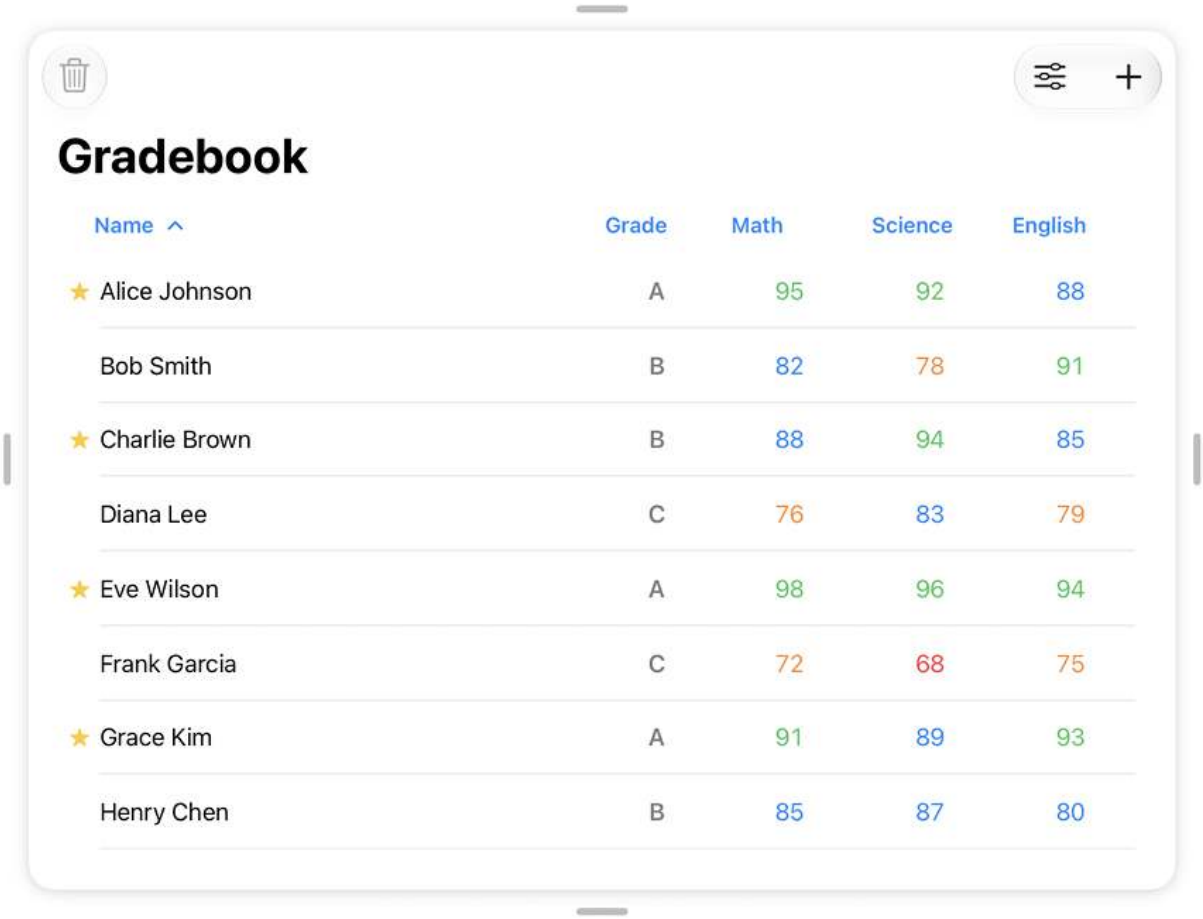


Figure 10.3: The Gradebook table on a wide iPad, with sortable columns, honor-roll stars, and color-coded editable sc ores

How it works…

Our `Table` lays its data out as a real two-dimensional grid: each `TableColumn` is a column, and `Table` renders one row per `Student`, lazily, the way `List` renders rows. The catch is that this multi-column layout only appears on iPad, in a regular-width window. On iPhone,

even a Max-sized model in landscape, and in a narrow iPad window, `Table` shows just the first column, which is why we read both size classes and branch: on a wide iPad the user sees every column, and everywhere else the first column has to carry the whole row on its own.

Sorting flows through the `sortOrder` binding, an array of `KeyPathComparator` values. Each column that passes a `value:` key path becomes tappable in the header, and tapping it rewrites `sortOrder`; the table does not reorder the data itself, so we do the actual sort in `.onChange(of: sortOrder)`. The binding and the key paths come as a package: a table with no `sortOrder` has no sort type at all, which is why our numeric columns could not pair a `value:` key path with a content closure until the binding existed. The `.width(min: 200)` floor on the name column exists for the reason the note in step 3 gives, a quirk that cost me time the first time I hit it: let the first column shrink too far and its header stops acting as a sort control, so sorting by name looks broken.

The editable cells are ordinary `TextField`s bound to the student's scores through `TextField(value:format:)` with the `.number` format, which parses the typed text back into an `Int`. Because each score is a live binding into the `students` array, every keystroke mutates the model, and since `average` and `letterGrade` are computed properties, the table re-renders those cells with fresh values immediately. The `binding(for:)` helper hands each cell a `Binding` straight into the array, a technique we explore in the *State and Binding Basics* section of *Chapter 5*. We track which field is being edited with a `@FocusState` keyed to a compound `ScoreFocus` value, a student ID plus a `ScoreField` case, so focus can identify one field in one row out of two dozen. We

first used `@FocusState` for keyboard navigation in the *Forms* section of *Chapter 4*; here the same tool commits an edit when the user selects a different row, because `.onChange(of: selection)` clears the focus.

Once the table carries six columns, sorting, selection, a context menu, and a toolbar, the `body` becomes hard to read. Moving the column definitions into a dedicated property, annotated with the `@TableColumnBuilder` result builder, restores that readability, and it is the idiomatic way to factor columns out of a table. The builder's two generic parameters, the `Student` row type and the `StudentComparator` sort type, give the property a concrete type, and the same property lets us route the three near-identical score columns through one `scoreCell(for:_:)` helper instead of repeating the binding-and-focus wiring three times. Column alignment is a column property too: `.alignment(.center)` centers a column's header together with its cells, which no modifier on the cell content alone could do for the header, while the score fields still center their typed text through `.multilineTextAlignment()`, because a text field fills its cell and decides on its own where the text sits.

Column customization has three moving parts. `.customizationID()` gives each column a stable key, `@SceneStorage` persists the user's choices across launches, and `.defaultVisibility(.hidden)` starts the average column hidden while `.disabledCustomizationBehavior(.all)` pins the name column so it can never be hidden or reordered. On the Mac a table exposes a built-in header menu for all of this; iPad does not (the note in step 13), which is why we drive the same `columnCustomization` state ourselves through a `Menu` in the toolbar. Its toggles read and write `columnCustomization[visibility:]`, and our

`columnVisible(_:)` binding resolves the `.automatic` case against each column's default so the checkmarks match what is on screen. The row menu comes from `.contextMenu(forSelectionType:)`, whose closure receives the IDs under the long press, and an empty set when the press lands on the header or below the last row, which is why the closure returns no items in that case. Both this `Menu` and the row `.contextMenu()` are covered in the *Menus and Disclosure Groups* section of *Chapter 4*.

When the width goes compact, the table hands the whole row to the first column, so `nameCell(for:)` swaps in `CompactRowView`, a report card that stacks the name, the three editable score lines, the average, and a large letter grade into one cell. We also disable the column-visibility menu there, since hiding and showing columns has no meaning when only one is visible. As the recipe intro noted, this is not only an iPhone concern: on iPadOS the same table can be dragged across that boundary within a single s ession.

See also

- `Table` documentation: https://x.swiftuicookbook.com/i6q29
- `TableColumn` documentation: https://x.swiftuicookbook.com/yucmr
- `TableColumnCustomization` documentation: https://x.swiftuicookbook.com/6v0wq
- `KeyPathComparator` documentation: https://x.swiftuicookbook.com/tkrhs

- `SceneStorage` documentation: https://x.swiftuicookbook.com/ej82j
- `FocusState` documentation: https://x.swiftuicookbook.com/nbtw1

Custom Layouts

Sooner or later a screen asks for an arrangement that none of the built-in containers can produce. The classic one is a row of tags: labels of different widths that should fill a line, then wrap to the next. An `HStack` will not wrap, it just keeps going and squeezes its children. A `LazyVGrid` wraps, but its columns are declared up front and every item takes its column's width, which is the wrong model for labels that are all different sizes. In UIKit this is the moment we would reach for a `UICollectionViewFlowLayout` and a delegate that sizes every item. SwiftUI hands us something smaller and more direct: the `Layout` protocol, where we answer the only two questions a container ever has to answer: how much space it needs, and where each of its children goes.

In this recipe, we build a screen of skill chips that wrap across as many lines as they need. We write a `FlowLayout` that conforms to `Layout`, implementing `sizeThatFits()` to report how much room the chips need and `placeSubviews()` to position each one, reading each chip's ideal size through its `LayoutSubview` proxy and taking the gaps from `ViewSpacing` rather than inventing them. Then, we write a second layout, a `RadialLayout` that arranges the same chips around a circle, and use `AnyLayout` to switch between the two so the chips animate

from one arrangement into the other instead of disappearing and coming back.

The two layouts are a good pair because they answer the sizing question in opposite ways: the flow layout measures its children and asks for exactly the space they need, while the radial layout takes whatever space it is offered and works within it. Seeing both is the fastest way to understand what `ProposedViewSize` is rea lly for.

Getting ready

Create a new SwiftUI iOS app named **Ski llCloud** .

How to do it...

We will build this the same way SwiftUI itself thinks about layout: first decide how big the container needs to be, then decide where each child goes. We start with the data and the chip, prove that no built-in container can arrange them, and only then write the layout that can. These are the steps:

1. For our data layer, create a new Swift file named `Skill.swift` in a `Models` folder. Each skill has a name and a color we use to tint its chip. Note the import: a plain Swift file starts with `import Foundation` , but we need `import SwiftUI` here because `Color` is a SwiftUI type:

```
import SwiftUI
struct Skill: Identifiable {
  let id = UUID()
  let name: String
  let color: Color
```

```
}
extension Skill {
  static let sampleData: [Skill] = [
    Skill(name: "SwiftUI", color: .blue),
    Skill(name: "Swift", color: .orange),
    Skill(name: "Xcode", color: .indigo),
    Skill(name: "SwiftData", color: .green),
    Skill(name: "Animation", color: .mint),
    Skill(name: "Testing", color: .red),
    Skill(name: "Charts", color: .purple),
    Skill(name: "Widgets", color: .brown)
  ]
}
```

2. Create a new Swift file named `SkillView.swift` in a `Views` folder for the chip itself. It is deliberately plain, a `Text` in a `Capsule`, because the interesting part of this recipe is where these chips end up, not what they look like. The chip fills with its color when selected and stays tinted when it is not:

```
import SwiftUI
struct SkillView: View {
  let skill: Skill
  let isSelected: Bool
  var body: some View {
    Text(skill.name)
      .font(.subheadline.weight(.medium))
      .padding(.horizontal, 14)
      .padding(.vertical, 8)
      .foregroundStyle(isSelected ? .white : skill.color)
      .background(
        Capsule()
          .fill(skill.color.opacity(isSelected ? 1 : 0.15))
      )
      .overlay(
        Capsule()
          .strokeBorder(skill.color.opacity(0.5), lineWidth
      )
  }
```

```
}
#Preview {
  VStack(spacing: 12) {
    SkillView(skill: Skill.sampleData[0], isSelected: false
    SkillView(skill: Skill.sampleData[0], isSelected: true)
  }
  .padding()
}
```

3. Now, let's see the problem for ourselves. Open `ContentView.swift` and replace the `ContentView` struct with an `HStack` of chips:

```
struct ContentView: View {
  private let skills = Skill.sampleData
  var body: some View {
    NavigationStack {
      HStack {
        ForEach(skills) { skill in
          SkillView(skill: skill, isSelected: false)
        }
      }
      .padding()
      .navigationTitle("SkillCloud")
    }
  }
}
```

   Look at the preview canvas. The eight chips are crammed into a single line, squeezed and clipped, because an `HStack` has exactly one row and will compress its children rather than move any of them to a second line. Wrapping is not a styling problem we can fix with a modifier; it is a layout the framework does not provide. So, we write it.

4. Create a new Swift file named `FlowLayout.swift` in the `Views` folder. A type conforms to `Layout` by answering two questions:

`sizeThatFits()` reports how much space the container needs, and `placeSubviews()` puts each child in position. Both of them need the same thing first, the chips grouped into rows, so both call a shared `arrange(_:width:)` helper, defined at the bottom of the file:

```
import SwiftUI
struct FlowLayout: Layout {
  var lineSpacing: CGFloat = 10
  func sizeThatFits(
    proposal: ProposedViewSize,
    subviews: Subviews,
    cache: inout ()
  ) -> CGSize {
    let maxWidth = proposal.replacingUnspecifiedDimensions(
    let rows = arrange(subviews, width: maxWidth)
    let width = rows.map(\.width).max() ?? 0
    let height = rows.reduce(0) { $0 + $1.height }
      + lineSpacing * CGFloat(max(rows.count - 1, 0))
    return CGSize(width: width, height: height)
  }
  func placeSubviews(
    in bounds: CGRect,
    proposal: ProposedViewSize,
    subviews: Subviews,
    cache: inout ()
  ) {
    let rows = arrange(subviews, width: bounds.width)
    var y = bounds.minY
    for row in rows {
      var x = bounds.minX
      for item in row.items {
        x += item.leadingGap
        subviews[item.index].place(
          at: CGPoint(x: x, y: y),
          anchor: .topLeading,
          proposal: ProposedViewSize(item.size)
        )
        x += item.size.width
      }
      y += row.height + lineSpacing
```

```
    }
  }
  private struct Item {
    let index: Int
    let size: CGSize
    let leadingGap: CGFloat
  }
  private struct Row {
    var items: [Item] = []
    var width: CGFloat = 0
    var height: CGFloat = 0
  }
  private func arrange(_ subviews: Subviews, width: CGFloat
    var rows: [Row] = []
    var row = Row()
    for index in subviews.indices {
      let size = subviews[index].sizeThatFits(.unspecified)
      var gap: CGFloat = 0
      if let last = row.items.last {
        let previous = subviews[last.index].spacing
        let current = subviews[index].spacing
        gap = previous.distance(to: current, along: .horizo
      }
      if !row.items.isEmpty, row.width + gap + size.width >
        rows.append(row)
        row = Row()
        gap = 0
      }
      row.items.append(Item(index: index, size: size, leadi
      row.width += gap + size.width
      row.height = max(row.height, size.height)
    }
    if !row.items.isEmpty {
      rows.append(row)
    }
    return rows
  }
}
```

The `arrange(_:width:)` helper does the real work: it walks the chips, asks each one for its natural size, and starts a new row

whenever the next chip would not fit; where those sizes and the gaps between chips come from is the subject of *How it works* . The row types and the helper are `private` because they are how the layout works, not part of how it is configured. The one stored property we leave non-private is `lineSpacing` , because it is the knob the caller turns in `FlowLayout(lineSpacing: 10)` , and Swift makes a struct's synthesized memberwise initializer private the moment any stored property is private, which would put that initializer out of reach of `ContentView` .

5. Back in `ContentView.swift` , swap the `HStack` for our new layout. A `Layout` is used exactly like any built-in container, we call it with a trailing closure of content:

```
FlowLayout(lineSpacing: 10) {
  ForEach(skills) { skill in
    SkillView(skill: skill, isSelected: false)
  }
}
.padding()
.navigationTitle("SkillCloud")
```

   Look at the preview canvas. The chips now fill a line and wrap onto the next, and none of them is squeezed. We just added a container that SwiftUI does not provide.

6. Make the chips selectable, so the screen behaves like a real skill picker rather than a static cloud. A `Set` of skill IDs, declared below `skills` , tracks the selection, tapping a chip adds it to or removes it from the set, and the `toggle(_:)` method that does so goes after `body` :

```
struct ContentView: View {
  private let skills = Skill.sampleData
  @State private var selection = Set<Skill.ID>()
  var body: some View {
    NavigationStack {
      FlowLayout(lineSpacing: 10) {
        ForEach(skills) { skill in
          SkillView(skill: skill, isSelected: selection.cor
            .onTapGesture {
              toggle(skill)
            }
        }
      }
      .padding()
      .navigationTitle("SkillCloud")
    }
  }
  private func toggle(_ skill: Skill) {
    withAnimation(.snappy) {
      if selection.contains(skill.id) {
        selection.remove(skill.id)
      } else {
        selection.insert(skill.id)
      }
    }
  }
}
```

7. Now, a second layout, to show that the same protocol can produce something completely different. Create a new Swift file named `RadialLayout.swift` in the `Views` folder. Where the flow layout measured its chips and asked for exactly the room they needed, this one accepts whatever space it is offered and arranges the chips on a circle inside it. The radius is inset by the widest and tallest chip so nothing spills outside the bounds, and it is clamped at zero so the layout survives even a tiny proposal.

The `animatableData` property is added because we will animate the rotation later, to provide a nicer visual transition:

```
import SwiftUI
struct RadialLayout: Layout {
  var rotation: Angle = .degrees(-90)
  var animatableData: Double {
    get { rotation.radians }
    set { rotation = .radians(newValue) }
  }
  func sizeThatFits(
    proposal: ProposedViewSize,
    subviews: Subviews,
    cache: inout ()
  ) -> CGSize {
    proposal.replacingUnspecifiedDimensions()
  }
  func placeSubviews(
    in bounds: CGRect,
    proposal: ProposedViewSize,
    subviews: Subviews,
    cache: inout ()
  ) {
    guard !subviews.isEmpty else { return }
    let sizes = subviews.map { $0.sizeThatFits(.unspecified
    let widest = sizes.map(\.width).max() ?? 0
    let tallest = sizes.map(\.height).max() ?? 0
    let radius = max(min(bounds.width - widest, bounds.heig
    let step = Angle.degrees(360 / Double(subviews.count))
    for (index, subview) in subviews.enumerated() {
      let angle = rotation + step * Double(index)
      subview.place(
        at: CGPoint(
          x: bounds.midX + radius * cos(angle.radians),
          y: bounds.midY + radius * sin(angle.radians)
        ),
        anchor: .center,
        proposal: ProposedViewSize(sizes[index])
      )
    }
  }
}
```

8. Wire both layouts into `ContentView` and let the user switch between them. The trick is `AnyLayout` : `FlowLayout` and `RadialLayout` are two unrelated types, and a single property cannot hold sometimes one and sometimes the other, so `AnyLayout` uses type erasure to hide either concrete type behind one shared type, the same idea as `AnyView` for views. Add the state, a `layout` property that picks one, and a `controls` section with the toggle:

```
private let skills = Skill.sampleData
@State private var selection = Set<Skill.ID>()
@State private var isRadial = false
@State private var rotation: Angle = .degrees(-90)
private var layout: AnyLayout {
  isRadial
    ? AnyLayout(RadialLayout(rotation: rotation))
    : AnyLayout(FlowLayout(lineSpacing: 10))
}
var body: some View {
  NavigationStack {
    VStack(spacing: 24) {
      layout {
        ForEach(skills) { skill in
          SkillView(skill: skill, isSelected: selection.cor
            .onTapGesture {
              toggle(skill)
            }
        }
      }
      .frame(maxWidth: .infinity, minHeight: isRadial ? 340
      Spacer()
      controls
    }
    .padding()
    .navigationTitle("SkillCloud")
  }
}
private var controls: some View {
```

```
    VStack(spacing: 16) {
      Toggle("Radial layout", isOn: $isRadial.animation(.spri
      Text("\(selection.count) selected")
        .font(.caption)
        .foregroundStyle(.secondary)
    }
  }
```

Flip the **Radial layout** toggle in the preview canvas. The chips travel from their rows onto the circle and back, because `AnyLayout` keeps them as the same views rather than tearing them down and building new ones.

9. Add a button that spins the circle. A `step` property, placed with the other computed properties above `body`, works out the angle between two neighboring chips, and a **Rotate** button in the `controls` stack, between the toggle and the count, advances the rotation by exactly one position. Rotating by one position rather than a fixed number of degrees keeps the arrangement predictable regardless of how many chips we have. This is where `animatableData` earns its place: because `RadialLayout` is animatable, SwiftUI interpolates the rotation and the chips glide around the circle:

```
struct ContentView: View {
  // ... the @State properties
  private var step: Angle {
    .degrees(360 / Double(max(skills.count, 1)))
  }
  // ... layout and body
  private var controls: some View {
    VStack(spacing: 16) {
      Toggle("Radial layout", isOn: $isRadial.animation(.sp
      Button("Rotate") {
        withAnimation(.spring(duration: 0.6)) {
```

```
            rotation += step
          }
        }
        .buttonStyle(.borderedProminent)
        .disabled(!isRadial)
        Text("\(selection.count) selected")
          .font(.caption)
          .foregroundStyle(.secondary)
      }
    }
    // ... toggle
}
```

10. Finally, let the circle decide the reading order. When the user turns the toggle off, the chip sitting at the top of the circle should become the first chip in the flow, with the rest following clockwise. Reordering the array means it has to be mutable, so replace the `skills` constant with a `@State` property. Then, add a `showRadial` binding that aligns the data before the layout changes, switch the `Toggle` in `controls` to it, and add the `alignSkillsToRotation()` method that does the reordering, next to `toggle(_:)`:

```
struct ContentView: View {
  @State private var skills = Skill.sampleData
  // ... selection, isRadial, rotation, step, and layout
  private var showRadial: Binding<Bool> {
    Binding(
      get: { isRadial },
      set: { newValue in
        if !newValue {
          alignSkillsToRotation()
        }
        withAnimation(.spring(duration: 0.6)) {
          isRadial = newValue
        }
      }
```

```
      )
    }
    // ... body
    private var controls: some View {
      VStack(spacing: 16) {
        Toggle("Radial layout", isOn: showRadial)
        // ... the Rotate button and the selected count
      }
    }
    private func alignSkillsToRotation() {
      let count = skills.count
      guard count > 0 else { return }
      let turns = (rotation.degrees + 90) / step.degrees
      let offset = ((-Int(turns.rounded()) % count) + count)
      if offset != 0 {
        skills = Array(skills[offset...] + skills[..<offset])
      }
      rotation = .degrees(-90)
    }
    // ... toggle
  }
```

Tap **Rotate** a few times, then turn the toggle off. The chip that was at the top of the circle is now the first one in the flow. The circle itself does not jump at the moment we switch, because shifting the array and resetting the rotation cancel each other out exactly.

11. We have finished the app. Use the preview canvas in live mode: tap a few chips to select them, flip the toggle to watch them travel onto the circle, tap **Rotate** a few times, then turn the toggle off and see the chip that was at the top lead the flow. For the complete code, see the book's GitHub repository.



Figure 10.4: The skills wrapped by the flow layout, and the same chips arranged by t he radial layout

How it works…

The real contract in a custom layout is that measurement and placement agree. Our `FlowLayout` answers both of the protocol's requirements, `sizeThatFits()` and `placeSubviews()`, from the same `arrange(_:width:)` helper, so the rows it reports as its size are exactly the rows it later draws. That is the honest way to write a layout: the

size we report and the positions we place must come from the same reasoning, or the container will claim one shape and draw another. Keeping one arrangement function also means a wrapping fix applies to both phases at once.

A `ProposedViewSize` is an offer the child can answer however it likes, and its width and height are optional on purpose. A `nil` dimension means "tell me your ideal size", `0` means "tell me your minimum", and `.infinity` means "tell me your maximum". SwiftUI will call `sizeThatFits()` more than once with different proposals to learn how flexible our container is. That is why the flow layout runs its proposed width through `replacingUnspecifiedDimensions()` before wrapping: it needs a number to wrap against, and a `nil` would otherwise put every chip on one endless line. The two layouts answer the offer in opposite ways, which is the clearest illustration of what the proposal is for. The flow layout measures its chips and reports exactly the space they need, so it is as small as its content. The radial layout returns the proposal untouched, so it fills whatever it is given and arranges the chips inside.

Inside both methods, `subviews` is not a collection of views but of `LayoutSubview` proxies, and a proxy is all we need: we can ask it what size it would like with `sizeThatFits(.unspecified)`, and we can tell it where to go with `place(at:anchor:proposal:)`. Two habits are worth keeping. Place relative to `bounds.minX` and `bounds.minY`, never from zero, because the origin moves as soon as the layout is padded or nested. And take the gap between two children from their `ViewSpacing`, using `spacing.distance(to:along:)`, rather than hardcoding a number. That is why our chips sit at the same distance from each other as they

would inside an `HStack`: the spacing comes from the views themselves, not from a constant we invented.

`RadialLayout` shows that a layout can be animated like any other value. Conforming to `Animatable` through `animatableData` exposes the rotation as a single `Double`, and SwiftUI interpolates it across the animation, recomputing `placeSubviews()` at every frame. The chips do not slide along a straight line to their new spots; they travel around the circle, because the angle is what is being animated, not the position. The animations themselves, driven by `withAnimation`, are the subject of the *Animations, Transitions, and Matched Geometry* section of *Chapter 12*.

`AnyLayout` is what makes the switch feel like one screen instead of two. Written the obvious way, with an `if` that returns a `FlowLayout` in one branch and a `RadialLayout` in the other, SwiftUI would treat them as different views: the chips would be destroyed and recreated, and they would fade rather than move. Erasing both layouts into `AnyLayout` keeps a single view tree with a single set of chips, so changing the layout only changes where those chips are, which SwiftUI knows how to animate.

The last piece runs the other way, letting the geometry feed the model. Each chip sits at `rotation + step * index`, so after rotating, the chip at the top of the circle is the one whose index cancels the rotation out. Shifting the array by that offset and resetting the rotation to its starting angle are two changes that undo each other visually, which is why the circle does not move at the instant we switch, and yet the array now reads clockwise from the top. It is a small reminder that a custom layout is ordinary code: the same

arithmetic that positions a view can just as easily tell us somethi ng about the data.

See also

- `Layout` documentation: https://x.swiftuicookbook.com/rhtwh
- `AnyLayout` documentation: https://x.swiftuicookbook.com/rkamw
- `ProposedViewSize` documentation: https://x.swiftuicookbook.com/76mgw
- `LayoutSubview` documentation: https://x.swiftuicookbook.com/oj1mo
- `ViewSpacing` documentation: https://x.swiftuicookbook.com/k1uqd
- Compose custom layouts with SwiftUI (WWDC 2022): https://x.swiftuicookbook.com/l98hz

11

Navigation, Sheets, and Deep Linking

In every project I have worked on, navigation is the part of the app that grows without anyone deciding to grow it, and it never stays the way I first wired it. One screen becomes two, a modal appears for a small task that did not deserve a screen of its own, and before long there is a tab bar, a sidebar on iPad, a sheet that opens another sheet, and a link in an email that is supposed to land on one particular product screen. In UIKit each of those was its own mechanism, with its own opinion about who owned the stack, and the hardest problems showed up where two of them overlapped, such as presenting a modal from a view controller buried inside a navigation controller inside a tab bar controller. In SwiftUI, pushing a screen, presenting a sheet, switching tabs, and opening a link from outside the app all rest on a single idea: we say which value should be on screen, and the framework works out how to get there. That turns navigation into something we can store, restore, and drive from code, instead of a sequence of pushes we hope stays in order.

We start with a browser for national parks that adapts from one column on iPhone to two on iPad, built from a `NavigationSplitView` wrapped around a `NavigationStack` . `NavigationLink(value:)` and

`.navigationDestination(for:)` turn park and region values into screens, while a `NavigationPath` kept in an `@Observable` router lets code move through the hierarchy as easily as a tap does. We make each park shareable straight from the navigation bar with `.navigationDocument()`, and connect `@Namespace`, `.matchedTransitionSource()`, and `.navigationTransition(.zoom)` so a tapped card grows into its detail. A media library follows, built on `TabView`, where each `Tab` can carry a `.badge()`, `Tab(role: .search)` puts search where users already look for it, and a `TabSection` collects playlists into a collapsible group in the iPad sidebar that `.tabViewStyle(.sidebarAdaptable)` unlocks. Liquid Glass arrives through `.tabViewBottomAccessory()` for a mini player that survives every tab change, `.tabBarMinimizeBehavior()` for a tab bar that steps aside while the user scrolls, and `Tab(role: .prominent)` for the one action that should not look like the others.

From there we move to presentation, starting with a contact manager. It reaches for `.sheet(isPresented:)` when a flag is enough, `.sheet(item:)` when the sheet needs to know which contact it is editing, `.fullScreenCover(isPresented:)` for work that deserves the whole display, and `.popover()` paired with `.presentationCompactAdaptation()` for something anchored to the control that opened it, while `.interactiveDismissDisabled()` keeps an unsaved edit from disappearing with a careless swipe. A map of landmarks then pushes sheets further. `.presentationDetents()` and `.presentationDragIndicator()` rest a drawer at a chosen height, `.presentationBackgroundInteraction()` keeps the map underneath interactive, and `.presentationCornerRadius()` and `.presentationContentInteraction()` shape the rest, with `@Environment(\.dismiss)` closing the drawer from the inside. Last, a product catalog opens from outside the app altogether. A custom

URL scheme and `.onOpenURL()` turn an arriving `shoplinks://` address into a pushed screen, `@Environment(\.openURL)` sends a web address out to Safari, and `OpenURLAction` intercepts a link before it leaves so an address pointing back into our own catalog is handled in place. Universal links trade the custom scheme for an ordinary `https` address the system can verify.

By the end of this chapter, you should be comfortable modeling an app's navigation as data rather than as a sequence of pushes, picking the right presentation for a task and tuning how it behaves, and letting a link from outside the app land on exactly the screen it names.

In this chapter, we will cover the following recipes:

- NavigationStack and NavigationPath
- TabViews and Liquid Glass
- Modal Sheets and Full-Screen Covers
- Dismissal and Presentation Customization
- Deep Linking and URL Handling

Technical requirements

The code in this chapter is based on Xcode 27.0 and iOS 27.0. You can download and install the latest version of Xcode from the App Store. All the code examples for this chapter can be found on GitHub at https://x.swiftuicookbook.com/ch11 .

NavigationStack and NavigationPath

Navigation is where a lot of UIKit boilerplate used to live. A push meant a `UINavigationController` and a segue or a manual `pushViewController(_:animated:)` , an iPad layout meant a `UISplitViewController` with its own delegate dance, and deep links meant reconstructing that stack by hand. SwiftUI turns navigation into something we describe with data: we keep an array or a path of the values we have navigated to, and the framework builds the matching stack of views for us. Changing that data changes the displayed view, which is exactly what makes programmatic navigation and deep linking straightforward.

In this recipe, we build a browser for national parks that adapts from a single stack on iPhone to a two-column layout on iPad. A `NavigationSplitView` gives us the adaptive shell, with a sidebar of regions beside a detail column that holds a `NavigationStack` . Inside that stack we drive navigation with data: value-based `NavigationLink` s and `.navigationDestination(for:)` push a park's detail, and a `NavigationPath` kept in an `@Observable` router lets us move through the hierarchy in code. Because the path is heterogeneous, it carries both parks and regions. We add a zoom transition, so a tapped card expands into its detail using `@Namespace` , `.matchedTransitionSource()` , and `.navigationTransition(.zoom)` , and we use `.navigationDocument()` to make each park's page shareable straight from the navigation bar .

Getting ready

Create a new SwiftUI iOS app named **ParkGuide** . The standard iOS app template is universal, so the same code runs on iPhone and iPad, which is what lets us watch the split view adapt between the m.

How to do it...

Our app is built from a handful of small pieces: two model types and a navigation router, three view files for the card, the detail, and the grid, and a `ContentView` that assembles them into the adaptive shell. We will build the models and subviews first so we can verify each one in its own preview, then wire the navigation together in `ContentView`. These are the steps:

1. Create `Region.swift` in a `Models` folder for the four regions the sidebar lists. The template's default `import Foundation` becomes `import SwiftUI`, because each region carries a `tint` color we use for its cards. The enum is `Identifiable` and `Hashable` so it can drive a `List` selection and later ride on the navigation path, and it exposes an SF Symbol name and a color per case:

```
import SwiftUI
enum Region: String, CaseIterable, Identifiable, Hashable {
  case west = "West"
  case rockies = "Rockies"
  case southwest = "Southwest"
  case east = "East"
  var id: String { rawValue }
  var symbolName: String {
    switch self {
    case .west: "tree.fill"
    case .rockies: "mountain.2.fill"
    case .southwest: "sun.max.fill"
    case .east: "leaf.fill"
    }
  }
  var tint: Color {
    switch self {
    case .west: .green
    case .rockies: .indigo
    case .southwest: .orange
    case .east: .teal
```

```
    }
  }
}
```

2. Create `Park.swift` in the `Models` folder for the parks. The model stays small and purpose-built: an `id` of type `Int` for stable identity on the navigation path, a `region` that files each park under one of the four regions, the descriptive fields each view shows, a `url` that points at the park's page so we can share it later, an `isFavorite` flag the cards mark with a star, and two computed strings for the detail view, `areaFormatted` and `establishedText`, the latter a plain string so the year never picks up a thousands separator. The `all` array seeds twelve parks, of which only the first is shown here. The `inRegion(_:)` and `favorites` helpers filter that list, and `featured` picks one park out for a quick jump:

```
import Foundation
struct Park: Identifiable, Hashable {
  let id: Int
  let name: String
  let region: Region
  let tagline: String
  let symbolName: String
  let established: Int
  let areaAcres: Int
  let highlights: [String]
  let url: URL
  var isFavorite: Bool = false
  var areaFormatted: String {
    "\(areaAcres.formatted()) acres"
  }
  var establishedText: String {
    String(established)
  }
```

```
}
extension Park {
  static let all: [Park] = [
    Park(
      id: 1,
      name: "Yosemite",
      region: .west,
      tagline: "Granite cliffs and giant sequoias",
      symbolName: "mountain.2.fill",
      established: 1890,
      areaAcres: 761266,
      highlights: ["El Capitan", "Half Dome", "Yosemite Fal
      url: URL(string: "https://www.nps.gov/yose/")!,
      isFavorite: true
    ),
    // ... the remaining eleven parks
  ]
  static let featured = all.first { $0.name == "Grand Canyo
  static func inRegion(_ region: Region) -> [Park] {
    all.filter { $0.region == region }
  }
  static var favorites: [Park] {
    all.filter { $0.isFavorite }
  }
}
```

For the complete initialization code of the `all` static property, see the book's GitHub repository.

3. Create `ParkRouter.swift` in the `Models` folder. This is the object that owns our navigation state. It holds a `NavigationPath`, the type-erased list of values the stack has pushed, and exposes small methods to append a park or a region and to clear the path. Keeping the path in an `@Observable` object, rather than a local `@State` value, is what lets any view drive navigation programmatically by calling a method instead of reaching into a binding:

```
import SwiftUI
@MainActor
@Observable
final class ParkRouter {
  var path = NavigationPath()
  func show(_ park: Park) {
    path.append(park)
  }
  func show(_ region: Region) {
    path.append(region)
  }
  func popToRoot() {
    path = NavigationPath()
  }
}
```

4. Create `ParkCardView.swift` in a `Views` folder. This is the smallest visible piece, the tile the grid repeats, so we build it first and confirm it in the preview canvas. It shows the park's symbol, a favorite star when the park is marked, and the name and tagline over a gradient drawn from the region's tint:

```
import SwiftUI
struct ParkCardView: View {
  let park: Park
  var body: some View {
    VStack(alignment: .leading, spacing: 8) {
      HStack {
        Image(systemName: park.symbolName)
          .font(.system(size: 36))
        Spacer()
        if park.isFavorite {
          Image(systemName: "star.fill")
        }
      }
      Spacer(minLength: 0)
      Text(park.name)
        .font(.headline)
      Text(park.tagline)
```

```
        .font(.caption)
        .lineLimit(2)
    }
    .foregroundStyle(.white)
    .padding()
    .frame(height: 150, alignment: .topLeading)
    .frame(maxWidth: .infinity, alignment: .leading)
    .background(park.region.tint.gradient, in: .rect(corner
  }
}
#Preview {
  ParkCardView(park: Park.all[0])
    .padding()
}
```

5. Create `ParkDetailView.swift` in the `Views` folder, the view we push when a card is tapped. We build it up over the next three steps; start with a `header` property, a large symbol over the region's color with the tagline beneath it. The preview wraps the view in a `NavigationStack` and hands it a `ParkRouter`, which the navigation touches we add shortly will read:

```
import SwiftUI
struct ParkDetailView: View {
  let park: Park
  var body: some View {
    ScrollView {
      VStack(alignment: .leading, spacing: 24) {
        header
      }
      .padding()
    }
    .navigationTitle(park.name)
    .navigationBarTitleDisplayMode(.inline)
  }
  private var header: some View {
    VStack(spacing: 12) {
      Image(systemName: park.symbolName)
```

```
        .font(.system(size: 64))
      Text(park.tagline)
        .font(.headline)
        .multilineTextAlignment(.center)
    }
    .foregroundStyle(.white)
    .frame(maxWidth: .infinity)
    .padding(.vertical, 48)
    .background(park.region.tint.gradient, in: .rect(corner
  }
}
#Preview {
  NavigationStack {
    ParkDetailView(park: Park.all[0])
  }
  .environment(ParkRouter())
}
```

6. Add a `facts` row and a `highlights` list below the `header`. The `facts` row shows the year the park was established and its area as two `Label`s, and the `highlights` list is a `ForEach` of checkmarked lines. Add both to the `VStack` in `body`, then add the two computed properties after `header`:

```
struct ParkDetailView: View {
  let park: Park
  var body: some View {
    ScrollView {
      VStack(alignment: .leading, spacing: 24) {
        header
        facts
        highlights
      }
      .padding()
    }
    // ... navigationTitle and display mode
  }
  // ... header
  private var facts: some View {
```

```
      HStack {
        Label("Est. \(park.establishedText)", systemImage: "c
        Spacer()
        Label(park.areaFormatted, systemImage: "ruler")
      }
      .font(.subheadline)
      .foregroundStyle(.secondary)
    }
    private var highlights: some View {
      VStack(alignment: .leading) {
        Text("Highlights")
          .font(.title2)
          .fontWeight(.bold)
        ForEach(park.highlights, id: \.self) { highlight in
          Label(highlight, systemImage: "checkmark.circle.fil
        }
      }
      .frame(maxWidth: .infinity, alignment: .leading)
    }
}
```

7. Finish the detail with its navigation touches. Read the `ParkRouter` from the environment, then add three things. First, `.navigationDocument(park.url)` makes the title shareable. On the running app, a small disclosure arrow appears next to the title, and tapping the title opens a menu with the park's link and a share icon. Second, a toolbar button clears the path with `popToRoot()`; because it is a `Button(_:systemImage:)`, the toolbar shows its list icon while the title stays as the accessibility label. Third, a bordered **More parks in** button pushes the park's region onto the path:

```
struct ParkDetailView: View {
  let park: Park
  @Environment(ParkRouter.self) private var router
  var body: some View {
```

```
      ScrollView {
        VStack(alignment: .leading, spacing: 24) {
          // ... header, facts, and highlights
          exploreButton
        }
        .padding()
      }
      // ... navigationTitle and display mode
      .navigationDocument(park.url)
      .toolbar {
        ToolbarItem(placement: .primaryAction) {
          Button("All Parks", systemImage: "list.bullet") {
            router.popToRoot()
          }
        }
      }
    }
    // ... header, facts, and highlights properties
    private var exploreButton: some View {
      Button {
        router.show(park.region)
      } label: {
        Label("More parks in \(park.region.rawValue)", syster
          .frame(maxWidth: .infinity)
      }
      .buttonStyle(.bordered)
    }
}
```

8. Create `ParkGridView.swift` in the `Views` folder, a titled grid of cards for a set of parks. Each card is wrapped in a value-based `NavigationLink(value: park)`, which is the data-driven form: instead of naming a destination view inline, the link just carries the park value, and a `.navigationDestination(for:)` we add later decides what to show. We also mark each card as a zoom source with `.matchedTransitionSource(id:in:)`, using the park's `id` and a shared namespace, so the detail can grow out of the exact card

the user tapped once we wire the matching transition while assembling the navigation:

```
import SwiftUI
struct ParkGridView: View {
  let title: String
  let parks: [Park]
  let namespace: Namespace.ID
  private let columns = [GridItem(.adaptive(minimum: 150),
  var body: some View {
    ScrollView {
      LazyVGrid(columns: columns, spacing: 16) {
        ForEach(parks) { park in
          NavigationLink(value: park) {
            ParkCardView(park: park)
              .matchedTransitionSource(id: park.id, in: nam
          }
          .buttonStyle(.plain)
        }
      }
      .padding()
    }
    .navigationTitle(title)
  }
}
#Preview {
  @Previewable @Namespace var namespace
  NavigationStack {
    ParkGridView(
      title: "West",
      parks: Park.inRegion(.west),
      namespace: namespace
    )
    .navigationDestination(for: Park.self) { park in
      ParkDetailView(park: park)
    }
  }
  .environment(ParkRouter())
}
```

9. Now, assemble the shell. Open `ContentView.swift` and replace everything except the `#Preview` at the bottom with the following. A small `SidebarItem` enum lets the sidebar select either a region or the favorites shortcut. The `NavigationSplitView` puts a `List` of regions and a **Favorites** row in the sidebar, and a `NavigationStack(path:)` in the detail column, whose root is a single `ParkGridView` fed by `detailTitle` and `detailParks`. We inject the router and declare the `@Namespace` here. A namespace is the shared identifier space the zoom transition uses to pair a source card with the detail it opens, so it belongs in the one view that owns both the grid and the detail:

```
import SwiftUI
enum SidebarItem: Hashable {
  case region(Region)
  case favorites
}
struct ContentView: View {
  @State private var router = ParkRouter()
  @State private var selection: SidebarItem? = .region(.wes
  @Namespace private var namespace
  var body: some View {
    NavigationSplitView {
      List(selection: $selection) {
        Section("Regions") {
          ForEach(Region.allCases) { region in
            Label(region.rawValue, systemImage: region.symb
              .foregroundStyle(region.tint)
              .tag(SidebarItem.region(region))
          }
        }
        Section {
          Label("Favorites", systemImage: "star.fill")
            .foregroundStyle(.yellow)
            .tag(SidebarItem.favorites)
        }
      }
```

```
        .navigationTitle("ParkGuide")
      } detail: {
        NavigationStack(path: $router.path) {
          ParkGridView(
            title: detailTitle,
            parks: detailParks,
            namespace: namespace
          )
        }
      }
      .environment(router)
    }
    private var detailTitle: String {
      switch selection {
      case .region(let region): region.rawValue
      case .favorites: "Favorites"
      case nil: "ParkGuide"
      }
    }
    private var detailParks: [Park] {
      switch selection {
      case .region(let region): Park.inRegion(region)
      case .favorites: Park.favorites
      case nil: []
      }
    }
}
```

With the preview canvas in live mode on an iPhone, the app opens on the West grid, because the selection starts at `.region(.west)` and a collapsed split view shows the selected detail first. Tap the back button at the top left to reach the sidebar, then select a region or **Favorites** to fill the detail with the matching grid. Tapping a card does nothing yet, because we have not told the stack what a `Park` value should show, which is the next step.

10. Make the cards navigate. A value-based `NavigationLink` only contributes a value; a `.navigationDestination(for: Park.self)` is what maps that value to a view. Attach one to the grid so a tapped park pushes its detail:

```
ParkGridView(
  title: detailTitle,
  parks: detailParks,
  namespace: namespace
)
.navigationDestination(for: Park.self) { park in
  ParkDetailView(park: park)
}
```

    Tapping a card now pushes the park's detail, with the standard slide animation.

11. The detail's **More parks in** button appends a `Region` to the same path, so register a destination for that type too. This is what makes the path heterogeneous: one `NavigationPath` carrying two different kinds of value. Add a second destination beside the first:

```
.navigationDestination(for: Park.self) { park in
  ParkDetailView(park: park)
}
.navigationDestination(for: Region.self) { region in
  ParkGridView(
    title: region.rawValue,
    parks: Park.inRegion(region),
    namespace: namespace
  )
}
```

From a park's detail, **More parks in** now pushes that region's grid.

12. Turn the plain push into a zoom, so the detail grows out of the card the user tapped. The cards already declare themselves as sources with `.matchedTransitionSource()`, so we only need to add the matching transition on the destination, using the same `park.id` and `namespace`:

```
.navigationDestination(for: Park.self) { park in
  ParkDetailView(park: park)
    .navigationTransition(.zoom(sourceID: park.id, in: name
}
// ... the Region destination
```

Tapping a card now zooms into its detail instead of sliding.

13. Add a toolbar button, shown as a sparkles icon, that jumps straight to a chosen park with `router.show(Park.featured)`, which shows that we can navigate in code without a link on screen at all. Attach the `.toolbar` to the grid, after the destinations:

```
// ... the grid and its two navigationDestination modifiers
.toolbar {
  ToolbarItem(placement: .primaryAction) {
    Button("Featured", systemImage: "sparkles") {
      router.show(Park.featured)
    }
  }
}
```

In the preview canvas, tap the sparkles icon and the Grand Canyon detail appears, with no link involved.

14. Finally, reset the path whenever the sidebar selection changes, so switching regions always starts the detail from a clean stack rather than leaving stale pushes on top. Add an `.onChange` to the `NavigationStack`:

```
NavigationStack(path: $router.path) {
  // ... the grid, its destinations, and the toolbar
}
.onChange(of: selection) {
  router.popToRoot()
}
```

15. Run the app on a simulated iPhone and iPad in Device Hub, or drag a resizable preview across the compact boundary, to see the same code adapt. On iPhone the `NavigationSplitView` collapses to a stack that opens on the selected region's grid, the back button leads to the sidebar, tapping a region drills into its grid, and tapping a card zooms into the detail. On iPad the sidebar and grid sit side by side in two columns. In both, the sparkles toolbar button jumps to a park, the **More parks in** button pushes a region grid, and tapping a park's title shares its page. For the complete `ContentView` code, see the book's GitHub repository.

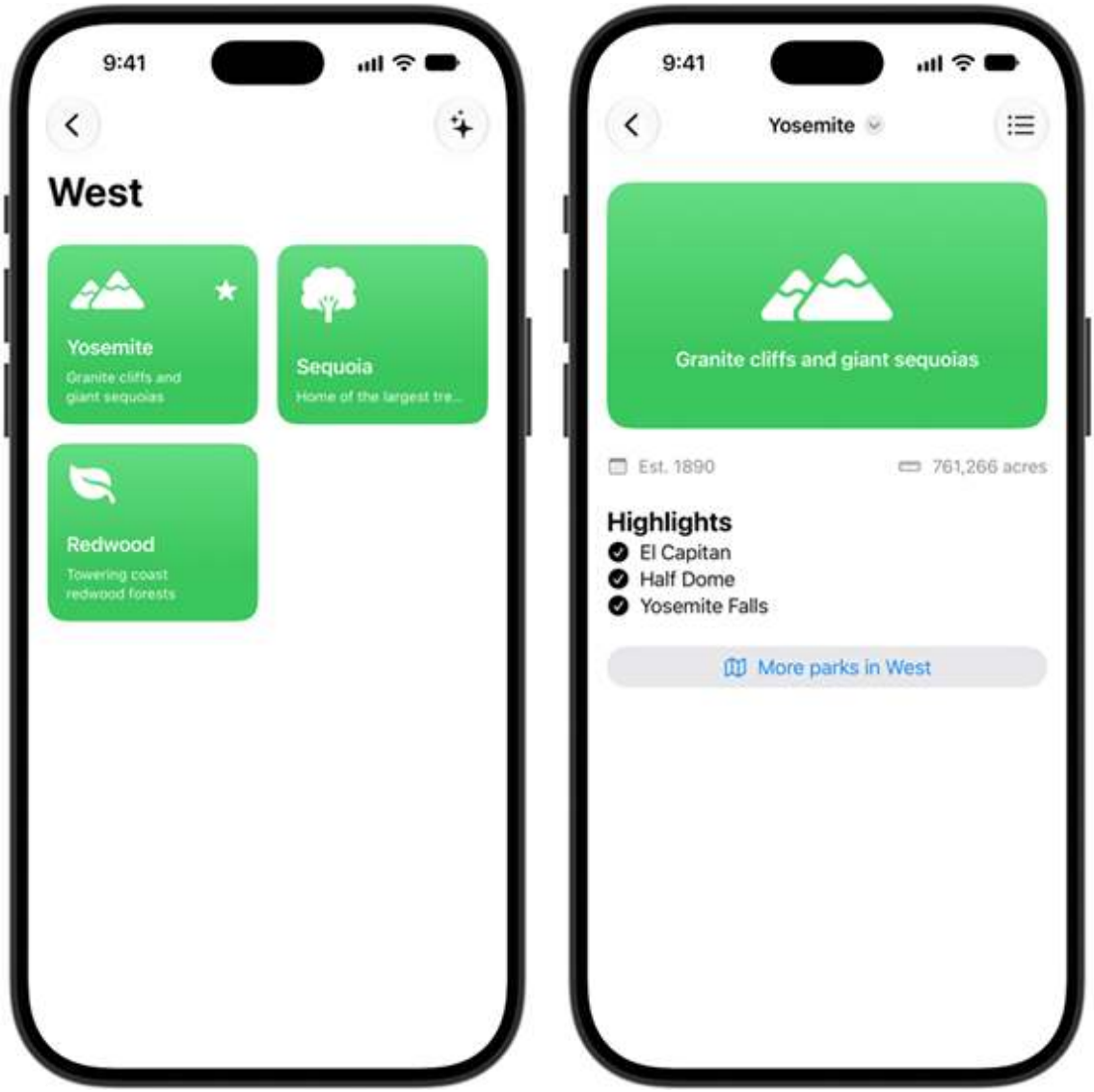


Figure 11.1: A region's parks and a park's detai l on iPhone

How it works…

The whole app hangs off one `NavigationSplitView`. It is an adaptive container: in a regular horizontal size class, like an iPad in landscape, it lays its columns out side by side, and in a compact size class, like any iPhone, it collapses into a single `NavigationStack` where selecting a sidebar row pushes the detail. Because our selection starts at a region

rather than `nil`, that collapsed stack opens on the region's grid instead of the sidebar, and on iPad the same default keeps the detail column from starting empty. We wrote the two-column form, sidebar plus detail, and got both behaviors for free. The pattern to internalize is that the split view owns the top-level structure while a `NavigationStack` inside the detail column owns the drill-down. Our detail is exactly that: a `NavigationStack(path:)` whose pushes, the park details and region grids, stack up inside the column on iPad and inside the single collapsed stack on iPhone.

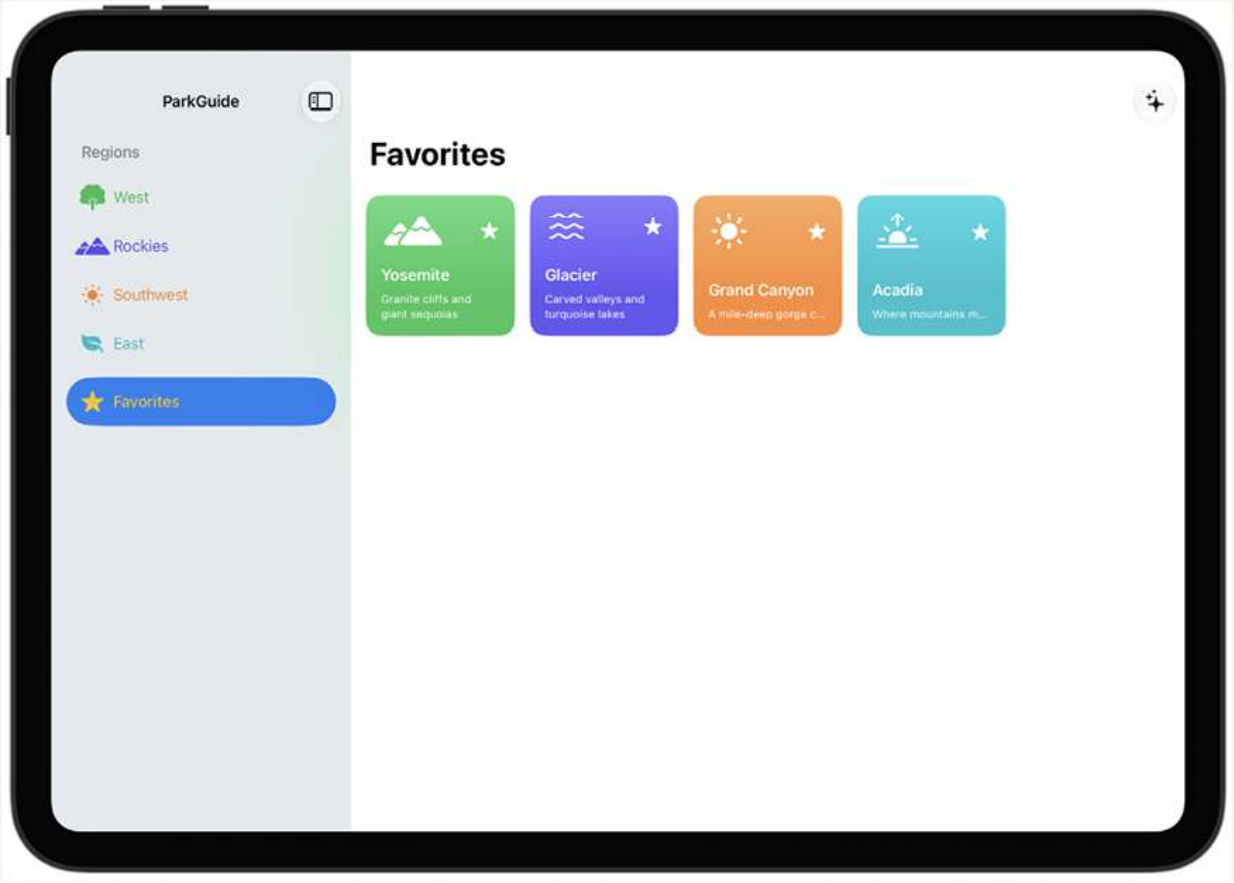


Figure 11.2: The same code on iPad, with the sidebar beside the Favorites grid

`NavigationSplitView` also comes in a three-column form, with a sidebar, a middle content list, and a detail, initialized as `NavigationSplitView {`

`sidebar } content: { list } detail: { detail }`. That form is driven entirely by selection bindings rather than a navigation path, and it suits larger, list-heavy apps such as Mail, where the user picks a mailbox, then a message, then reads it. We chose the two-column form here precisely because it keeps a real `NavigationStack` in the detail, which is what this recipe is about.

Data-driven navigation is the thread running through the detail column. A value-based `NavigationLink(value:)` does not name a destination; it just contributes a value, and a `.navigationDestination(for:)` elsewhere in the stack maps that type to a view. Because we registered destinations for two types, `Park` and `Region`, the stack's `NavigationPath` is heterogeneous: tapping a card appends a `Park`, and the detail's **More parks in** button appends a `Region`, and each lands on the right view. Keeping that path in an `@Observable ParkRouter` is what unlocks programmatic navigation: the sparkles toolbar button pushes a park by calling `router.show(Park.featured)`, and the detail's list-icon toolbar button empties the path with `popToRoot()`, neither of which needs a link on screen. We will look at the `@Observable` and `@Environment` machinery behind the router in the *State and Binding Basics* section of *Chapter 5*.

The zoom transition is a three-part contract, all keyed on one shared `@Namespace`. The source card marks itself with `.matchedTransitionSource(id: park.id, in: namespace)`, the destination requests `.navigationTransition(.zoom(sourceID: park.id, in: namespace))`, and because the `id` and namespace match, SwiftUI animates the detail as if it grew out of that specific card. Using the park's own `id` as the source `id` is what keeps the match correct when a whole grid

of cards shares the namespace. Zoom transitions read best when the source is a sizable tile, like our cards, rather than a thin list row, where the change in aspect ratio makes the animation look awkward, which is why the grid, and not the sidebar list, is the thing that zooms. This zoom is our first use of SwiftUI's transition system, and the shared `@Namespace` it relies on is the same mechanism behind the matched-geometry effect; we cover both in depth in the *Animations, Transitions, and Matched Geometry* section of *Chapter 12*.

Two smaller decisions keep the navigation dependable. `.navigationDocument(park.url)` hands the navigation bar a document to represent. The modifier marks the title as tappable with a small disclosure arrow, and the menu that opens from it shares the park's page; it is a system affordance we get from a single modifier rather than a custom share button. And resetting the path in `.onChange(of: selection)` matters because the router's path outlives any single region: without the reset, switching regions in the sidebar would leave stale pushes on top of the new root. Clearing it means every sidebar change starts the detail from a clean grid.

See also

- `NavigationSplitView` documentation: https://x.swiftuicookbook.com/ilpnw
- `NavigationStack` documentation: https://x.swiftuicookbook.com/1ocv4
- `NavigationPath` documentation: https://x.swiftuicookbook.com/f6kji

- `navigationDestination(for:destination:)` documentation: https://x.swiftuicookbook.com/67722
- `matchedTransitionSource(id:in:)` documentation: https://x.swiftuicookbook.com/2gno2
- `navigationTransition(_:)` documentation: https://x.swiftuicookbook.com/u4ejs
- `navigationDocument(_:)` documentation: https://x.swiftuicookbo ok.com/a1vk4

TabViews and Liquid Glass

A tab bar used to be a `UITabBarController` holding an array of view controllers, each one carrying a `UITabBarItem` configured separately from the controller it belonged to. Nothing tied an item to the selected index except an integer tag, so getting one wrong was a runtime surprise rather than a compile error. SwiftUI closed that gap: a tab declares its label, its selection value, and its content in one place, and the compiler checks that every tab in the same `TabView` agrees on the type of that value.

In this recipe, we build a music library whose entire structure is its tab bar. Five destinations give us a reason to use each part of the API: a `Tab` with a `.badge()` for the library, a `TabSection` grouping two playlists, a search destination that takes `Tab(role: .search)` so the system places it by convention, and a station promoted with `Tab(role: .prominent)` to a pinned slot of its own. Around them, `.tabViewStyle(.sidebarAdaptable)` renders the same tabs as a bottom bar on iPhone and a sidebar on iPad, `.tabBarMinimizeBehavior(.onScrollDown)` gets the bar out of the way while the user scrolls, and

`.tabViewBottomAccessory()` parks a mini player above the bar that follows it down as it minimizes. Those last two are where the Liquid Glass design shows up in practice: the bar floats over the content instead of sitting under it, so it can aff ord to shrink.

Getting ready

Create a new SwiftUI iOS app named **MediaLibrary** . As in the previous recipe, the standard template is universal, so the same code runs on iPhone and iPad, which is what makes the sidebar be havior visible.

How to do it...

The app has one data file, two small views, and a `ContentView` that is almost nothing but a `TabView` . We will put the data and the views in place first, then start from the simplest tab bar that runs and layer one API onto it at a time, checking each addition before moving on. These are the steps:

1. Create `Track.swift` in a `Models` folder. The model stays deliberately small, because the tab bar is the subject here and the data only has to be convincing enough to fill five destinations: an `id` for identity in a `List` , a title and an artist to display, and a duration in seconds. A row has to show that duration as text rather than as a number, and `durationText` is where that conversion lives: it wraps the count in a `Duration` and formats it with the `.minuteSecond` pattern to produce the familiar `3:34` . Reaching for a format style instead of dividing by 60 and zero-padding by hand keeps the intent in the code, and it leaves the

separator to Foundation, which picks the one the user's locale expects. The `all` array holds twelve tracks, of which two are shown here:

```
import Foundation
struct Track: Identifiable, Hashable {
  let id: Int
  let title: String
  let artist: String
  let duration: Int
  var durationText: String {
    Duration.seconds(duration).formatted(.time(pattern: .mi
  }
}
extension Track {
  static let all: [Track] = [
    Track(id: 1, title: "Night Drive", artist: "Marisa Vale
    Track(id: 2, title: "Paper Lanterns", artist: "Blue Hou
    // ... the remaining ten tracks
  ]
}
```

For the complete initialization code of the `all` static property, see the book's GitHub repository.

2. Every tab shows a different slice of the `all` array. Add the helpers that produce those slices to the same extension. `recent` feeds a badge count, `radio` and the two playlists `focus` and `workout` pick out fixed subsets. `matching(_:)` backs the search destination, returning everything when the query is empty, so an untouched search tab still shows the whole library:

```
extension Track {
  // ... the all array
  static var recent: [Track] {
```

```
      Array(all.prefix(4))
    }
    static var radio: [Track] {
      all.filter { $0.id.isMultiple(of: 3) }
    }
    static var focus: [Track] {
      Array(all.prefix(5))
    }
    static var workout: [Track] {
      Array(all.suffix(5))
    }
    static func matching(_ text: String) -> [Track] {
      guard !text.isEmpty else { return all }
      return all.filter {
        $0.title.localizedStandardContains(text)
        || $0.artist.localizedStandardContains(text)
      }
    }
}
```

3. Create `TrackListView.swift` in a `Views` folder. Every destination displays the same thing, a list of tracks, so a single view parameterized by a title and an array serves all five. Tapping a row writes to a `nowPlaying` binding, which the mini player will read later. Wrapping the row in a plain-styled `Button` keeps the row's own layout, and `.contentShape(.rect)` gives that layout a tappable rectangle, so the whole row responds rather than only the text: the gap the `Spacer` opens up is empty space, and empty space is not hit-testable on its own:

```
import SwiftUI
struct TrackListView: View {
  let title: String
  let tracks: [Track]
  @Binding var nowPlaying: Track
  var body: some View {
    List(tracks) { track in
```

```
        Button {
          nowPlaying = track
        } label: {
          HStack {
            Image(systemName: "music.note")
              .foregroundStyle(.pink)
              .frame(width: 28)
            VStack(alignment: .leading) {
              Text(track.title)
                .font(.headline)
              Text(track.artist)
                .font(.subheadline)
                .foregroundStyle(.secondary)
            }
            Spacer()
            Text(track.durationText)
              .font(.caption)
              .foregroundStyle(.secondary)
              .monospacedDigit()
          }
          .contentShape(.rect)
        }
        .buttonStyle(.plain)
      }
      .navigationTitle(title)
    }
}
#Preview {
  @Previewable @State var nowPlaying = Track.all[0]
  NavigationStack {
    TrackListView(title: "Library", tracks: Track.all, nowP
  }
}
```

4. Let's start in `ContentView.swift` with the smallest `TabView` that works: two `Tab` values, each declaring its title, its SF Symbol, and its content in one place. Pass the outline variant of a symbol rather than the filled one, `square.stack` and not `square.stack.fill`, because the system picks the filled variant for a tab bar and the

outline variant for a sidebar on its own. Replace the `ContentView` struct with the following:

```
struct ContentView: View {
  @State private var nowPlaying = Track.all[0]
  var body: some View {
    TabView {
      Tab("Library", systemImage: "square.stack") {
        NavigationStack {
          TrackListView(title: "Library", tracks: Track.all
                        nowPlaying: $nowPlaying)
        }
      }
      Tab("Radio", systemImage: "dot.radiowaves.left.and.ri
        NavigationStack {
          TrackListView(title: "Radio", tracks: Track.radic
                        nowPlaying: $nowPlaying)
        }
      }
    }
  }
}
```

With the preview canvas in live mode, the two tabs already switch content.

5. Nothing outside the `TabView` currently knows which tab is showing. Give the selection a type by declaring an enum above `ContentView` with one case per destination, then bind it and tag each tab with its `value`. This is what the `Tab` API buys us over the older `.tabItem()` modifier: the compiler now checks that every tab in this `TabView` carries the same kind of value, instead of trusting loose integer tags:

```
enum LibraryTab: Hashable {
  case library
  case focus
  case workout
  case radio
  case search
}
struct ContentView: View {
  @State private var selection: LibraryTab = .library
  // ... nowPlaying
  var body: some View {
    TabView(selection: $selection) {
      Tab("Library", systemImage: "square.stack", value: .l
        // ... the Library content
      }
      Tab("Radio", systemImage: "dot.radiowaves.left.and.ri
          value: .radio) {
        // ... the Radio content
      }
    }
  }
}
```

6. Advertise how many tracks arrived recently with a badge. `.badge()` attaches to the tab itself rather than to its content, and takes a count or a string:

```
struct ContentView: View {
  // ... the state properties
  var body: some View {
    TabView(selection: $selection) {
      Tab("Library", systemImage: "square.stack", value: .l
        // ... the Library content
      }
      .badge(Track.recent.count)
      // ... the Radio tab
    }
  }
}
```

In the preview canvas the **Library** tab now carries a 4. Change the `prefix(4)` in `recent` to another count and the badge follows.

7. Group the two playlists into a `TabSection`, placed between the library and radio tabs. In a tab bar its tabs sit alongside the others, but as soon as the layout has room for a sidebar the section becomes a titled, collapsible group. Spell out the type on each `value` here: `TabSection` works out its selection type from the tabs nested inside it, so a leading-dot expression would make that inference circular and the code would not compile:

```
struct ContentView: View {
  // ... the state properties
  var body: some View {
    TabView(selection: $selection) {
      // ... the Library tab and its badge
      TabSection("Playlists") {
        Tab("Focus", systemImage: "moon", value: LibraryTab
          NavigationStack {
            TrackListView(title: "Focus", tracks: Track.foc
                          nowPlaying: $nowPlaying)
          }
        }
        Tab("Workout", systemImage: "figure.run", value: Li
          NavigationStack {
            TrackListView(title: "Workout", tracks: Track.w
                          nowPlaying: $nowPlaying)
          }
        }
      }
      // ... the Radio tab
    }
  }
}
```

8. One modifier lets those same tabs become a sidebar wherever there is room for one. Add it to the `TabView`:

```
TabView(selection: $selection) {
  // ... all the tabs
}
.tabViewStyle(.sidebarAdaptable)
```

   On iPhone nothing changes. On iPad the tabs move into a floating bar at the top, with a button that swaps that bar for a sidebar where **Playlists** appears as a group holding the two playlists.

9. Give the station a slot of its own. Setting the `prominent` role on a tab pulls it out of the evenly spaced run and pins it to a separate position at the trailing edge of the tab bar, which suits a destination that should always be one tap away. Unlike the search role we add next, a prominent tab keeps its title, its symbol, and its content, and is simply emphasized. Add the role to the existing `Radio` tab:

```
struct ContentView: View {
  // ... the state properties
  var body: some View {
    TabView(selection: $selection) {
      // ... the Library tab and the Playlists section
      Tab("Radio", systemImage: "dot.radiowaves.left.and.ri
          value: .radio, role: .prominent) {
        // ... the Radio content
      }
    }
    .tabViewStyle(.sidebarAdaptable)
  }
}
```

In the preview canvas, **Radio** now sits apart from the other tabs, in a capsule of its own at the trailing edge.



New in iOS 27. The `prominent` role is new to `TabRole` in iOS 27. On iPad, where the tabs render in a floating top bar or a sidebar rather than a bottom tab bar, the promoted tab appears in line with the others instead of pinned.

10. Add the search destination. `Tab(role: .search)` needs neither a title nor a symbol, because the system supplies the magnifying glass and positions the tab by platform convention. Its content is the same list, filtered by a `searchText` property that `.searchable()` binds to, so add that property alongside the others and the tab after the radio tab:

```
struct ContentView: View {
  // ... the other state properties
  @State private var searchText = ""
  var body: some View {
    TabView(selection: $selection) {
      // ... the Library tab, the Playlists section, and th
      Tab(value: .search, role: .search) {
        NavigationStack {
          TrackListView(title: "Search", tracks: Track.matc
                        nowPlaying: $nowPlaying)
            .searchable(text: $searchText)
        }
      }
    }
    .tabViewStyle(.sidebarAdaptable)
  }
}
```

We cover `.searchable()` and the filtering patterns around it in the *Search and Filtering* section of *Chapter 2*.

11. In the Liquid Glass design the tab bar floats above the content rather than sitting below it, which means it can get out of the way. Ask it to minimize as the user scrolls down:

```
TabView(selection: $selection) {
  // ... all the tabs
}
.tabViewStyle(.sidebarAdaptable)
.tabBarMinimizeBehavior(.onScrollDown)
```

    Scroll the library list in the preview canvas and the bar shrinks to a capsule showing only the current tab; scrolling back up restores it.

12. Create `NowPlayingView.swift` in the `Views` folder, the mini player that will ride above the tab bar. It reads `tabViewBottomAccessoryPlacement` from the environment, the value SwiftUI uses to tell an accessory how much room it has: `.expanded` while the tab bar is at full size, and a narrower placement once the bar minimizes and the accessory has to share a row with it. We show the artist only in the expanded case:

```
import SwiftUI
struct NowPlayingView: View {
  let track: Track
  @Binding var isPlaying: Bool
  @Environment(\.tabViewBottomAccessoryPlacement) private v
  var body: some View {
    HStack(spacing: 12) {
      Image(systemName: "music.note")
        .foregroundStyle(.pink)
      if placement == .expanded {
```

```
          VStack(alignment: .leading) {
            Text(track.title)
              .font(.caption)
              .fontWeight(.semibold)
              .lineLimit(1)
            Text(track.artist)
              .font(.caption2)
              .foregroundStyle(.secondary)
              .lineLimit(1)
          }
        } else {
          Text(track.title)
            .font(.caption)
            .lineLimit(1)
        }
        Spacer()
        let image = isPlaying ? "pause.fill" : "play.fill"
        Button(isPlaying ? "Pause" : "Play", systemImage: ima
          isPlaying.toggle()
        }
        .labelStyle(.iconOnly)
      }
      .padding(.horizontal)
    }
}
#Preview {
  @Previewable @State var isPlaying = false
  NowPlayingView(track: Track.all[0], isPlaying: $isPlaying
}
```

13. Attach the mini player to the `TabView`. The accessory belongs to the tab bar rather than to any one tab, so it stays on screen as the user moves between destinations, and the system gives it a Liquid Glass capsule background without any styling from us. Add the play state alongside the other properties, and the modifier after the two we already have:

```
struct ContentView: View {
  @State private var selection: LibraryTab = .library
  @State private var nowPlaying = Track.all[0]
  @State private var isPlaying = false
  @State private var searchText = ""
  var body: some View {
    TabView(selection: $selection) {
      // ... all the tabs
    }
    .tabViewStyle(.sidebarAdaptable)
    .tabBarMinimizeBehavior(.onScrollDown)
    .tabViewBottomAccessory {
      NowPlayingView(track: nowPlaying, isPlaying: $isPlayi
    }
  }
}
```

Back in the preview canvas, tapping a row loads that track into the mini player, and scrolling down slides the player inline beside the minimized bar, where it drops the artist line.

14. Run the app on a simulated iPhone and iPad in Device Hub to see how far one `TabView` stretches. On iPhone the four ordinary destinations share the bar while **Radio** sits pinned in its own capsule to their right, and scrolling collapses the whole arrangement. On iPad the same tabs float in a bar at the top, the sidebar button turns them into a list with **Playlists** as a group, and the mini player spans the bottom. For the complete `ContentView` code, see the book's GitHub repository.



Figure 11.3: The tab bar with the mini player, at f ull size and minimized

How it works…

The `Tab` type replaced a pattern where a tab's identity lived in two places at once: a `.tabItem()` modifier on the content supplied the label, and a separate `.tag()` supplied the selection value. Declaring all three together is what lets the compiler tie a `TabView` to a single selection type, so our `LibraryTab` enum turns an invalid tab reference into a

build error. `TabSection` is the one place that inference needs help, because it derives its selection type from the tabs nested inside it, which is why the playlists spell out `LibraryTab.focus` rather than `.focus`. The older `.tabItem()` form still compiles, but it gives up sidebar adaptability and sections, so there is little reason to reach for it in new code.

Roles are how a tab asks to be positioned rather than merely listed, and our two are complementary. `Tab(role: .search)` gives up its title and symbol to the system, which supplies the magnifying glass and puts search where each platform expects it. `Tab(role: .prominent)` keeps all three and asks only to be pulled out of the evenly spaced run into a pinned trailing slot. Both are requests rather than guarantees, which is the useful thing to internalize: on iPad, where `.sidebarAdaptable` moves the tabs into a floating top bar or a sidebar, the prominent tab renders in line with the rest. Space in a tab bar is finite, so keep the ordinary destinations to roughly four alongside a pinned tab. Adding more pushes the overflow into a **More** item, and a search tab that lands there has lost the very placement its role asked for.

The two Liquid Glass modifiers work as a pair, and they only make sense because the bar now floats over the content instead of reserving space below it. `.tabBarMinimizeBehavior(.onScrollDown)` trades the bar for a single capsule while the user reads, and `.tabViewBottomAccessory()` attaches a view to the bar rather than to a tab, so our mini player survives every switch between destinations. When the bar minimizes, the accessory travels with it, which is what the `tabViewBottomAccessoryPlacement` environment value reports. Branching on `.expanded` is not decoration: the inline placement is genuinely

narrower, so `NowPlayingView` drops the artist line and keeps the title readable rather than letting both truncate.

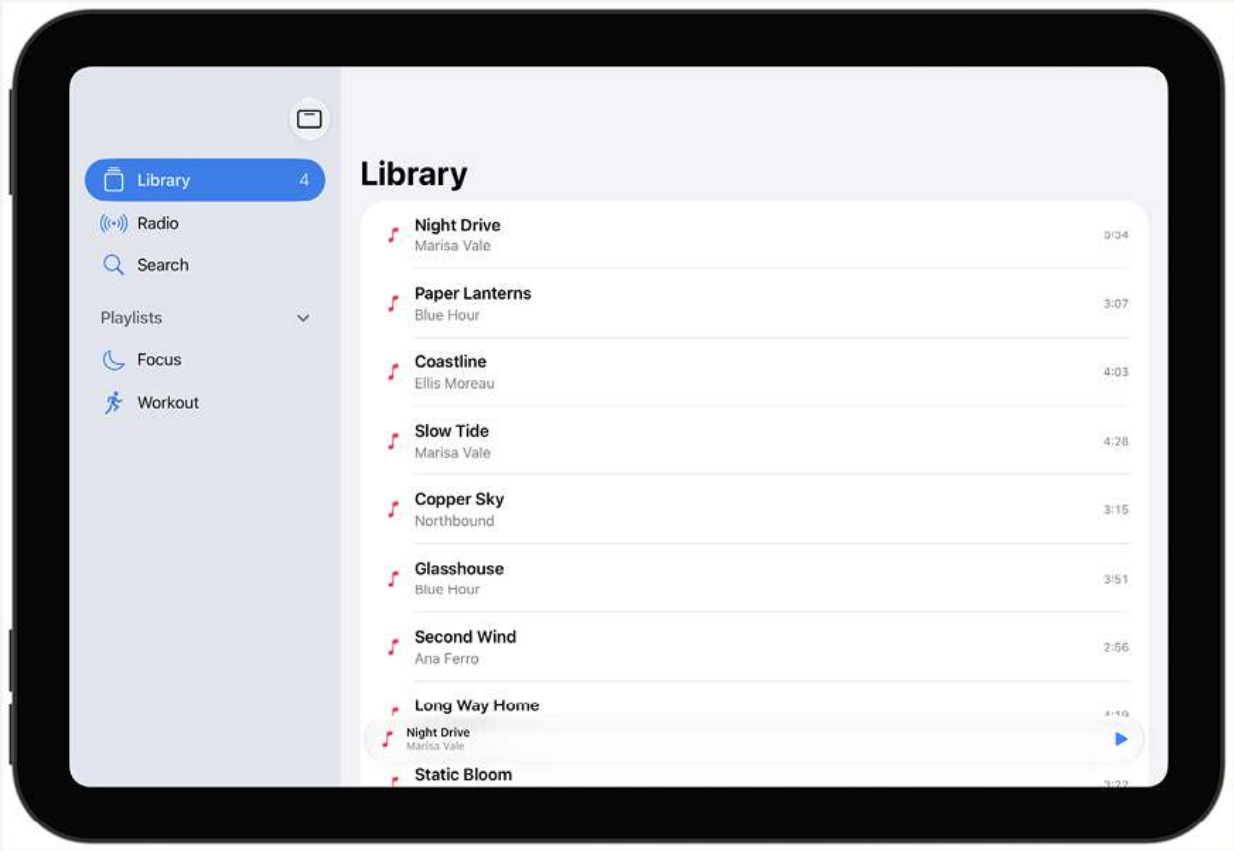


Figure 11.4: The iPad sidebar, with the Playlists section as a collapsible group

One structural rule is worth stating plainly, because getting it backward causes trouble that is hard to diagnose: a `TabView` contains `NavigationStack` containers, never the other way around. Each of our tabs owns its own stack, so pushing a detail in one destination leaves the others untouched and the tab bar stays where it is. We covered stacks and paths in the *NavigationStack and NavigationPath* section, and the toolbar side of the Liquid Glass design, including how a navigation bar decides what to keep when space runs short, in the *Toolbars and the Liquid Glass Desig n* section of [`Chapter 4`](#) .

See also

- `TabView` documentation: https://x.swiftuicookbook.com/2xe2v
- `Tab` documentation: https://x.swiftuicookbook.com/6g3bz
- `TabRole` documentation: https://x.swiftuicookbook.com/nlatl
- `TabSection` documentation: https://x.swiftuicookbook.com/0dcj3
- `tabViewBottomAccessory(content:)` documentation: https://x.swiftuicookbook.com/jcs9k
- `tabBarMinimizeBehavior(_:)` documentation: https://x.swiftuicookbook.com/3fzzs
- Build a SwiftUI app with the new design (WWDC 2025): https://x.sw iftuicookbook.com/wliha

Modal Sheets and Full-Screen Covers

Presenting a screen modally in UIKit meant picking a presentation style, configuring a transitioning delegate, and calling `present(_:animated:)` from a view controller that had to know about the one it was showing. SwiftUI inverts that: a presentation is a modifier on the view that triggers it, driven by state. We do not call a method to show a sheet, we flip a value and describe what should appear, and the framework handles the presentation and the dismissal.

In this recipe, we build a small contact list that reaches for four different presentations, because choosing the right one is most of the

skill. A `.sheet(item:)` opens a contact for editing, driven by the selected value rather than a flag. A `.sheet(isPresented:)` opens the same form empty for a new contact, guarded by `.interactiveDismissDisabled()` so a half-typed contact cannot be swiped away by accident. A `.fullScreenCover(isPresented:)` takes over the display for a task that should not be interrupted, and a `.popover()` shows a short explanation anchored to the button that asked for it. The presentations differ in how they are triggered and how much they commit the user, not in how they are written, which is the u seful thing to take away.

Getting ready

Create a new SwiftUI iOS app named **ContactManager** .

How to do it...

The app is a list of contacts with a toolbar, one form that serves both editing and adding contacts, and a placeholder screen for the full-screen cover. We will build the pieces first, then attach the presentations one at a time so it is clear which modifier produces which behavior. These are the steps:

1. Create `Contact.swift` in a `Models` folder. The model carries only what a row and a form need. Conforming to `Identifiable` is a requirement here rather than a convenience: the `.sheet(item:)` modifier we reach for in step 7 accepts only an `Identifiable` value, because it watches the `id` to decide what to do as the bound value changes, presenting a sheet when a contact appears, dismissing it when the value goes back to nil, and swapping the

content when a different contact takes its place. The `id` is generated on creation, so copying a contact and editing the copy keeps the same identity, which is how we will match an edited contact back to its row in a list. Two of the five samples are shown here:

```
import Foundation
struct Contact: Identifiable, Hashable {
  let id = UUID()
  var name: String
  var role: String
  var email: String
}
extension Contact {
  static let samples: [Contact] = [
    Contact(name: "Alice Johnson", role: "iOS Engineer",
            email: "alice@example.com"),
    Contact(name: "Bruno Costa", role: "Product Designer",
            email: "bruno@example.com"),
    // ... the remaining three contacts
  ]
}
```

For the complete initialization code of the `samples` static property, see the book's GitHub repository.

2. Create `ContactFormView.swift` in a `Views` folder. One form serves both jobs: it opens populated when we pass a contact, and empty when we do not. Holding the edits in a single `draft` copy of the whole contact, rather than in separate strings, means the `id` rides along and saving can find the original row. The `initial` copy is what we compare against to know whether anything changed:

```
import SwiftUI
struct ContactFormView: View {
  let title: String
  let onSave: (Contact) -> Void
  private let initial: Contact
  @Environment(\.dismiss) private var dismiss
  @State private var draft: Contact
  init(contact: Contact? = nil, onSave: @escaping (Contact)
    let start = contact ?? Contact(name: "", role: "", emai
    self.title = contact == nil ? "New Contact" : "Edit Cor
    self.onSave = onSave
    self.initial = start
    self.draft = start
  }
  private var hasChanges: Bool {
    draft.name != initial.name
      || draft.role != initial.role
      || draft.email != initial.email
  }
  var body: some View {
    NavigationStack {
      Form {
        TextField("Name", text: $draft.name)
        TextField("Role", text: $draft.role)
        TextField("Email", text: $draft.email)
          .keyboardType(.emailAddress)
          .textContentType(.emailAddress)
      }
      .navigationTitle(title)
      .navigationBarTitleDisplayMode(.inline)
    }
  }
}
#Preview {
  ContactFormView(contact: Contact.samples[0]) { _ in }
}
```

3. Give the form its toolbar. The dismiss button is always there, and the confirm button appears only once something has actually changed, so opening a contact just to read it shows a single close

control. Instead of a title, the dismiss button takes a role, `.cancel` while there are edits to throw away and `.close` otherwise. The system draws both as its standard X in this toolbar, and the role tells the system and assistive technologies what the button means. Add the `.toolbar` to the `Form`, after the title modifiers:

```
var body: some View {
  NavigationStack {
    Form {
      // ... the three text fields
    }
    .navigationTitle(title)
    .navigationBarTitleDisplayMode(.inline)
    .toolbar {
      ToolbarItem(placement: .cancellationAction) {
        Button(role: hasChanges ? .cancel : .close) {
          dismiss()
        }
      }
      if hasChanges {
        ToolbarItem(placement: .confirmationAction) {
          Button("Save", systemImage: "checkmark") {
            onSave(draft)
            dismiss()
          }
          .disabled(draft.name.isEmpty)
        }
      }
    }
  }
}
```

Opening a contact and changing nothing now shows a single close button, and the save control appears as soon as the first character changes.

4. Stop a half-written contact from being swiped away. `.interactiveDismissDisabled()` takes a condition, so the sheet resists the downward drag only while there are unsaved edits, and a form the user merely opened still closes with a swipe. Attach it to the `NavigationStack`, not to the `Form`, since it applies to the whole presented view:

```
  var body: some View {
    NavigationStack {
      // ... the Form, its title, and its toolbar
    }
    .interactiveDismissDisabled(hasChanges)
  }
```

5. Create `ImportView.swift` in the `Views` folder, the screen our full-screen cover will show. It stands in for a real import flow, so `ContentUnavailableView` gives us a centered icon, title, and message without any layout work. The close button matters more here than on a sheet: a full-screen cover has no drag-to-dismiss, so without a control the user would be stuck. The `.close` role supplies the system's X, so the button needs no title of its own:

```
import SwiftUI
struct ImportView: View {
  @Environment(\.dismiss) private var dismiss
  var body: some View {
    NavigationStack {
      ContentUnavailableView(
        "Import Contacts",
        systemImage: "square.and.arrow.down",
        description: Text("Bring in contacts from a vCard f
      )
      .navigationTitle("Import")
      .navigationBarTitleDisplayMode(.inline)
      .toolbar {
```

```
        ToolbarItem(placement: .cancellationAction) {
          Button(role: .close) {
            dismiss()
          }
        }
      }
    }
  }
}
#Preview {
  ImportView()
}
```

6. Open `ContentView.swift` and replace the `ContentView` struct with the list of contacts. Each row is a plain-styled `Button` that records which contact was tapped. The two modifiers under the `VStack` are what make the whole row respond, the same trick our track rows used in the previous recipe: the frame widens the label to the full row, and `.contentShape(.rect)` turns that widened area into a tappable rectangle:

```
struct ContentView: View {
  @State private var contacts = Contact.samples
  @State private var selectedContact: Contact?
  var body: some View {
    NavigationStack {
      List(contacts) { contact in
        Button {
          selectedContact = contact
        } label: {
          VStack(alignment: .leading, spacing: 2) {
            Text(contact.name)
              .font(.headline)
            Text(contact.role)
              .font(.subheadline)
              .foregroundStyle(.secondary)
          }
          .frame(maxWidth: .infinity, alignment: .leading)
```

```
            .contentShape(.rect)
          }
          .buttonStyle(.plain)
        }
        .navigationTitle("Contacts")
      }
    }
}
```

7. Turn a tapped contact into a sheet. `.sheet(item:)` watches an optional and presents whenever it becomes non-nil, handing the unwrapped value to its closure, so the selected contact both triggers the presentation and supplies its content. Saving writes the edited copy back over the row with the matching `id`:

```
struct ContentView: View {
  // ... the state properties
  var body: some View {
    NavigationStack {
      List(contacts) { contact in
        // ... the row button
      }
      .navigationTitle("Contacts")
      .sheet(item: $selectedContact) { contact in
        ContactFormView(contact: contact) { updated in
          if let i = contacts.firstIndex(where: { $0.id ==
            contacts[i] = updated
          }
        }
      }
    }
  }
}
```

With the preview canvas in live mode, tapping any part of a row opens that contact in the form. The binding carries values

in both directions: we write to `selectedContact` to present the sheet, and the modifier writes `nil` back into it once the presentation ends, whether the sheet was swiped away or a button called `dismiss()`. Nothing in our code has to clear the selection.

8. Add a contact with the same form and a different trigger. Here, there is no value to present, only the intent to show something, so a `Bool` drives it. Add the property and the toolbar button, and present the form with no contact so it opens empty:

```
struct ContentView: View {
  // ... the other state properties
  @State private var isAdding = false
  var body: some View {
    NavigationStack {
      // ... the List and its navigationTitle
      .toolbar {
        ToolbarItem(placement: .primaryAction) {
          Button("Add", systemImage: "plus") {
            isAdding = true
          }
        }
      }
      // ... the sheet(item:) modifier
      .sheet(isPresented: $isAdding) {
        ContactFormView { contacts.append($0) }
      }
    }
  }
}
```

Type a name in the new form and try to swipe it down: the sheet refuses, because the guard we added in step 4 is now doing its job.

9. Present a task that should not be interrupted. A full-screen cover takes the whole display and cannot be dismissed by dragging, which suits an import flow better than a sheet. It is driven by a `Bool`, exactly like the previous step, so only the modifier name changes:

```
struct ContentView: View {
  // ... the other state properties
  @State private var isImporting = false
  var body: some View {
    NavigationStack {
      // ... the List and its navigationTitle
      .toolbar {
        // ... the Add button
        ToolbarItem(placement: .topBarTrailing) {
          Button("Import", systemImage: "square.and.arrow.d
            isImporting = true
          }
        }
      }
      // ... the two sheet modifiers
      .fullScreenCover(isPresented: $isImporting) {
        ImportView()
      }
    }
  }
}
```

   Tap the import icon in the preview canvas: the cover takes the whole display, and only its close button ends it.

10. Finish with a popover, the lightest of the four modals. Unlike the others it attaches to the control that triggers it rather than to the container, so it stays anchored to that control. Under the Liquid Glass design a popover appears as a floating panel beside its anchor rather than the arrowed bubble of earlier releases, on

iPhone and iPad alike. `.presentationCompactAdaptation(.popover)` is what keeps it a popover on iPhone; without it the system adapts a compact-width popover into a sheet, and the anchoring that makes a popover worth using is lost:

```
struct ContentView: View {
  // ... the other state properties
  @State private var isShowingInfo = false
  var body: some View {
    NavigationStack {
      // ... the List and its navigationTitle
      .toolbar {
        ToolbarItem(placement: .topBarLeading) {
          Button("About", systemImage: "info.circle") {
            isShowingInfo = true
          }
          .popover(isPresented: $isShowingInfo) {
            Text("Tap a contact for details, or add one wit
              .font(.subheadline)
              .padding()
              .frame(width: 260)
              .presentationCompactAdaptation(.popover)
          }
        }
        // ... the Add and Import buttons
      }
      // ... the sheet and fullScreenCover modifiers
    }
  }
}
```

11. Exercise all four in the preview canvas using live mode: tap a row to edit a contact and save it, use the plus button to add one, open the import cover and close it, and tap the info icon for the popover. Notice how differently each one behaves as we try to

leave it, which is the whole point of choosing between them. For the complete `ContentView` code, see the book's GitHub repository.

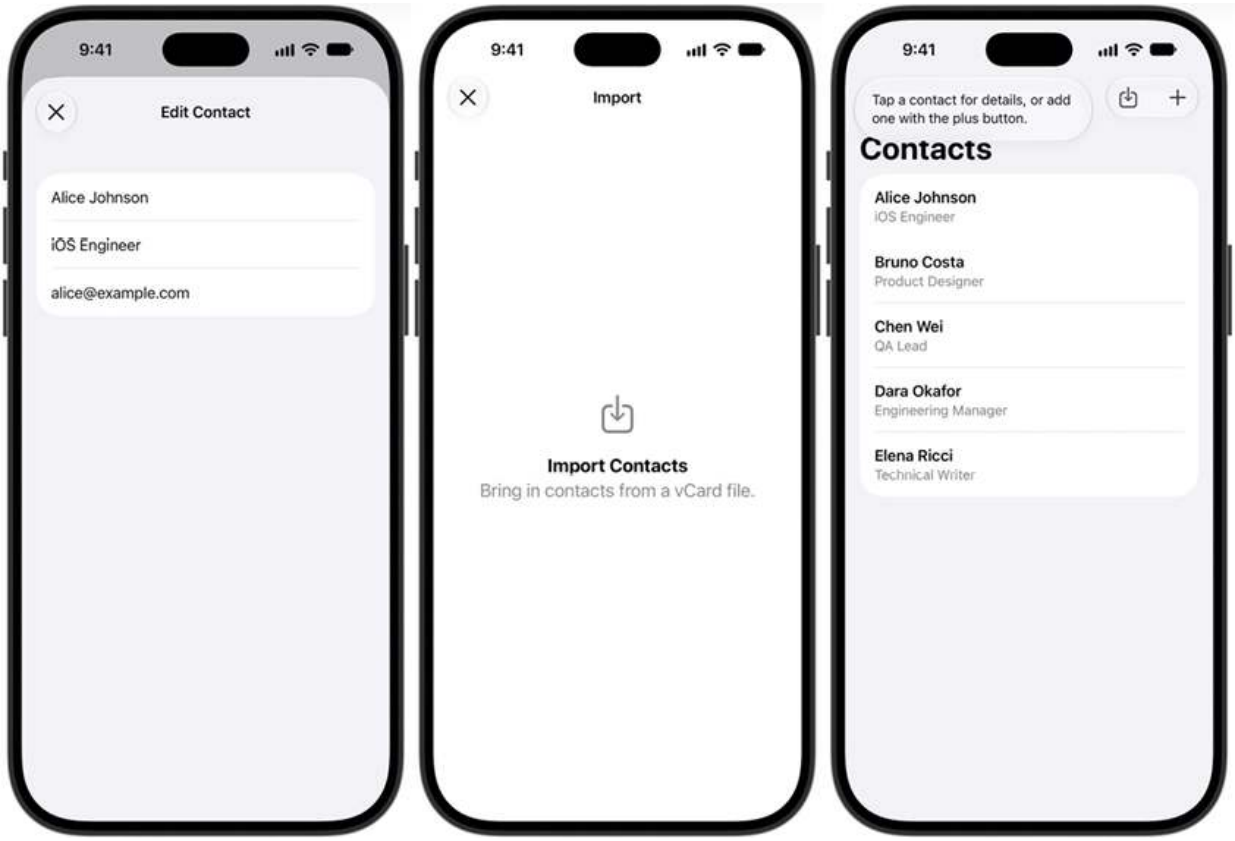


Figure 11.5: An edit sheet, the full-screen impor t cover, and an anchored popover

How it works...

The four modifiers share one idea: a presentation is a function of state, not a command we issue. We never tell a sheet to appear, we make a value true or non-nil and describe what should be built, and SwiftUI presents it, animates it, and writes the state back when the user dismisses it. That last part is why none of our dismiss buttons touch the flags they were presented by. Choosing between `isPresented:` and `item:` follows from what the presentation needs: a flag is enough when the content is always the same, while `item:` is the

right tool when the content depends on a value, because it makes presenting without that value impossible. Our edit sheet cannot open with a stale or missing contact, since the contact **is** the trigger.

The difference between a sheet, a cover, and a popover is about how much the presentation commits the user. A sheet leaves the parent visible behind it and can be swiped away, which suits reversible work. `.interactiveDismissDisabled(hasChanges)` narrows that escape hatch to exactly the moments it should be narrowed, and passing a condition rather than calling it bare is what keeps a read-only visit from being trapped. A full-screen cover removes the parent and the swipe together, so it needs its own close control and suits work that should not be half-finished. A popover is anchored, transient, and dismissed by tapping outside, which makes it right for explanation and wrong for input. On iPhone the system would rather show a sheet than a small anchored bubble, so `.presentationCompactAdaptation(.popover)` is what holds it to a popover.

Two details in the form are worth keeping. The draft is a copy of the whole `Contact`, so the `id` survives editing and `firstIndex(where:)` can find the row to overwrite; rebuilding a contact from three separate strings would mint a new `id` and silently append a duplicate instead. And `hasChanges` compares the draft against the value the sheet opened with, rather than asking whether the fields are non-empty, which is what lets the same form serve both jobs without trapping the user in an unchanged contact. We stay with default-sized sheets throughout this recipe; resizing them to a half-height detent, controlling the drag indicator, and letting the user interact with the content behind them are the subject of the *Dismissal and Presentation Customization* section, which also covers th e `dismiss` action our buttons call.

See also

- `sheet(isPresented:onDismiss:content:)` documentation: https://x.swiftuicookbook.com/1zned
- `sheet(item:onDismiss:content:)` documentation: https://x.swiftuicookbook.com/eao46
- `fullScreenCover(isPresented:onDismiss:content:)` documentation: https://x.swiftuicookbook.com/m5y4z
- `popover(isPresented:attachmentAnchor:arrowEdge:content:)` documentation: https://x.swiftuicookbook.com/1czxr
- `interactiveDismissDisabled(_:)` documentation: https://x.swiftuicookbook.com/uj66o
- `presentationCompactAdaptation(_:)` documentation: https://x.swiftuicookbook.com/dbyuc
- *Modal presentations* article: https://x.swiftuicookbook.com/ctfbb

Dismissal and Presentation Customization

A sheet used to be all or nothing. It covered the screen, the app behind it stopped, and the only way forward was to deal with it or dismiss it. The bottom drawer changed that: a panel that rests near the bottom of the display, can be dragged up when the user wants more, and leaves the app underneath alive the whole time. Maps popularized the pattern, and for a long time building it meant a custom container and a pan gesture recognizer. SwiftUI now assembles the whole thing from modifiers on an ordinary sheet.

In this recipe, we build a map of San Francisco landmarks with exactly that kind of drawer. Tapping a marker opens a panel resting at the bottom of the display, and `.presentationDetents()` gives it three heights the user can drag between. `.presentationBackgroundInteraction()` is what turns a sheet into a drawer, keeping the map pannable while the panel sits at its lower heights, and `.presentationContentInteraction()` decides whether a drag scrolls the panel's text or resizes the panel itself. Around those, `.presentationDragIndicator()` and `.presentationCornerRadius()` handle appearance, and the panel closes itself with `@Environment(\.dismiss)`, the action that ends whatever pr esentation a view happens to be in.

Getting ready

Create a new Swi ftUI iOS app named **LocationExplorer** .

How to do it...

The app is a map with four markers and one panel. We will build the model and the panel first, get the map showing markers, then turn the plain sheet it presents into a drawer one modifier at a time, so it is clear what each one contributes. These are the steps:

1. Create `Landmark.swift` in a `Models` folder. The model needs `import MapKit` rather than the template's `import Foundation`, because it exposes a `coordinate` the map can place a marker at. Latitude and longitude are stored as plain `Double` values and combined in a computed property, which keeps the type `Hashable` so it can drive the map's selection. The `facts` array gives the panel enough text to scroll. One of the four samples is shown here:

```
import MapKit
struct Landmark: Identifiable, Hashable {
  let id = UUID()
  let name: String
  let tagline: String
  let latitude: Double
  let longitude: Double
  let facts: [String]
  var coordinate: CLLocationCoordinate2D {
    CLLocationCoordinate2D(latitude: latitude, longitude: l
  }
}
extension Landmark {
  static let samples: [Landmark] = [
    Landmark(
      name: "Golden Gate Bridge",
      tagline: "Suspension bridge, opened 1937",
      latitude: 37.8199,
      longitude: -122.4783,
      facts: [
        "Towers rise 746 feet above the water",
        "Painted in International Orange for visibility in
        "The main span was the longest in the world until 1
        "Around 112,000 vehicles cross on a typical day"
      ]
    ),
    // ... the remaining three landmarks
  ]
}
```

For the complete initialization code of the `samples` static property, see the book's GitHub repository.

2. Create `LandmarkDetailView.swift` in a `Views` folder, the panel the map presents. It is a `ScrollView` on purpose: the facts need to overflow the drawer's smallest height so we have something to scroll later. Nothing in this view knows it will be shown in a sheet, and `dismiss` is what keeps it that way, because it ends

whichever presentation the view finds itself in. Its close button takes the `.close` role, so the system supplies the X:

```
import SwiftUI
struct LandmarkDetailView: View {
  let landmark: Landmark
  @Environment(\.dismiss) private var dismiss
  var body: some View {
    ScrollView {
      VStack(alignment: .leading, spacing: 16) {
        header
        ForEach(landmark.facts, id: \.self) { fact in
          Label(fact, systemImage: "info.circle")
            .font(.subheadline)
        }
      }
      .padding()
    }
  }
  private var header: some View {
    HStack(alignment: .top) {
      VStack(alignment: .leading, spacing: 4) {
        Text(landmark.name)
          .font(.title2)
          .fontWeight(.bold)
        Text(landmark.tagline)
          .font(.subheadline)
          .foregroundStyle(.secondary)
      }
      Spacer()
      Button(role: .close) {
        dismiss()
      }
      .buttonStyle(.bordered)
      .buttonBorderShape(.circle)
    }
  }
}
#Preview {
  LandmarkDetailView(landmark: Landmark.samples[0])
}
```

3. Open `ContentView.swift`. This file needs `import MapKit` beneath the template's `import SwiftUI`, so add it and then replace the `ContentView` struct with the map. A `Marker` per landmark gives us the standard balloon pin, and tagging each one with its landmark lets the map's `selection` binding hand us back the whole value rather than an identifier we would have to look up. The starting region is a `private static let` so it is created once rather than on every redraw:

```
import SwiftUI
import MapKit
struct ContentView: View {
  private let landmarks = Landmark.samples
  @State private var selected: Landmark?
  var body: some View {
    Map(initialPosition: .region(Self.bayArea), selection:
      ForEach(landmarks) { landmark in
        Marker(landmark.name, coordinate: landmark.coordina
          .tag(landmark)
      }
    }
  }
  private static let bayArea = MKCoordinateRegion(
    center: CLLocationCoordinate2D(latitude: 37.818, longit
    span: MKCoordinateSpan(latitudeDelta: 0.10, longitudeDe
  )
}
```

   In the preview canvas, the four markers appear over San Francisco, and tapping one enlarges it. Nothing else happens yet.

4. Present the panel. The obvious move is `.sheet(item: $selected)`, but we deliberately use `.sheet(isPresented:)` driven by a computed binding that reports whether anything is selected. The

reason matters and we come back to it in *How it works* : an item-driven sheet is torn down and rebuilt when the selected value changes, and the rebuilt presentation loses the detents we are about to add. Keeping one presentation alive and swapping only its content avoids that entirely. Add the binding above `body` , then the sheet:

```
struct ContentView: View {
  // ... the landmarks and selected properties
  private var isShowingDetail: Binding<Bool> {
    Binding(get: { selected != nil }, set: { if !$0 { selec
  }
  var body: some View {
    Map(initialPosition: .region(Self.bayArea), selection:
      // ... the ForEach of markers
    }
    .sheet(isPresented: isShowingDetail) {
      if let landmark = selected {
        LandmarkDetailView(landmark: landmark)
      }
    }
  }
  // ... the bayArea region
}
```

Tap a marker in the preview canvas and the panel appears. Swipe it away and the marker deselects, because the setter closes the loop: when the sheet dismisses, SwiftUI writes `false` and the binding clears `selected` .

5. Give the sheet its three heights. `.presentationDetents()` takes a set, the user drags between them, and the sheet opens at the smallest. A fixed `.height(140)` leaves the panel resting at the bottom like a

drawer, while `.medium` and `.large` are the system's half and full heights:

```
LandmarkDetailView(landmark: landmark)
  .presentationDetents([.height(140), .medium, .lar
```

Drag the panel by its top edge in the preview canvas and it snaps between the three heights.

6. Two appearance modifiers make it read as a drawer rather than a sheet that failed to open. The grabber tells the user the panel is draggable, and a larger corner radius softens the top edge:

```
LandmarkDetailView(landmark: landmark)
  .presentationDetents([.height(140), .medium, .lar
  .presentationDragIndicator(.visible)
  .presentationCornerRadius(28)
```

7. Now, the modifier that actually makes it a drawer. By default, a sheet blocks the view behind it, so our map would be frozen while the panel is open. `.presentationBackgroundInteraction()` lifts that block, and passing `.enabled(upThrough: .medium)` scopes it: the map stays pannable while the panel rests at its two lower heights, and goes inert once the user drags the panel to `.large` and clearly wants to read rather than browse:

```
LandmarkDetailView(landmark: landmark)
  .presentationDetents([.height(140), .medium, .lar
  .presentationDragIndicator(.visible)
  .presentationCornerRadius(28)
  .presentationBackgroundInteraction(.enabled(upTh
```

With the panel at its smallest height, pan and zoom the map behind it. Drag the panel up to full height and the map stops responding.

8. Decide what a drag inside the panel does. With scrollable content in a resizable sheet, one gesture has two plausible meanings, and the system has to pick. `.scrolls` says the content wins, so the facts scroll under the user's finger instead of hauling the whole panel up:

```
        LandmarkDetailView(landmark: landmark)
          .presentationDetents([.height(140), .medium, .lar
          .presentationDragIndicator(.visible)
          .presentationCornerRadius(28)
          .presentationBackgroundInteraction(.enabled(upThr
          .presentationContentInteraction(.scrolls)
```

   The grabber still resizes the panel, so both interactions remain available.

9. Use the preview canvas in live mode to exercise the finished drawer. Tap a marker to open it, pan the map behind it, and scroll the facts, then drag the panel up to full height and confirm the map stops responding. Tap another marker while the drawer is open and the content swaps without the panel jumping to a different height. For the complete `ContentView` code, see the book's GitHub repository.

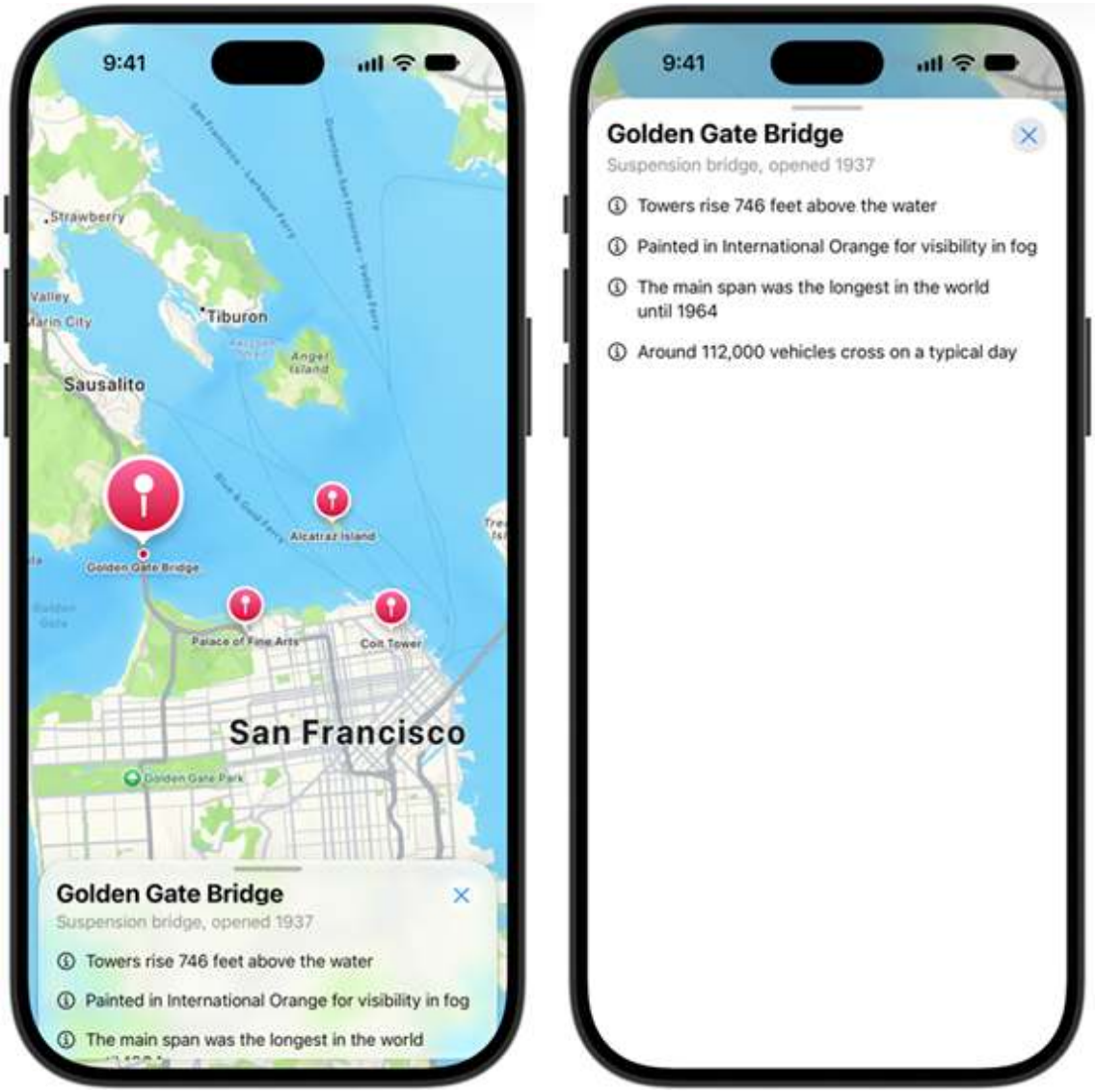


Figure 11.6: The drawer resting at its smallest deten t over a live map, and expanded to read

How it works…

The drawer is one ordinary sheet with five modifiers on it, and they divide into two groups. `.presentationDragIndicator()` and `.presentationCornerRadius()` are appearance: the grabber and the top corners. The other three change behavior, and they are the ones worth understanding. `.presentationDetents()` supplies the heights, and

the set can mix `.medium` and `.large` with a fixed `.height()` or a proportional `.fraction()`; the sheet opens at the smallest and the user drags between the rest. There is also a `presentationDetents(_:selection:)` overload that binds the current detent, which lets code read or move the sheet between heights. One more appearance modifier, `.presentationSizing()`, sets the sheet's footprint on wide displays, where `.page` is noticeably wider than the default `.form`. On iPhone the sheet already spans the width, so the drawer does without it.

`.presentationBackgroundInteraction()` is what separates a drawer from a sheet. A sheet is modal by definition, so the view behind it stops accepting touches; passing `.enabled(upThrough: .medium)` narrows that modality to the heights where it makes sense. The user gets a map they can still explore while a panel of detail rests at the bottom, and full modality returns the moment they pull the panel up. The `upThrough:` form is worth preferring over a bare `.enabled`, because a full-height sheet with a live view behind it invites taps the user cannot see the results of. `.presentationContentInteraction()` resolves the ambiguity that detents create: a drag inside a scrollable sheet could scroll the content or resize the sheet, and `.scrolls` puts the content first while leaving the grabber as the deliberate way to resize.

Our sheet is presented with `.sheet(isPresented:)` and a computed binding rather than the `.sheet(item:)` that this app's shape suggests, and the reason is a trap worth remembering. When an item-driven sheet's value changes while the sheet is already on screen, SwiftUI ends that presentation and starts a new one, and the new presentation does not inherit the detents. The symptom is a drawer that behaves correctly until the user taps a second marker without closing the first, at which point it opens at full height. Driving the

sheet from a `Bool` keeps a single presentation alive for as long as anything is selected, so changing `selected` only rebuilds the content inside it. The computed binding reads `selected != nil` and, on dismissal, writes `nil` back, which is what deselects the marker.

`dismiss` deserves its own note, because it is not a sheet API at all. `@Environment(\.dismiss)` gives a view a `DismissAction` that ends whatever presentation encloses it: a sheet, a full-screen cover, a popover, or a `NavigationStack` push, where it pops instead. That is why `LandmarkDetailView` contains no flag, no binding, and no knowledge of the drawer it lives in, and why the same view would work unchanged if we pushed it onto a stack. Presenting it is `ContentView`'s business; ending the presentation is the framework's. We used the sheet, cover, and popover presentations themselves in the *Mod al Sheets and Full-Screen Covers* section.

See also

- `presentationDetents(_:)` documentation: https://x.swiftuicookbook.com/v2531
- `PresentationDetent` documentation: https://x.swiftuicookbook.com/ogu53
- `presentationBackgroundInteraction(_:)` documentation: https://x.swiftuicookbook.com/2alpo
- `presentationContentInteraction(_:)` documentation: https://x.swiftuicookbook.com/efvho
- `presentationSizing(_:)` documentation: https://x.swiftuicookbook.com/190bg

- `DismissAction` documenta tion: https://x.swiftuicookbook.com/9gp6i

Deep Linking and URL Handling

Everything in this chapter so far has moved the user around inside our app. Deep linking is the opposite direction: something outside the app, a web page, a message, a notification, or another app entirely, names a screen and asks us to show it. The awkward part has never been receiving the URL, it is that the app may be launching from scratch, so the screen has to be reachable from a standing start rather than from wherever the user happened to be. That is exactly what the data-driven navigation from the first recipe in this chapter is good at: a URL becomes a value, and the value becomes a path.

In this recipe, we build a small product catalog that can be entered from outside. We register a `shoplinks://` URL scheme so the system knows our app owns those links, and `.onOpenURL()` turns an incoming `shoplinks://product/3` into the product it names and pushes it. Travel goes the other way too: `@Environment(\.openURL)` hands a web address to Safari, and `OpenURLAction` lets us intercept a link before it leaves, so a web link that names one of our own products is handled in place instead of bouncing the user out to Safari.

Getting ready

Create a new SwiftUI iOS app named **ShopLinks** . Testing a deep link needs Device Hub, because handing a URL from one app to another is precisely what the pr eview canvas does not do, even in live mode.

How to do it...

The app is a list of products and a detail screen. We will build the catalog first, register the URL scheme, then wire up the three directions a URL can travel: in, out, and intercepted. These are the steps:

1. Create `Product.swift` in a `Models` folder. Beyond the fields the list and the detail screen show, the model carries the URLs this recipe needs in both directions: a `webURL` pointing at the product's page on the web, a `storeURL` for the store's home page, and a `from(_:)` helper that turns an incoming URL back into a product. That helper accepts two spellings of a product address, our own `shoplinks://product/3` and the web page `https://www.example.com/product/3`, and is deliberately strict about both: it returns nil unless the scheme, the host, and a numeric identifier all check out, because it is the front door for input we do not control. The `related` property picks the next product in the catalog, wrapping at the end, so the detail screen always has somewhere to link to. Two of the five products are shown here:

```
import Foundation
struct Product: Identifiable, Hashable {
  let id: Int
  let name: String
  let tagline: String
  let price: Decimal
  var priceText: String {
    price.formatted(.currency(code: "USD"))
  }
  var webURL: URL {
    URL(string: "https://www.example.com/product/\(id)")!
  }
}
```

```
extension Product {
  static let all: [Product] = [
    Product(id: 1, name: "Field Notebook",
            tagline: "Pocket sized, 96 pages", price: 12),
    Product(id: 2, name: "Fountain Pen",
            tagline: "Medium nib, converter included", pric
    // ... the remaining three products
  ]
  static let storeURL = URL(string: "https://www.example.cc
  static func withID(_ id: Int) -> Product? {
    all.first { $0.id == id }
  }
  var related: Product {
    Product.withID(id % Product.all.count + 1) ?? Product.a
  }
  static func from(_ url: URL) -> Product? {
    let isDeepLink = url.scheme == "shoplinks" && url.host(
    let isWebLink = url.scheme == "https" && url.host() ==
      && url.pathComponents.dropFirst().first == "product"
    guard isDeepLink || isWebLink,
          let id = Int(url.lastPathComponent) else { returr
    return withID(id)
  }
}
```

For the complete initialization code of the `all` static property, see the book's GitHub repository.

2. Register the URL scheme so the system routes `shoplinks://` links to our app. Select the project, the blue icon at the very top of the Project navigator, choose the **ShopLinks** target, open the **Info** tab, expand **URL Types** , and click the plus button. Set **Identifier** to `com.SwiftUICookbook.ShopLinks` and **URL Schemes** to `shoplinks` , leaving the role as **Editor** . Xcode adds an `Info.plist` file to the project the first time we do this, because the settings it generates on its own have nowhere to put a structured value like this one.

Beyond these settings there is no code to write in this step, but every link in the rest of the recipe depends on them.

3. Create `ProductDetailView.swift` in a `Views` folder. The `openURL` environment action lives here. It sends the store's web address out to whichever app handles it, normally Safari, and it is called like a function without needing to know who will answer. The `Link` below it points at the related product's own web page, which is the link step 8 will intercept:

```
import SwiftUI
struct ProductDetailView: View {
  let product: Product
  @Environment(\.openURL) private var openURL
  var body: some View {
    VStack(alignment: .leading, spacing: 16) {
      Text(product.tagline)
        .font(.headline)
        .foregroundStyle(.secondary)
      Text(product.priceText)
        .font(.largeTitle)
        .fontWeight(.bold)
      Button("Visit the store", systemImage: "safari") {
        openURL(Product.storeURL)
      }
      .buttonStyle(.borderedProminent)
      Link("See the \(product.related.name)",
           destination: product.related.webURL)
        .font(.callout)
      Spacer()
    }
    .padding()
    .navigationTitle(product.name)
    .navigationBarTitleDisplayMode(.inline)
  }
}
#Preview {
  NavigationStack {
    ProductDetailView(product: Product.all[2])
```

```
  }
}
```

4. Open `ContentView.swift` and replace the `ContentView` struct with the catalog. The navigation is deliberately path-driven rather than using inline destinations, because a deep link has to be able to set the whole path in one assignment, without a link on screen to tap:

```
struct ContentView: View {
  private let products = Product.all
  @State private var path: [Product] = []
  var body: some View {
    NavigationStack(path: $path) {
      List(products) { product in
        NavigationLink(value: product) {
          VStack(alignment: .leading, spacing: 2) {
            Text(product.name)
              .font(.headline)
            Text(product.priceText)
              .font(.subheadline)
              .foregroundStyle(.secondary)
          }
        }
      }
      .navigationTitle("ShopLinks")
      .navigationDestination(for: Product.self) { product i
        ProductDetailView(product: product)
      }
    }
  }
}
```

5. Receive the link. `.onOpenURL()` fires whenever the system hands our app a URL, whether the app was already running or has just launched for it. Assigning to `path` is all the navigation we need,

because the stack rebuilds itself from that value. Attach it to the `NavigationStack`, not to anything inside it, so it is listening no matter where the user had navigated to:

```
struct ContentView: View {
  // ... the products and path properties
  var body: some View {
    NavigationStack(path: $path) {
      // ... the List and its navigationDestination
    }
    .onOpenURL { url in
      if let product = Product.from(url) {
        path = [product]
      }
    }
  }
}
```

Note what happens to a URL we do not recognize: `from(_:)` returns nil, we leave `path` alone, and the app simply opens where it was. A deep link that quietly does nothing is far better than one that crashes or lands the user on the wrong screen.

6. Test it. Run the app in Device Hub, then press *Cmd* + *Shift* + *H* to go to the home screen and open Safari. Type `shoplinks://product/3` into the address bar and press Return, the key labeled **Go** on the on-screen keyboard. Safari asks whether to open the page in **ShopLinks**; confirm, and the app comes to the front on the Desk Lamp detail with a **Safari** back link in the status bar. Try `shoplinks://product/99` as well, which names a product that does not exist, and confirm the app comes back to the front exactly where it was, still on the Desk Lamp, because an unrecognized link changes nothing.



The confirmation Safari shows is the system asking permission to switch apps, not a sign that something went wrong. It appears for any custom scheme, whether the target app is already running or has to launch. Universal links, which we come back to in *How it works* , do not show it.

7. Now, send a URL the other way. There is no code to add here, only something to try: with the app running in Device Hub, tap **Field Notebook** in the list to open its detail screen, then tap **Visit the store** . Safari opens the store's page on `example.com` , and returning to the app leaves us exactly where we were. That button calls `openURL(Product.storeURL)` , which hands the address to the system rather than opening anything by our app. The asymmetry is worth a pause: receiving a URL is a modifier on our view, while sending one is an action we call.
8. Before adding anything, see the problem for ourselves. On that same detail screen, tap the **See the Fountain Pen** link at the bottom. It points at `https://www.example.com/product/2` , the web page of a product we could show ourselves, but a `Link` hands its destination to the system, so Safari opens the page and the user leaves the app for a screen we already have. `OpenURLAction` lets us make that decision instead: return `.handled` when we dealt with it ourselves, or `.systemAction` to let the system carry on as normal. Add the custom action to the `NavigationStack` in `ContentView` , below `.onOpenURL()` :

```
        .onOpenURL { url in
          // ... the existing handler
```

```
        }
        .environment(\.openURL, OpenURLAction { url in
          if let product = Product.from(url) {
            path = [product]
            return .handled
          }
          return .systemAction
        })
```

Run the app in Device Hub again and repeat the same tap. This time the app moves straight to the Fountain Pen without leaving, because every `openURL` call from a view inside this stack now passes through our handler first, and a `Link` opens its destination the same way. Anything that is not one of our product pages, including the store address from the previous step, still returns `.systemAction` and carries on to Safari untouched.

9. Try the whole loop once more, in the order a real user would meet it: open a `shoplinks://` link from Safari, follow the related-product link to move between products without leaving, then send yourself out to the web and come back. For the complete `ContentView` code, see the book's GitHub repository.

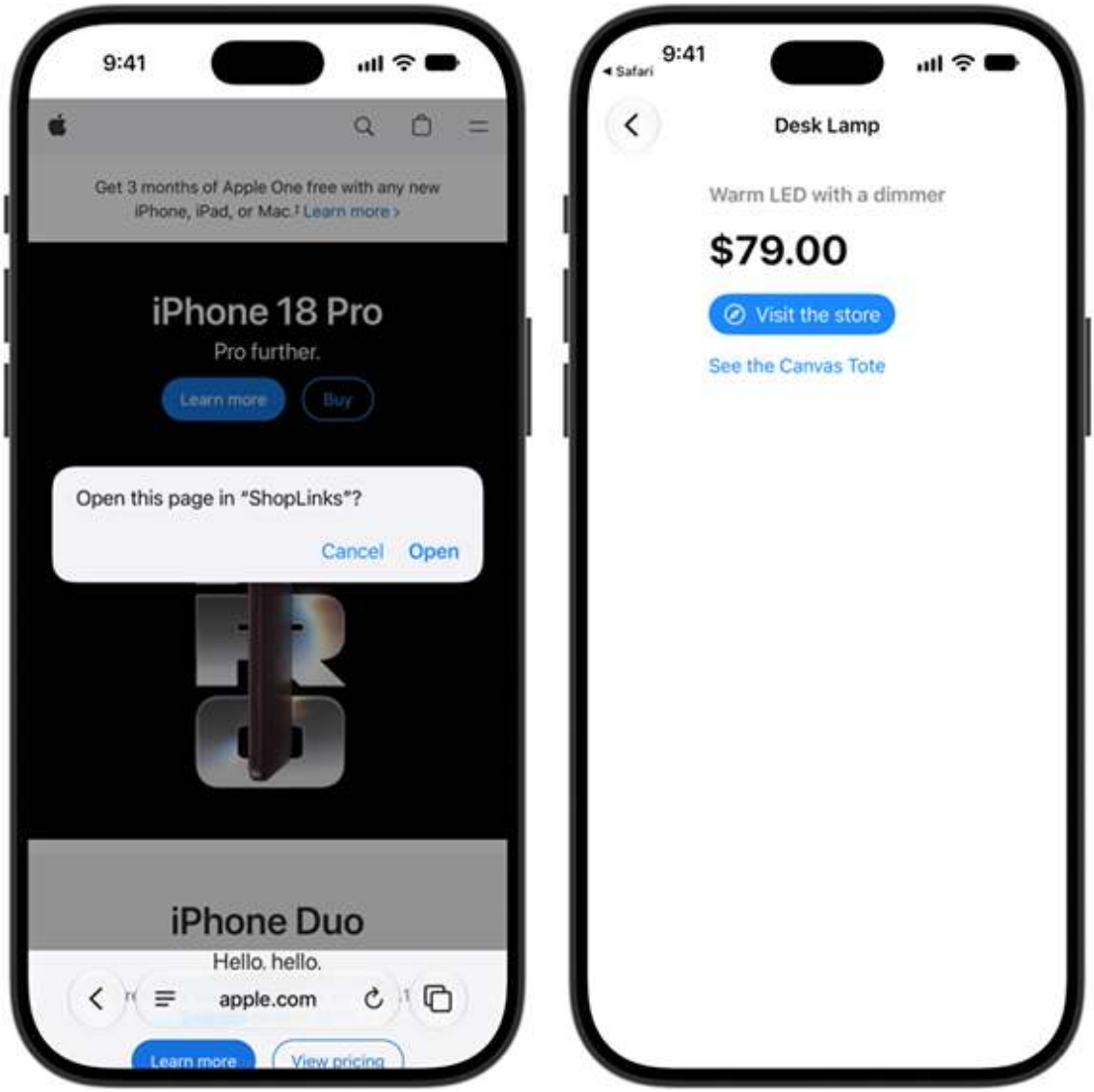


Figure 11.7: Safari handing a shoplinks address to the app, which opens on that product

How it works…

A deep link is a small parsing problem attached to a navigation problem, and the parsing half deserves more suspicion than it usually gets. `Product.from(_:)` accepts exactly two shapes of address, our scheme and our web page, checks the scheme, the host, and that the last path component is a number that matches a real product, and

returns nil at the first thing that fails. That strictness matters because a URL is input from outside the app: anyone can type `shoplinks://product/99999` or `shoplinks://admin/1`. Returning nil and doing nothing is the right answer to all of it. The navigation half is then almost trivial, and only because of the groundwork in the *NavigationStack and NavigationPath* section: `path = [product]` reconstructs the whole stack in one assignment, which works identically whether the app was already open on another product or has just launched from nothing.

URLs travel in three directions here, and it is worth separating them. `.onOpenURL()` is arrival, a modifier that listens for the system handing us a URL. `openURL` is departure, an action we call to hand a URL to whoever handles it, which keeps us out of the business of knowing that `https` means Safari. `OpenURLAction` sits between the two: installing one into the environment makes every `openURL` call from the views inside consult us first, and returning `.handled` or `.systemAction` decides whether the URL leaves. That is what lets a web link that names one of our products navigate in place while the store's own address still goes to the browser. Because it is an environment value, its reach is exactly the subtree it is attached to, so a different part of the app can install a different policy or none at all.

Custom schemes like ours are the quickest route to a working deep link, and they have a real limitation: any app can claim `shoplinks://`, so the system cannot know which one is genuine, which is why iOS asks the user to confirm before switching. **Universal links** solve that with ordinary `https` addresses. The app declares an Associated Domains entitlement, the website serves an `apple-app-site-association` file listing the paths the app handles, and iOS verifies the pairing by

fetching that file. Because the association is proven rather than claimed, a universal link opens the app with no confirmation, and falls back to the website when the app is not installed. The receiving code is already in place, which is the encouraging part: universal links arrive through `.onOpenURL()` too, and `from(_:)` already accepts the `https` form of a product address, so a verified link to `www.example.com/product/3` would land on the same screen. What they add is setup rather than code, a paid developer account for th e entitlement and a file on a server you control.

See also

- `onOpenURL(perform:)` documentation: https://x.swiftuicookbook.com/ats7o
- `OpenURLAction` documentation: https://x.swiftuicookbook.com/mi2eu
- *Defining a custom URL scheme for your app* article: https://x.swiftuicookbook.com/u1ozi
- *Supporting universal links in your app* article: https://x.swiftuicookbook.com/4pvx7

12

Drawing, Animations, and Visual Effects

I still have muscle memory for `draw(_:)`. In UIKit, anything the stock controls could not draw meant subclassing a view, overriding that method, and issuing Core Graphics calls against a context we were handed and had to give back in the state we found it. Animation lived somewhere else entirely, in a layer tree with its own timing model, and materials were a third thing again. What strikes me about SwiftUI is not that it made any one of those easier, but that it stopped treating them as separate subjects: a shape is a view, an animation is something that happens to a value, and a sheet of glass is a modifier. Once they stop being separate systems, combining them stops being an integration problem and starts being a design decision.

We start with a summary card for a single hike. Built-in shapes such as `Rectangle`, `Circle`, `Capsule`, `RoundedRectangle`, and the per-corner `UnevenRoundedRectangle` with its `RectangleCornerRadii` sit next to one we write ourselves with `Path` and the `Shape` protocol. Gradients and `.fill()` color them, `.stroke()` and `.strokeBorder()` with a `StrokeStyle` show why a centered stroke and an inset border are not the same line, and `GeometryReader` supplies the space to draw into. A star chart

comes next, drawing the night sky as it looks this evening. Four hundred stars, the Big Dipper, and three planets are rendered by a single `Canvas` through its `GraphicsContext`, which resolves symbols and images into the scene, and `TimelineView` redraws it on a schedule so the sky keeps moving. Then a gallery wall, where a tapped tile grows into a card: `@Namespace` and `.matchedGeometryEffect()` pair the two ends of that hero animation, `withAnimation` and `.animation(_:value:)` decide who drives it, `.transaction()` holds one element out of the shared timing, and a picker re-runs the whole app through `.easeInOut()`, `.snappy()`, `.bouncy()`, and `.interpolatingSpring()` so four curves can be compared by feel rather than read about. A curator's note unfolds through `.transition()`, `AnyTransition`, `.asymmetric()`, and a `Transition` of our own.

A pour-over coffee timer then splits multi-stage motion in two. `PhaseAnimator`, reached through the `.phaseAnimator()` modifier, walks a badge through poses that each carry their own curve, while `KeyframeAnimator` interpolates a falling drop along three independent `KeyframeTrack` timelines assembled from `MoveKeyframe`, `LinearKeyframe`, `CubicKeyframe`, and `SpringKeyframe`. A `KeyframeTimeline` then hands that same choreography to a slider so it can be stepped through by hand, and the ring around the cup is a custom shape whose two animated properties are wired up by the `@Animatable` macro, with `@AnimatableIgnored` marking the one that has no meaningful midpoint. The chapter closes on an artwork viewer whose entire interface floats over the picture. `.backgroundExtensionEffect()` carries the artwork up under the navigation bar, `.glassEffect()` inside a `GlassEffectContainer` gives the controls their material, `.glassEffectUnion()` fuses two of them into a single shape while

`.glassEffectID()` morphs one into three, and `Glass.tint(_:)` and `Glass.interactive(_:)` decide how they take color and answer a press, with a `.regularMaterial` caption card left in place so the older treatment and Liquid Glass can be judged side by side.

By the end of this chapter, you should be comfortable drawing whatever the built-in views cannot, choosing between a retained `Shape` and an immediate-mode `Canvas` for a given scene, and reaching for the right animation tool rather than the first one that comes to mind, whether the motion is a single value change, a chain of poses, or a timeline authored frame by frame.

We will cover these topics in the following recipes:

- Shapes, Paths, and Drawing
- Canvas and Custom Rendering
- Animations, Transitions, and Matched Geometry
- Keyframe and Phase Animations
- Glass and Material Effects

Technical requirements

The code in this chapter is based on Xcode 27.0 and iOS 27.0. You can download and install the latest version of Xcode from the App Store. All the code examples for this chapter can be found on GitHub at https://x.swiftuicookbook.com/ch12 .

Shapes, Paths, and Drawing

Custom drawing in UIKit meant subclassing `UIView`, overriding `draw(_:)`, and issuing commands into a Core Graphics context that existed only for the length of that call. Anything that changed the drawing had to remember to ask for a redraw. SwiftUI turns that around: a shape is a value that describes an outline, the framework decides when to render it, and the same description that draws a line on a chart can just as easily clip an image or mask a gradient.

In this recipe, we build a summary card for a single hike. One card turns out to be enough to need the whole drawing toolkit, and keeping the shapes together in a single composition is what makes their differences concrete: the choice between a centered stroke and an inset border, or between a built-in shape and one we write ourselves, is far easier to judge when both are on screen at the same time. The card is built from two stacked surfaces, a gradient header sitting directly on a white body, and each one rounds only the two corners along its own outer edge so the pair meets flush across the middle. We get that from `UnevenRoundedRectangle` and its `RectangleCornerRadii`, which let a shape round some corners and leave the others square. Two concentric progress rings come from a custom `Shape` built on a single `Path` arc, laid over track rings drawn with `strokeBorder()`. An elevation profile turns an array of altitudes into a `Path` of straight segments inside a `GeometryReader`, stroked as a line, filled underneath as an area, and standing on a `Rectangle` baseline inside a `RoundedRectangle` well. A row of `Capsule` chips closes the card, one of them outlined with a dashed `StrokeStyle` instead of filled, which is the conventional way to show a statistic with no value to dis play.

Getting ready

Create a new SwiftUI iOS app named **Trail Stats** .

How to do it...

We start with the data, then the custom shape the rings are made of, then the three pieces of the card, and finish with the view that assembles them. Each view we create has its own preview, so a geometry mistake surfaces while we are still looking at the piece that caused it, rather than after the whole card is assembled. These are the steps:

1. Create `Hike.swift` in a `Models` folder. The model carries the numbers the card draws with, and two pairs of them are goals rather than measurements, because a progress ring needs something to be a fraction of. The `elevationSamples` array is the raw material for the profile, and `distanceText` and `elevationText` build their display strings here rather than in the view, so the formatting lives with the data:

```
import Foundation
struct Hike {
  let name: String
  let location: String
  let distance: Double
  let distanceGoal: Double
  let elevationGain: Double
  let elevationGoal: Double
  let difficulty: String
  let surface: String
  let elevationSamples: [Double]
  var distanceProgress: Double {
    min(distance / distanceGoal, 1)
```

```
  }
  var elevationProgress: Double {
    min(elevationGain / elevationGoal, 1)
  }
  var distanceText: String {
    "\(distance.formatted()) / \(distanceGoal.formatted())
  }
  var elevationText: String {
    "\(elevationGain.formatted()) / \(elevationGoal.format
  }
}
```

2. Add a sample hike in the same file, so every preview in the recipe has something to draw. The eleven elevation samples climb from 820 to 2,970 feet and come back down, so the summit sits exactly 2,150 feet above the trailhead, which is what the ring reports:

```
extension Hike {
  static let sample = Hike(
    name: "Eagle Ridge Loop",
    location: "Cascade Range, Washington",
    distance: 8.4,
    distanceGoal: 10,
    elevationGain: 2150,
    elevationGoal: 3000,
    difficulty: "Strenuous",
    surface: "Rocky",
    elevationSamples: [
      820, 1140, 1080, 1560, 2010, 2480, 2970, 2640, 2180,
    ]
  )
}
```

3. Create `ProgressArc.swift` in a `Views` folder, our first custom shape. Conforming to `Shape` means implementing one method,

`path(in:)`, which receives the rectangle the shape has been given and returns the outline to draw inside it. Everything is expressed relative to that rectangle, never in absolute coordinates, which is what lets the same arc serve both of our rings at different sizes. Starting the sweep at -90 degrees puts zero progress at the top of the circle instead of at three o'clock:

```
import SwiftUI
struct ProgressArc: Shape {
  var progress: Double
  func path(in rect: CGRect) -> Path {
    let startAngle = Angle.degrees(-90)
    return Path { path in
      path.addArc(
        center: CGPoint(x: rect.midX, y: rect.midY),
        radius: min(rect.width, rect.height) / 2,
        startAngle: startAngle,
        endAngle: startAngle + .degrees(360 * progress),
        clockwise: false
      )
    }
  }
}
#Preview {
  ProgressArc(progress: 0.7)
    .stroke(.teal, style: StrokeStyle(lineWidth: 16, lineC
    .padding(8)
    .frame(width: 144, height: 144)
}
```

4. Create `HikeHeaderView.swift` in the `Views` folder. This is the top half of the card, and it is where `UnevenRoundedRectangle` earns its place: the header rounds only its top two corners, because the white surface underneath it will round the bottom two. `RectangleCornerRadii` names the corners we want and leaves the

rest at zero. Passing the shape to `background(_:in:)` fills it with the gradient in one step, and the `maxWidth` frame is what stretches the gradient across the card instead of letting it hug the text:

```
import SwiftUI
struct HikeHeaderView: View {
  let hike: Hike
  private let headerGradient = LinearGradient(
    colors: [.teal, .green],
    startPoint: .topLeading,
    endPoint: .bottomTrailing
  )
  var body: some View {
    VStack(alignment: .leading, spacing: 4) {
      Text(hike.name)
        .font(.title2.bold())
      Text(hike.location)
        .font(.subheadline)
        .foregroundStyle(.white.opacity(0.85))
    }
    .frame(maxWidth: .infinity, alignment: .leading)
    .foregroundStyle(.white)
    .padding(20)
    .background(
      headerGradient,
      in: UnevenRoundedRectangle(
        cornerRadii: RectangleCornerRadii(topLeading: 24,
      )
    )
  }
}
#Preview {
  HikeHeaderView(hike: .sample)
    .padding()
}
```

5. Create `ProgressRingsView.swift` in the `Views` folder for the two concentric rings. The geometry is the interesting part: rather than hard-coding a diameter that happens to work, we state the ring thickness, the gap between rings, and the size of the hole in the middle, and let `ringDiameter` work out the rest. Each ring is a track drawn with `strokeBorder()` and a `ProgressArc` on top of it, and the `level` argument insets the inner ring by one ring plus one gap:

```
import SwiftUI
struct ProgressRingsView: View {
  let hike: Hike
  private let lineWidth: Double = 16
  private let ringSpacing: Double = 8
  private let holeDiameter: Double = 64
  private var ringDiameter: Double {
    holeDiameter + 2 * (2 * lineWidth + ringSpacing)
  }
  private var ringStyle: StrokeStyle {
    StrokeStyle(lineWidth: lineWidth, lineCap: .round)
  }
  var body: some View {
    ZStack {
      ring(progress: hike.distanceProgress, color: .teal,
      ring(progress: hike.elevationProgress, color: .orang
    }
    .frame(width: ringDiameter, height: ringDiameter)
  }
  private func ring(progress: Double, color: Color, level:
    ZStack {
      Circle()
        .strokeBorder(color.opacity(0.2), lineWidth: lineW
      ProgressArc(progress: progress)
        .stroke(color.gradient, style: ringStyle)
        .padding(lineWidth / 2)
    }
    .padding(Double(level) * (lineWidth + ringSpacing))
  }
```

```
}
#Preview {
  ProgressRingsView(hike: .sample)
    .padding()
}
```

6. Rings on their own do not say what they are counting, so add a legend beside them. In `body`, wrap the `ZStack` and its `.frame()` modifier in an `HStack`, and give it a column of labels for company. Each label row pairs a small filled `Circle` with the title and the formatted value the model already prepared, and the trailing `Spacer()` keeps the whole group pinned to the leading edge when the card is wider than the group itself:

```
struct ProgressRingsView: View {
  // ... the hike property and the ring measurements
  var body: some View {
    HStack(spacing: 20) {
      ZStack {
        // ... the two ring calls
      }
      .frame(width: ringDiameter, height: ringDiameter)
      VStack(alignment: .leading, spacing: 16) {
        legend(color: .teal, title: "Distance", value: hik
        legend(color: .orange, title: "Elevation", value:
      }
      Spacer()
    }
  }
  // ... the ring(progress:color:level:) method
  private func legend(color: Color, title: String, value:
    HStack {
      Circle()
        .fill(color)
        .frame(width: 12, height: 12)
      VStack(alignment: .leading) {
        Text(title)
```

```
            .font(.caption)
            .foregroundStyle(.secondary)
          Text(value)
            .font(.subheadline.weight(.medium))
        }
      }
    }
  }
```

7. Create `ElevationProfileView.swift` in the `Views` folder. This is the first shape whose outline depends on how much room it is given, so it needs a `GeometryReader`, a container that hands its content the size it has to work with. `profilePoints(in:)` turns each altitude into a point by scaling it between the lowest and highest sample, subtracting the result from the height because y grows downward on screen. `addLines()` then strings those points into one open path:

```
import SwiftUI
struct ElevationProfileView: View {
  let samples: [Double]
  private let lineStyle = StrokeStyle(
    lineWidth: 2,
    lineCap: .round,
    lineJoin: .round
  )
  var body: some View {
    GeometryReader { proxy in
      let points = profilePoints(in: proxy.size)
      linePath(through: points)
        .stroke(.teal, style: lineStyle)
    }
    .frame(height: 96)
  }
  private func profilePoints(in size: CGSize) -> [CGPoint]
    guard samples.count > 1,
```

```
            let lowest = samples.min(),
            let highest = samples.max(),
            highest > lowest
      else {
        return []
      }
      let stepX = size.width / Double(samples.count - 1)
      return samples.enumerated().map { index, elevation in
        let ratio = (elevation - lowest) / (highest - lowest
        return CGPoint(x: Double(index) * stepX, y: size.hei
      }
    }
    private func linePath(through points: [CGPoint]) -> Path
      Path { path in
        path.addLines(points)
      }
    }
}
#Preview {
    ElevationProfileView(samples: Hike.sample.elevationSampl
      .padding()
}
```

8. A bare line reads as a graph, but filling the area underneath it makes the climb legible at a glance. The area starts from the same points as the line, adds the two bottom corners, and ends with `closeSubpath()`, which draws the last edge back to the first point and turns the open line into a region we can fill. A `Rectangle` one point tall, aligned to the bottom of the `ZStack`, gives the profile a baseline to stand on, and a tinted `RoundedRectangle` behind everything separates the chart from the rest of the card. That makes four changes to `ElevationProfileView`: add the `areaGradient` property after `samples`, wrap the stroked line inside the `GeometryReader` in a `ZStack` with the filled area before it and the baseline after it, add the padding and the

background after the frame, and add `areaPath(through:in:)` after `linePath(through:)`:

```
struct ElevationProfileView: View {
  // ... the samples property
  private let areaGradient = LinearGradient(
    colors: [.teal.opacity(0.35), .teal.opacity(0.02)],
    startPoint: .top,
    endPoint: .bottom
  )
  // ... the lineStyle property
  var body: some View {
    GeometryReader { proxy in
      let points = profilePoints(in: proxy.size)
      ZStack(alignment: .bottom) {
        areaPath(through: points, in: proxy.size)
          .fill(areaGradient)
        linePath(through: points)
          .stroke(.teal, style: lineStyle)
        Rectangle()
          .fill(.secondary.opacity(0.35))
          .frame(height: 1)
      }
    }
    .frame(height: 96)
    .padding(12)
    .background(.teal.opacity(0.06), in: RoundedRectangle(
  }
  // ... profilePoints and linePath
  private func areaPath(through points: [CGPoint], in size
    Path { path in
      guard let first = points.first, let last = points.la
      path.addLines(points)
      path.addLine(to: CGPoint(x: last.x, y: size.height))
      path.addLine(to: CGPoint(x: first.x, y: size.height)
      path.closeSubpath()
    }
  }
}
```

9. Open `ContentView.swift` and assemble the card. The two halves stack with no spacing between them so the gradient meets the white surface flush, and `lowerCardShape` rounds the bottom two corners to mirror the header's top two. Replace the `ContentView` struct with the following:

```
struct ContentView: View {
  private let hike = Hike.sample
  private var lowerCardShape: UnevenRoundedRectangle {
    UnevenRoundedRectangle(
      cornerRadii: RectangleCornerRadii(bottomLeading: 24,
    )
  }
  var body: some View {
    ScrollView {
      VStack(spacing: 0) {
        HikeHeaderView(hike: hike)
        VStack(alignment: .leading, spacing: 20) {
          ProgressRingsView(hike: hike)
          ElevationProfileView(samples: hike.elevationSamp
        }
        .padding(20)
        .background(.background, in: lowerCardShape)
      }
      .padding()
    }
    .background(Color(.systemGroupedBackground))
  }
}
```

10. Close the card with a row of chips. The two filled ones use `Capsule` as a background shape, which is the right choice over a rounded rectangle whenever the corner should follow the height. The third is the same capsule stroked with a dashed `StrokeStyle` instead of filled, so it reads as an empty slot rather

than a value. Add `chips` as the last item of the inner `VStack`, and the three helpers below `body`:

```
struct ContentView: View {
  // ... the hike property and lowerCardShape
  var body: some View {
    ScrollView {
      VStack(spacing: 0) {
        HikeHeaderView(hike: hike)
        VStack(alignment: .leading, spacing: 20) {
          // ... the rings and the elevation profile
          chips
        }
        // ... the existing padding and background
      }
      // ... the existing padding
    }
    // ... the existing background
  }
  private var chips: some View {
    HStack {
      chip(hike.difficulty, systemImage: "figure.hiking")
      chip(hike.surface, systemImage: "mountain.2")
      placeholderChip("Heart rate")
    }
  }
  private func chip(_ title: String, systemImage: String)
    Label(title, systemImage: systemImage)
      .font(.caption.weight(.medium))
      .foregroundStyle(.teal)
      .padding(.horizontal, 12)
      .padding(.vertical, 8)
      .background(.teal.opacity(0.15), in: Capsule())
  }
  private func placeholderChip(_ title: String) -> some Vi
    Text(title)
      .font(.caption)
      .foregroundStyle(.secondary)
      .padding(.horizontal, 12)
      .padding(.vertical, 8)
      .overlay {
```

```
            Capsule()
              .stroke(
                .secondary.opacity(0.5),
                style: StrokeStyle(lineWidth: 1, dash: [4, 4])
              )
          }
      }
  }
```

11. Look at the finished card in the preview canvas. Every mark on it is a shape: the header is a rectangle with two rounded corners, the rings are arcs over circles, the profile is a path built from the hike's own numbers, and the chips are capsules. Change `distance` in the sample hike and watch the teal ring redraw, or drop a sample out of `elevationSamples` and watch the profile rescale itself, since nothing in the drawing code assumes a fixed count. For the complete `ContentView`, see the book's GitHub repository.



Figure 12.1: The finished hike summary card

How it works…

Our custom shape answers one question in `path(in:)`: given this rectangle, what outline should be drawn inside it? The rectangle arrives already sized by the layout system, so a shape never chooses how big it is, only what it looks like at whatever size it is handed. That is why `ProgressArc` measures from `rect.midX` and `rect.midY` and

reads its radius from `min(rect.width, rect.height) / 2`, and it is what lets the same type serve as both of our rings at different diameters with no changes. The coordinate system has its origin at the top-left with y increasing downward, which is worth holding on to, because it explains the two places our geometry looks odd. Starting the sweep a quarter turn back is what puts zero progress at the top of the ring rather than at three o'clock, and it is also why `profilePoints(in:)` subtracts the ratio from the height rather than using it directly, since the highest altitude has to become the smallest y.

The rings are where `stroke()` and `strokeBorder()` show why the distinction matters. A stroke is centered on the outline, so half of a 16-point line falls outside the shape and half falls inside. A border is drawn entirely within the frame, which is what `strokeBorder()` gives us on any `InsettableShape`, and every built-in shape is one. Our track is a `Circle` drawn with `strokeBorder()`, so it stays inside its frame, while `ProgressArc` is a custom shape with no inset behavior of its own and would spill 8 points past the track on both sides. The `.padding(lineWidth / 2)` on the arc is what pulls it back into alignment, and it is the reason the ring geometry is expressed as a formula rather than a measured constant: change `lineWidth` and the padding, the inset between rings, and the overall diameter all follow. Two other details are worth naming. `lineCap: .round` is what gives each arc its rounded ends, and `color.gradient` turns a flat color into a subtle shaded version of itself without us building a `Gradient` by hand.

Rounding only some corners used to mean drawing the rectangle ourselves. `UnevenRoundedRectangle` takes a `RectangleCornerRadii`, whose

four arguments each default to zero, so naming two of them is all it takes. Our card uses the type twice and in mirror image, once for the header's top corners and once for the body's bottom corners, and that is the honest reason to reach for it here: if we clipped the whole card with a plain `RoundedRectangle` instead, the two halves could not meet flush in the middle. The chips make the opposite point about picking a built-in shape. A `Capsule` always ends in half-circles whatever its height, so a chip keeps its pill shape as the text size changes, where a `RoundedRectangle` with a fixed radius would slowly stop looking like a pill.

The elevation profile is the one drawing that cannot be described until layout time, since the horizontal step between samples depends on the width available. `GeometryReader` exists for exactly that: it proposes its own size to its content and hands over a proxy describing it. It is worth using sparingly, because it takes all the space offered rather than sizing to its content, which is why ours is pinned with `.frame(height: 96)`. Notice that the line and the area come from the same array of points, computed once per layout pass. Both open with `addLines()`, which is shorthand for the two primitives underneath it: a `move(to:)` to the first point, then an `addLine(to:)` for every point after that. The built-in move is also a constraint, because it means `addLines()` can only ever start a subpath and never extend one, which is why the area's two bottom corners go on with individual `addLine(to:)` calls. `linePath` leaves the run open and strokes it, while `areaPath` adds those corners and calls `closeSubpath()`, and that closing call is the whole difference between an outline to stroke and a region to fill. A `Path` that is not closed can still be filled, but SwiftUI closes it implicitly with a straight line,

which here would have cut a diagonal from the last sample back to the first.

One shortcut deliberately left out of the code is `trim(from:to:)`, which takes any shape and keeps a fraction of its outline, so `Circle().trim(from: 0, to: progress)` would have produced our rings without a custom `Shape` at all. Building `ProgressArc` by hand is the better teacher here, and it also leaves us somewhere to go: a shape with a value like `progress` in it can be animated, so SwiftUI interpolates the outline rather than cross-fading between two pictures of it. We look at how to make that happen, and at the `@Animatable` macro that removes most of the work, in the *Keyframe and Phase Animations* section. When a scene needs hundreds of marks rather than the handful drawn here, the view-per-shape approach stops paying for itself, and the immediate-mode alternative is the subject of the *Canvas and Custom Rendering* section.

See also

- `Shape` documentation: https://x.swiftuicookbook.com/poule
- `Path` documentation: https://x.swiftuicookbook.com/uq7f1
- `UnevenRoundedRectangle` documentation: https://x.swiftuicookbook.com/qx6c0
- `strokeBorder(_:lineWidth:antialiased:)` documentation: https://x.swiftuicookbook.com/qw17x
- `GeometryReader` documentation: https://x.swiftuicookbook.com/krbw7

- *Drawing paths and shapes* tutorial: https://x.swiftuicookbook.com/41j8h

Canvas and Custom Rendering

Every mark in the previous recipe was a view, and that is the right trade until it stops being one. A few dozen shapes cost nothing, but a few hundred means a few hundred pieces of view identity for SwiftUI to create, compare, and lay out on every update, none of which the drawing needed. `Canvas` is the way out. It hands us a drawing context and asks us to fill it, much as overriding `draw(_:)` did in UIKit, except that it is still a SwiftUI view, still reads the environment, and still redraws when its inputs change.

In this recipe, we draw tonight's sky: four hundred stars, the Big Dipper, and three planets, all of them rendered by one `Canvas`. A night sky is a good vehicle for immediate-mode drawing because the whole scene is data rather than interface, and because it needs every part of `GraphicsContext` for an honest reason. We `fill` circles for the stars and `stroke` a `Path` through the constellation. We take a copy of the context and give it its own origin, rotation, and clip with `translateBy(x:y:)`, `rotate(by:)`, and `clip(to:)`, which is what makes the sky turn inside a fixed horizon ring while the ring itself stays put. A `TimelineView` on the `.animation` schedule drives that rotation. The planets come from the `symbols:` closure, where ordinary SwiftUI views are declared and then fetched with `resolveSymbol(id:)`, and the flare on the brightest stars is one `Image` passed through `context.resolve(_:)` and drawn many times. Along the way `opaque: true` tells the renderer there is nothing to blend against.

Getting ready

Create a new SwiftUI iOS app named **SkyChart** .

How to do it...

A canvas paints in the order its drawing calls run, so whatever is drawn later covers whatever came before it. That makes the sequence of the code the same as the stacking of the picture, and the steps follow it: the star data first, then the night background, then the stars, and finally the constellation and the planets that sit over them. Each layer arrives as its own private method plus a single call to it, so a step adds code rather than rewriting the step before it. These are the steps:

1. Create `Star.swift` in a `Models` folder. A star is a position and a brightness, and the two computed properties turn that brightness into something to draw: brighter stars get wider dots, and the brightest few earn a flare. Keeping those decisions on the model leaves the drawing code free of them. Positions are in unit space, measured from the center of the sky disc, so a coordinate of 1 lands on its edge:

```
import SwiftUI
struct Star {
  let position: CGPoint
  let magnitude: Double
  var dotDiameter: Double { 1 + magnitude * 3.2 }
  var isBright: Bool { magnitude > 0.97 }
}
```

2. Still in `Star.swift`, below the `Star` struct, add a seeded random number generator and an extension that builds the star field with it. Any type conforming to `RandomNumberGenerator` needs one method, `next()`, and this one advances a fixed seed through the standard linear congruential formula, so the sky looks the same on every launch and matches the one printed in this book. I seed anything procedural for that reason alone, because a rendering bug is hard enough to find without the art rearranging itself between runs. Where the field is built matters just as much: a canvas reruns its drawing closure from scratch on every redraw, and once the sky starts turning that means on every frame the display draws. Anything random inside the closure would answer differently each time it ran, so four hundred stars generated there would jitter instead of hanging still. A `static let` computes them once and fixes them in place:

```
struct SeededGenerator: RandomNumberGenerator {
  private var state: UInt64
  init(seed: UInt64) {
    state = seed
  }
  mutating func next() -> UInt64 {
    state = state &* 6364136223846793005 &+ 14426950408889
    return state
  }
}
extension Star {
  static let field: [Star] = {
    var generator = SeededGenerator(seed: 1972)
    return (0..<400).map { _ in
      Star(
        position: CGPoint(x: Double.random(in: -1...1, usi
                          y: Double.random(in: -1...1, usi
        magnitude: Double.random(in: 0.15...1, using: &gen
      )
```

```
    }
  }()
}
```

3. Create `SkyCanvasView.swift` in a `Views` folder with the empty sky. Everywhere else in SwiftUI, we hand a container a closure that returns views and let the framework arrange them. The closure we hand `Canvas` returns nothing. It receives a `GraphicsContext` and the `CGSize` it has to fill, and we produce the picture by calling drawing methods on that context, one after another, in the order we want them painted. Here that is a gradient over the whole canvas, then the horizon ring the sky sits inside. The `square(at:side:)` helper exists because almost everything here is a circle, and `Path(ellipseIn:)` wants the square that encloses it rather than a center and a radius:

```
import SwiftUI
struct SkyCanvasView: View {
  private let discInset: Double = 16
  private let nightGradient = Gradient(colors: [
    Color(red: 0.04, green: 0.06, blue: 0.18),
    Color(red: 0.01, green: 0.01, blue: 0.05)
  ])
  var body: some View {
    Canvas(opaque: true) { context, size in
      draw(in: context, size: size)
    }
  }
  private func draw(in context: GraphicsContext, size: CGS
    let center = CGPoint(x: size.width / 2, y: size.height
    let radius = min(size.width, size.height) / 2 - discIn
    context.fill(Path(CGRect(origin: .zero, size: size)),
                 with: .linearGradient(nightGradient, star
                                       endPoint: CGPoint(x
    context.stroke(Path(ellipseIn: square(at: center, side
```

```
                    with: .color(.white.opacity(0.25)), lin
  }
  private func square(at center: CGPoint, side: Double) ->
    CGRect(x: center.x - side / 2, y: center.y - side / 2,
           width: side, height: side)
  }
}
#Preview {
  SkyCanvasView()
}
```

4. Open `ContentView.swift` and put the sky on screen. The canvas ignores the safe area so the sky reaches under the navigation bar and the home indicator, and the app is locked to the dark color scheme, which keeps the navigation title legible over it. Replace the `ContentView` struct with the following:

```
struct ContentView: View {
  var body: some View {
    NavigationStack {
      SkyCanvasView()
        .ignoresSafeArea()
        .navigationTitle("Tonight's Sky")
        .navigationBarTitleDisplayMode(.inline)
    }
    .preferredColorScheme(.dark)
  }
}
```

5. Back in `SkyCanvasView.swift`, add the stars. Rather than convert every unit coordinate by hand, we take a copy of the context and move its origin to the middle of the sky disc, so a star at `(0, 0)` draws at the center and `point(_:radius:)` only has to scale, while the `clip(to:)` on the same copy trims the square field down to the round sky. The brightest stars also get a flare, and

where that image is resolved matters: `context.resolve(_:)` measures an `Image` and prepares it for drawing, so hoisting it above the loop prepares one image rather than a dozen. A resolved image carries a `shading` property too, which is how an SF Symbol gets tinted without building a view around it:

```
struct SkyCanvasView: View {
  private let discInset: Double = 16
  private let flareDiameter: Double = 14
  // ... the nightGradient and body properties
  private func draw(in context: GraphicsContext, size: CGS
    // ... the center and radius
    // ... the background fill and the horizon ring
    var sky = context
    sky.translateBy(x: center.x, y: center.y)
    sky.clip(to: Path(ellipseIn: square(at: .zero, side: r
    var flare = context.resolve(Image(systemName: "sparkle
    flare.shading = .color(.white.opacity(0.9))
    drawStars(in: sky, radius: radius, flare: flare)
  }
  private func drawStars(in context: GraphicsContext, radi
                         flare: GraphicsContext.ResolvedIm
    for star in Star.field {
      let center = point(star.position, radius: radius)
      context.fill(Path(ellipseIn: square(at: center, side
                   with: .color(.white.opacity(star.magnit
      if star.isBright {
        context.draw(flare, in: square(at: center, side: f
      }
    }
  }
  private func point(_ unit: CGPoint, radius: Double) -> C
    CGPoint(x: unit.x * radius, y: unit.y * radius)
  }
  // ... the square(at:side:) method
}
```

6. Set the sky turning. Wrap the `Canvas` in a `TimelineView`, add a `date` parameter to `draw(in:size:)`, and pass it `timeline.date`. A `TimelineView` redraws its content on a schedule and hands the closure the date of each update, and the `.animation` schedule asks for a redraw on every frame the display can show. That date is the only input the rotation needs: the two new lines at the top of `draw` divide the time elapsed by `rotationPeriod`, the length of one turn, and multiply by a full circle. Adding `rotate(by:)` to the `sky` copy, right after the translation, turns everything drawn through it, while the background and the horizon ring, drawn through the original context, stay where they are:

```
struct SkyCanvasView: View {
  private let rotationPeriod: Double = 120
  // ... the existing constants
  var body: some View {
    TimelineView(.animation) { timeline in
      Canvas(opaque: true) { context, size in
        draw(in: context, size: size, date: timeline.date)
      }
    }
  }
  private func draw(in context: GraphicsContext, size: CGS
    let elapsed = date.timeIntervalSinceReferenceDate
    let angle = Angle.degrees(elapsed / rotationPeriod * 3
    // ... the center and radius
    // ... the background fill and the horizon ring
    var sky = context
    sky.translateBy(x: center.x, y: center.y)
    sky.rotate(by: angle)
    // ... the clip, the flare, and the call to drawStars
  }
  // ... drawStars, point, and square
}
```

7. Create `Constellation.swift` in the `Models` folder. A constellation is a handful of named stars and a route through them. Storing the route as indices rather than as pairs of points lets one index appear twice, which is how the Big Dipper closes its bowl and still continues into its handle: the line visits Megrez at index 3, carries on around the bowl, and comes back to it at the end:

```
import SwiftUI
struct Constellation {
  let name: String
  let stars: [CGPoint]
  let outline: [Int]
}
extension Constellation {
  static let bigDipper = Constellation(
    name: "Big Dipper",
    stars: [
      CGPoint(x: 0.56, y: -0.28),
      CGPoint(x: 0.32, y: -0.40),
      CGPoint(x: 0.10, y: -0.48),
      CGPoint(x: -0.08, y: -0.44),
      CGPoint(x: -0.16, y: -0.24),
      CGPoint(x: -0.40, y: -0.26),
      CGPoint(x: -0.36, y: -0.52)
    ],
    outline: [0, 1, 2, 3, 4, 5, 6, 3]
  )
}
```

8. Draw the constellation over the field and name it. Mapping the outline through the points gives the line its route, and `addLines()` strings them into a single `Path` in one call, the same shorthand we used for the elevation profile in the *Shapes, Paths, and Drawing* section. Drawing the seven stars afterward puts them on top of the line rather than under it. The label needs more

care, because text drawn through the rotating context would turn upside down with everything else: `upright(_:at:angle:)` takes another copy, moves its origin to the star, and rotates it back by the same angle, leaving the label level whatever the sky is doing. Note that `context.resolve(_:)` works on a `Text` exactly as it did on an `Image`:

```
struct SkyCanvasView: View {
  // ... the rotationPeriod, discInset, and flareDiameter
  private let namedStarDiameter: Double = 5
  private let labelOffset: Double = 12
  // ... the nightGradient and body properties
  private func draw(in context: GraphicsContext, size: CGS
    // ... everything up to the call to drawStars
    drawConstellation(in: sky, radius: radius, angle: angl
  }
  // ... the drawStars method
  private func drawConstellation(in context: GraphicsConte
                                 radius: Double,
                                 angle: Angle,
                                 flare: GraphicsContext.Re
    let dipper = Constellation.bigDipper
    let points = dipper.stars.map { point($0, radius: radi
    var line = Path()
    line.addLines(dipper.outline.map { points[$0] })
    context.stroke(line, with: .color(.cyan.opacity(0.45))
                   style: StrokeStyle(lineWidth: 1.5, line
    for center in points {
      context.fill(Path(ellipseIn: square(at: center, side
                   with: .color(.white))
      context.draw(flare, in: square(at: center, side: fla
    }
    let label = context.resolve(Text(dipper.name).font(.ca
      .foregroundStyle(.cyan.opacity(0.8)))
    upright(context, at: points[0], angle: angle)
      .draw(label, at: CGPoint(x: 0, y: labelOffset), anch
  }
  private func upright(_ context: GraphicsContext, at poin
                       angle: Angle) -> GraphicsContext {
```

```
      var local = context
      local.translateBy(x: point.x, y: point.y)
      local.rotate(by: -angle)
      return local
    }
    // ... the point and square methods
  }
```

9. Create `Planet.swift` in the `Models` folder. To the naked eye a planet is a dot like any other, so what distinguishes one here is its color and size. The `id` doubles as the tag the canvas will look the planet up by, which is why it is a name rather than a number:

```
import SwiftUI
struct Planet: Identifiable {
  let id: String
  let color: Color
  let diameter: Double
  let position: CGPoint
}
extension Planet {
  static let visibleTonight = [
    Planet(id: "Venus", color: .white, diameter: 12,
           position: CGPoint(x: -0.40, y: 0.32)),
    Planet(id: "Mars", color: .orange, diameter: 9,
           position: CGPoint(x: 0.04, y: 0.44)),
    Planet(id: "Saturn", color: .yellow, diameter: 10,
           position: CGPoint(x: 0.40, y: 0.24))
  ]
}
```

10. Add the planets, which takes three additions: a `symbols:` closure on the `Canvas`, a call to `drawPlanets` at the end of `draw`, and the `drawPlanets` method itself. The `symbols:` closure is the second one `Canvas` accepts, and unlike the renderer it does return views:

everything in it is ordinary SwiftUI, laid out and rendered normally, which is how each marker gets a real `Text` and a gradient-filled `Circle` for free. Each one is labeled with `.tag()`, and `resolveSymbol(id:)` looks a marker up by that label, returning `nil` if nothing carries it, which is why the loop guards. The markers go through the same `upright` helper as the constellation label, so the planet names stay readable as the sky turns:

```
struct SkyCanvasView: View {
  // ... the constants
  var body: some View {
    TimelineView(.animation) { timeline in
      Canvas(opaque: true) { context, size in
        draw(in: context, size: size, date: timeline.date)
      } symbols: {
        ForEach(Planet.visibleTonight) { planet in
          VStack(spacing: 4) {
            Circle()
              .fill(planet.color.gradient)
              .frame(width: planet.diameter, height: plane
            Text(planet.id)
              .font(.caption2)
              .foregroundStyle(.white.opacity(0.75))
          }
          .tag(planet.id)
        }
      }
    }
  }
  private func draw(in context: GraphicsContext, size: CGS
    // ... everything up to the call to drawConstellation
    drawPlanets(in: sky, radius: radius, angle: angle)
  }
  // ... the drawStars and drawConstellation methods
  private func drawPlanets(in context: GraphicsContext,
                           radius: Double, angle: Angle) {
    for planet in Planet.visibleTonight {
      guard let marker = context.resolveSymbol(id: planet.
```

```
            let center = point(planet.position, radius: radius)
            upright(context, at: center, angle: angle).draw(mark
        }
    }
    // ... the upright, point, and square methods
}
```

11. Watch the sky in the preview canvas using live mode. The sky completes a turn every two minutes, so give it a few seconds and the Big Dipper will visibly swing while the horizon ring holds still and the three planet names stay level. Change the seed in `Star.field` for a different sky, or raise the star count well past four hundred and notice how little the frame rate cares. For the complete `SkyCanvasView`, see the book's GitHub repository.



Figure 12.2: Tonight's sky, with the Big Dipper and three planets

How it works…

Our sky is four hundred stars and not one view. A canvas draws in immediate mode, which is the opposite of how the rest of SwiftUI works: there is nothing to give identity to, and nothing to compare against a previous version. The renderer closure runs top to bottom and paints, and it runs again in full whenever anything it depends

on changes. That is the source of the performance win, and also of the one rule the recipe bends everything else around: the closure must be a pure function of its inputs. Four hundred calls to `Double.random` inside it would produce four hundred different stars on every frame, which is why `Star.field` is a `static let` computed once. What we give up is everything view identity buys: a canvas has no hit testing, no transitions, and no accessibility of its own, so interactive or assistive content belongs in real views layered over it.

The context is a value, and the recipe leans on that constantly. Drawing methods such as `fill`, `stroke`, and `draw` are not mutating, because they send their output straight to the canvas the context belongs to, which every copy shares. Transforms and clips are different: `translateBy(x:y:)`, `rotate(by:)`, and `clip(to:)` all mutate the context, so they change only the copy they are called on. That distinction is what lets one drawing work in several coordinate systems at once. The background and the horizon ring go through the original context and stay fixed to the canvas, while the `sky` copy carries a moved origin, a rotation, and a circular clip, so everything drawn through it turns and gets trimmed together.

Order matters inside that copy, because each transform applies to the coordinate system left by the one before it. Translating and then rotating spins the sky around the middle of the canvas, where rotating first would send it into orbit around the top-left corner instead. The same rule explains two details that look odd on their own. The clip is stated as `square(at: .zero, ...)` rather than at the canvas center, because by the time it runs the origin has already moved there. And `upright(_:at:angle:)` moves to the star while the rotation is still in effect, then undoes that rotation locally, so the label

sits at a rotated position without being rotated itself. That helper also shows the value semantics paying off, since it can build a context, hand it back as a return value, and leave every other copy untouched.

`TimelineView` supplies the one input that changes on its own. The `.animation` schedule redraws as fast as the display refreshes, and each redraw arrives with a `date`, which the rotation is derived from directly rather than accumulated frame by frame. Deriving values from a timestamp rather than stepping them cannot drift and stays correct when frames are dropped. The schedule also accepts a `paused` argument, and there the same habit needs care: pausing stops the dates rather than the clock, so an animation read from an absolute timestamp lurches forward when the schedule restarts unless the time spent paused is tracked and subtracted from it.

Two details in the drawing code are there for performance rather than appearance. `opaque: true` promises the renderer that every pixel gets painted, which it can then take advantage of by skipping the blend against whatever is behind the canvas; the promise is only safe here because the gradient fills the entire size before anything else is drawn. And `context.resolve(_:)` moves the measuring of an image or a text out of the loop, as the flare does. Neither will be measurable with four hundred stars on a modern device, which is the point worth taking away: reach for `Canvas` when the shape count genuinely justifies it, and stay with the views from the *Shapes, Paths, and Drawing* section when it does not , because those bring animation, layout, and accessibility that a canvas asks us to rebuild by hand.

See also

- `Canvas` documentation: https://x.swiftuicookbook.com/h43ki
- `GraphicsContext` documentation: https://x.swiftuicookbook.com/9pm0x
- `TimelineView` documentation: https://x.swiftuicookbook.com/4hr68
- `TimelineSchedule` documentation: https://x.swiftuicookbook.com/fpmdj
- `resolveSymbol(id:)` documentation: https://x.swiftuicookbook.com/klcb8
- Add rich graphics to your SwiftUI app (WWDC 2021): https://x.swiftuicookbook.com/sowzi

Animations, Transitions, and Matched Geometry

Animating a view in UIKit meant describing the change twice: once as the new frame or alpha, and once inside an animation block that told the system how long to take to get there. SwiftUI removes the second description. We change a value, and the framework works out that something moved, how far, and interpolates it. What is left for us to choose is not **what** to animate but **how** it should feel, and that turns out to be a design decision rather than a mechanical one.

In this recipe, we build a gallery wall: a grid of small framed pieces that expand into a large card when tapped, and settle back into place

when dismissed. A wall is a good vehicle for motion because a tapped piece has to read as one object growing, rather than as a small view disappearing while a large one appears somewhere else. That effect has a name, the hero animation, and `@Namespace` together with `.matchedGeometryEffect()` is how we build one. A picker along the bottom re-drives every animation in the app, so `.easeInOut()` , `.snappy()` , `.bouncy()` , and `.interpolatingSpring()` can be compared by feel instead of read about in a table. Along the way we use `withAnimation` where a tap should move several things at once, `.animation(_:value:)` where a view should animate its own property no matter who changed it, `.transaction()` to keep one element out of th e shared animation, and `.transition()` with `.asymmetric()` a nd a custom `Transition` for a panel that unfolds.

Getting ready

Create a new SwiftUI iOS app named **ArtWall** .

How to do it...

We build the data first, then the smallest visible view, and only then the wall that arranges them, because a mistake in the tile is far easier to see in a preview of one tile than in a grid of six. The motion comes last, layered onto a wall that already lays out correctly. These are the steps:

1. Create `ArtPiece.swift` in a `Models` folder. Each piece carries what the wall and the card both need to draw it, and the `id` is a plain string rather than a `UUID` because it doubles as the matched-geometry identifier later, and a stable, readable value makes

that pairing easy to reason about. The `year` gets a `yearText` companion so it renders as `1971` rather than picking up a thousands separator:

```
import SwiftUI
struct ArtPiece: Identifiable {
  let id: String
  let title: String
  let artist: String
  let year: Int
  let symbol: String
  let colors: [Color]
  let note: String
  var yearText: String { String(year) }
}
extension ArtPiece {
  static let wall: [ArtPiece] = [
    ArtPiece(id: "tideline", title: "Tideline",
             artist: "Mara Ellis", year: 1971,
             symbol: "water.waves", colors: [.teal, .blue]
             note: "Ellis painted this coast every morning
    // ... the remaining five art pieces
  ]
  static let sample = wall[0]
}
```

For the complete initialization code of the `wall` static property, see the book's GitHub repository.

2. Create `MotionStyle.swift` in the same folder. This is the whole comparison the app is built around, reduced to one property: four cases, four `Animation` values. Every view that animates takes a `MotionStyle` and reads its `animation` property, so a single selection drives every animation in the app. That is what makes

the four comparable: the same gesture, animated four different ways:

```
import SwiftUI
enum MotionStyle: String, CaseIterable, Identifiable {
  case ease = "Ease"
  case snappy = "Snappy"
  case bouncy = "Bouncy"
  case spring = "Spring"
  var id: String { rawValue }
  var animation: Animation {
    switch self {
    case .ease: .easeInOut(duration: 0.6)
    case .snappy: .snappy(duration: 0.25)
    case .bouncy: .bouncy(duration: 0.5, extraBounce: 0.1)
    case .spring: .interpolatingSpring(duration: 1, bounce
    }
  }
}
```

3. Create `ArtworkView.swift` in a `Views` folder. The tile on the wall and the expanded card are the two ends of the hero animation, and both draw the same artwork, so it lives in one view that takes its symbol size as a parameter. Using a gradient and an SF Symbol instead of image assets keeps the focus on the motion, and `.aspectRatio(1, contentMode: .fit)` makes every artwork square regardless of the space it is given:

```
import SwiftUI
struct ArtworkView: View {
  let piece: ArtPiece
  let symbolSize: Double
  var body: some View {
    LinearGradient(colors: piece.colors, startPoint: .topL
                   endPoint: .bottomTrailing)
```

```
      .aspectRatio(1, contentMode: .fit)
      .overlay {
        Image(systemName: piece.symbol)
          .font(.system(size: symbolSize))
          .foregroundStyle(.white.opacity(0.9))
      }
  }
}
#Preview {
  ArtworkView(piece: .sample, symbolSize: 60)
    .padding()
}
```

4. Create `ArtTileView.swift` in the `Views` folder to represent the framed piece as it hangs on the wall. The interesting line is the last modifier on the caption: `.animation(motion.animation, value: showsLabel)` attaches the animation to the label itself, so the caption fades whenever that flag changes without the code that flipped it having to know. Animating `opacity` rather than removing the `Text` keeps every tile the same height, which matters once the tiles start handing their geometry to a card:

```
import SwiftUI
struct ArtTileView: View {
  let piece: ArtPiece
  let motion: MotionStyle
  let showsLabel: Bool
  var body: some View {
    VStack(spacing: 8) {
      ArtworkView(piece: piece, symbolSize: 40)
      Text(piece.title)
        .font(.footnote.weight(.medium))
        .lineLimit(1)
        .opacity(showsLabel ? 1 : 0)
        .animation(motion.animation, value: showsLabel)
    }
    .padding(10)
```

```
    .background(.background)
    .clipShape(RoundedRectangle(cornerRadius: 10))
    .shadow(color: .black.opacity(0.15), radius: 6, y: 3)
  }
}
#Preview {
  ArtTileView(piece: .sample, motion: .snappy, showsLabel:
    .padding()
}
```

5. Replace the `ContentView` struct with the following, where a scrolling grid of tiles sits above the motion picker, and a toolbar button toggles the captions. Nothing here mentions animation, yet the tag button already animates all six captions, because each tile brought its own `.animation()` along:

```
struct ContentView: View {
  @State private var motion: MotionStyle = .snappy
  @State private var showsLabels = true
  private let columns = [GridItem(.adaptive(minimum: 150),
  var body: some View {
    NavigationStack {
      VStack(spacing: 0) {
        grid
        motionPicker
      }
      .navigationTitle("ArtWall")
      .toolbar {
        Button("Labels", systemImage: showsLabels ? "tag.f
          showsLabels.toggle()
        }
      }
    }
  }
  private var grid: some View {
    ScrollView {
      LazyVGrid(columns: columns, spacing: 16) {
        ForEach(ArtPiece.wall) { piece in
```

```
            ArtTileView(piece: piece, motion: motion, showsL
          }
        }
        .padding(16)
      }
    }
    private var motionPicker: some View {
      Picker("Motion", selection: $motion) {
        ForEach(MotionStyle.allCases) { style in
          Text(style.rawValue).tag(style)
        }
      }
      .pickerStyle(.segmented)
      .padding(.horizontal, 16)
      .padding(.vertical, 10)
      .background(.bar)
    }
}
```

With the preview canvas in live mode, pick a motion and tap the tag icon in the toolbar. The captions fade with whichever animation is selected, although on a fade the four motions differ mostly in timing. Opacity cannot go past fully visible or fully hidden, so the overshoot of a spring is clipped away: **Bouncy** looks here like a quicker ease, and only **Spring** bounces far enough to make a disappearing caption flicker back for an instant. The bounces show in full in step 9, once a card grows out of its tile.

6. Now the first half of the hero animation. Add the selection state and a `@Namespace`, which is the shared identifier space SwiftUI uses to pair two views as the same object, then rebuild `grid` so the selected piece leaves a hidden copy of its own tile behind.

Reusing `ArtTileView` for the placeholder rather than a `Color.clear` guarantees the gap is exactly the size of the tile that left it:

```
struct ContentView: View {
  @State private var motion: MotionStyle = .snappy
  @State private var selected: ArtPiece?
  @State private var showsLabels = true
  @Namespace private var wall
  // ... the columns property and the body
  private var grid: some View {
    ScrollView {
      LazyVGrid(columns: columns, spacing: 16) {
        ForEach(ArtPiece.wall) { piece in
          if selected?.id == piece.id {
            ArtTileView(piece: piece, motion: motion, show
              .hidden()
          } else {
            ArtTileView(piece: piece, motion: motion, show
              .matchedGeometryEffect(id: piece.id, in: wal
              .onTapGesture {
                withAnimation(motion.animation) {
                  selected = piece
                }
              }
          }
        }
      }
      .padding(16)
    }
  }
  // ... motionPicker
}
```

7. A tapped tile now vanishes with nowhere to go, so let's give the wall something to recede behind. Inside the `NavigationStack`, wrap the `VStack` in a `ZStack` and add a dim backdrop after the `VStack`, so it lies over the wall and dismisses the selection when

tapped. The backdrop stays in the view tree permanently and animates its `opacity`, rather than being inserted and removed, because `.transaction()` can only override the animation of a value change; that override is what keeps the dim flat while the card bounces. A fully transparent view still absorbs taps, so `.allowsHitTesting()` has to follow the selection:

```
struct ContentView: View {
  // ... the properties
  var body: some View {
    NavigationStack {
      ZStack {
        VStack(spacing: 0) {
          grid
          motionPicker
        }
        backdrop
      }
      .navigationTitle("ArtWall")
      // ... the toolbar
    }
  }
  // ... the grid property
  private var backdrop: some View {
    Rectangle()
      .fill(.black.opacity(0.5))
      .opacity(selected == nil ? 0 : 1)
      .ignoresSafeArea()
      .transaction { $0.animation = .easeInOut(duration: 0
      .allowsHitTesting(selected != nil)
      .onTapGesture {
        withAnimation(motion.animation) {
          selected = nil
        }
      }
  }
  // ... the motionPicker property
}
```

8. Create `ArtCardView.swift` in the `Views` folder, the expanded form of a piece. It draws the same `ArtworkView` at a larger symbol size, which is what lets the two ends of the animation read as one object growing rather than one view swapping for another. The card takes the current `motion` because it will run an animation of its own shortly:

```
import SwiftUI
struct ArtCardView: View {
  let piece: ArtPiece
  let motion: MotionStyle
  var body: some View {
    VStack(spacing: 0) {
      ArtworkView(piece: piece, symbolSize: 90)
      VStack(alignment: .leading, spacing: 10) {
        Text(piece.title)
          .font(.title3.weight(.semibold))
        Text("\(piece.artist), \(piece.yearText)")
          .font(.subheadline)
          .foregroundStyle(.secondary)
      }
      .frame(maxWidth: .infinity, alignment: .leading)
      .padding(16)
    }
    .background(.background)
    .clipShape(RoundedRectangle(cornerRadius: 16))
    .shadow(color: .black.opacity(0.3), radius: 20, y: 8)
  }
}
#Preview {
  ArtCardView(piece: .sample, motion: .snappy)
    .padding()
}
```

9. Complete the hero by putting the card in the `ZStack` above the backdrop, carrying the same `matchedGeometryEffect` identifier as the tile that vanished. The `.id(piece.id)` matters more than it

looks: it gives each piece's card its own identity, so choosing a different piece while the previous card is still animating inserts a new card rather than repurposing the old one:

```
struct ContentView: View {
  // ... the properties
  var body: some View {
    NavigationStack {
      ZStack {
        // ... the VStack and the backdrop
        if let piece = selected {
          ArtCardView(piece: piece, motion: motion)
            .matchedGeometryEffect(id: piece.id, in: wall)
            .id(piece.id)
            .padding(28)
            .zIndex(1)
        }
      }
      // ... the navigation title and the toolbar
    }
  }
  // ... grid, backdrop, and motionPicker
}
```

Back in the preview canvas, tap a piece. It grows out of its own tile and settles into the middle of the dimmed wall, and tapping outside sends it home.

10. Return to `ArtCardView.swift` and add a control that reveals the curator's note. The button lives below the artist line inside the caption stack, and the chevron rotates to point at the state it will move to. Wrapping the change in `withAnimation` is the explicit form: the caller decides that this particular state change should animate, and everything affected by it moves together:

```
struct ArtCardView: View {
  // ... the piece and motion properties
  @State private var showsNote = false
  var body: some View {
    VStack(spacing: 0) {
      // ... the artwork
      VStack(alignment: .leading, spacing: 10) {
        // ... the title and the artist line
        noteButton
      }
      // ... the frame and padding
    }
    // ... the background, clip shape, and shadow
  }
  private var noteButton: some View {
    Button {
      withAnimation(motion.animation) {
        showsNote.toggle()
      }
    } label: {
      HStack(spacing: 4) {
        Text("Curator's note")
        Image(systemName: "chevron.down")
          .rotationEffect(.degrees(showsNote ? 180 : 0))
      }
      .font(.subheadline.weight(.medium))
    }
  }
}
```

11. Still in `ArtCardView.swift`, add the note below the button along with the transition that reveals it. None of the built-in transitions fits: `.move(edge: .top)` slides a full-size panel in from outside its own frame, so it crosses over the button on the way, and what we want is the note growing out of its own top edge, the way the card grows around it. `UnfoldTransition` conforms to the `Transition` protocol, which lets a type describe its own

arrival and departure. Wrapping it in `AnyTransition` lets it sit inside `.asymmetric()`, which is what gives the two directions different treatment: unfolding in reads as the card making room, while unfolding back out would draw attention to something the user has finished reading, so the removal simply fades:

```
struct ArtCardView: View {
  // ... the properties
  var body: some View {
    VStack(spacing: 0) {
      // ... the artwork
      VStack(alignment: .leading, spacing: 10) {
        // ... the title, the artist line, and the note bu
        if showsNote {
          Text(piece.note)
            .font(.callout)
            .transition(.asymmetric(insertion: .unfold, re
        }
      }
      // ... the frame and padding
    }
    // ... the background, clip shape, and shadow
  }
  // ... noteButton
}
struct UnfoldTransition: Transition {
  func body(content: Content, phase: TransitionPhase) -> s
    content
      .scaleEffect(y: phase.isIdentity ? 1 : 0, anchor: .t
      .opacity(phase.isIdentity ? 1 : 0)
  }
}
extension AnyTransition {
  static var unfold: AnyTransition { AnyTransition(UnfoldT
}
```

12. Exercise the finished app in the preview canvas using live mode. Tap a piece and watch it grow out of its tile, open the curator's note, then dismiss the card by tapping the dimmed wall. Now change the motion and do it again: **Ease** glides and stops dead, **Snappy** is over almost before it starts, **Bouncy** overshoots once, and **Spring** wobbles its way to rest. Try dismissing a card and immediately tapping a different piece, and both cards animate at once, each on its o wn spring. For the complete `ContentView`, see the book's GitHub repository.

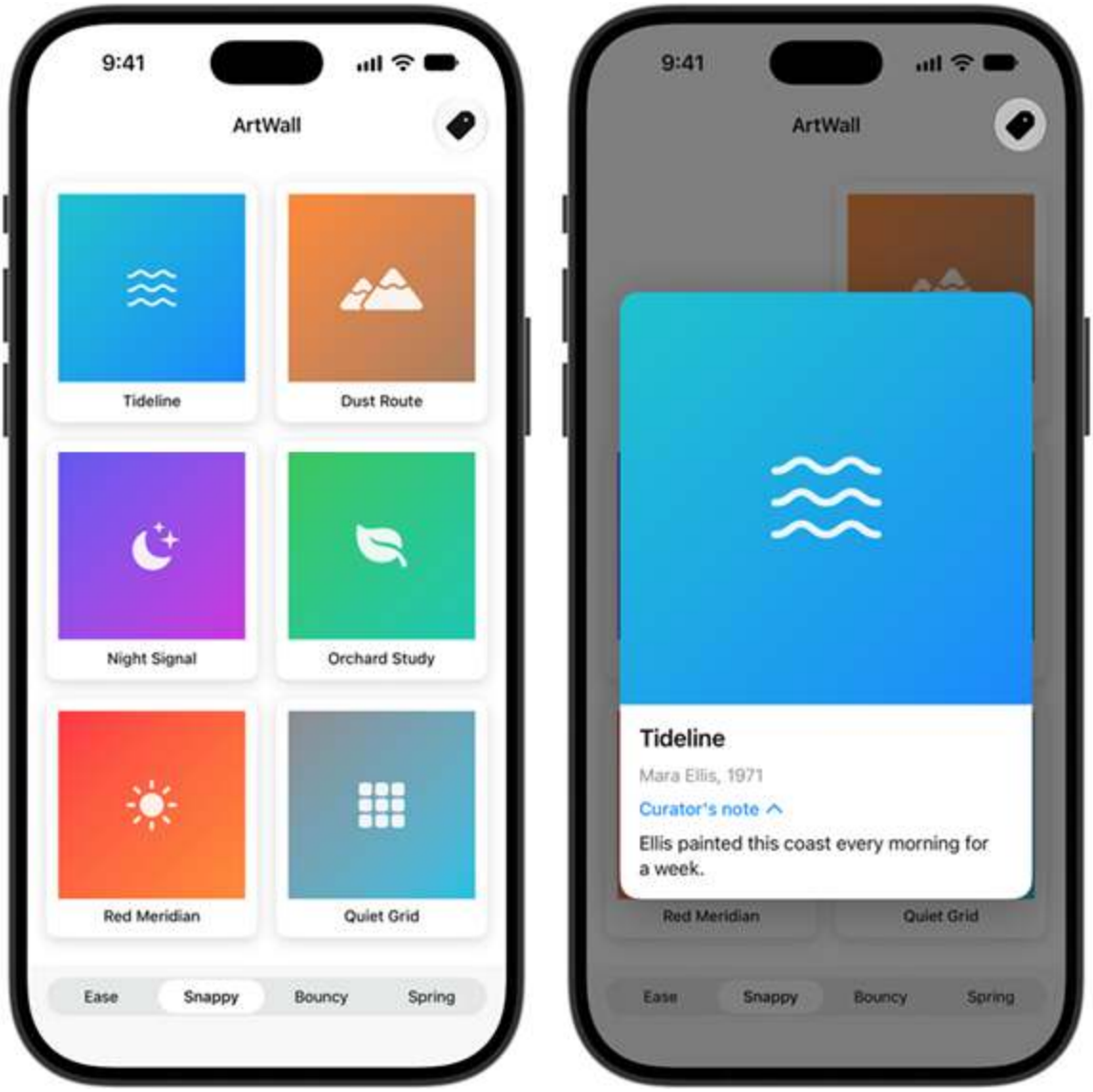


Figure 12.3: The gallery wall and one piece expanded with its curator's note

How it works...

Every animation in this app consists of the same three ingredients: something changes a value, a view's appearance depends on that value, and SwiftUI is told how to get from the old appearance to the new one. What varies is who supplies that last ingredient. When we tapped a tile, `withAnimation(motion.animation)` supplied it at the call

site, and it reached everything that changed as a result: the tile's removal, the card's insertion, the geometry pairing between them, and the backdrop's opacity. That is the **explicit** form, and it is the right one whenever a single gesture should move several things as one. The caption on each tile takes the **implicit** form instead. `.animation(motion.animation, value: showsLabel)` sits next to the property it animates, so the toolbar button can flip `showsLabels` with a bare `toggle()` and six captions still fade correctly. The rule of thumb is: reach for `withAnimation` when the caller knows what the change means, and for `.animation(_:value:)` when the view does.

The four motions in our picker split into two families that behave quite differently. `.easeInOut(duration:)` is a timing curve: it interpolates from start to finish over a fixed duration and stops exactly on the target. The other three are springs, simulations of a mass on a spring settling toward a resting point, and their `duration` is a perceptual setting rather than a hard stop. `.snappy()` and `.bouncy()` are the same spring with different amounts of bounce baked in, while `.interpolatingSpring(duration:bounce:)` lets us name the bounce directly and, unlike the presets, preserves velocity when an animation is interrupted mid-flight, which is why it recovers gracefully when the user taps a second piece before the first has settled. The older `.spring(response:dampingFraction:)` spelling is still available and describes the same physics in different units.

A spring overshoots its target and comes back, and that has a consequence worth knowing when the animated property has a fixed range. Watch a caption disappear under **Spring** : the opacity undershoots past zero, where it is clamped and invisible, then rebounds above zero, so the caption briefly reappears before settling.

Nothing is wrong, it is what a bouncy spring genuinely does, but it is the reason a fade usually wants a calmer animation than a move or a scale does. Our dim backdrop is the case where we did something about it. Left alone it inherited the tap's spring and pulsed light and dark behind the card, so `.transaction { $0.animation = .easeInOut(duration: 0.25) }` overrides the inherited animation for that one subtree and nothing else. There is a catch that costs people an afternoon: a `.transaction()` modifier has no effect on the animation of an insertion or removal, only on value changes. That is why the backdrop is always present and animates its `opacity`, rather than appearing and disappearing with the card.

When we tapped a tile, nothing actually moved. The tile was removed from the grid and a card was inserted into the `ZStack`, two separate events in two different parts of the view hierarchy, and the hero animation is what ties them together. It works by having two different views claim to be the same thing. `@Namespace` declares an identifier space, and `.matchedGeometryEffect(id:in:)` enrolls a view in it under a given `id`. When one enrolled view is removed in the same transaction that inserts another with the same `id`, SwiftUI stops treating them as two events and instead animates the frame of the departing view into the frame of the arriving one. That is why the tile is not simply hidden with `.opacity(0)`: it has to genuinely leave the hierarchy for the pairing to happen. Reusing `ArtTileView` as the invisible placeholder keeps the grid from reflowing while its slot is empty, and `.zIndex(1)` keeps the card above the wall for the whole journey. The cross-navigation cousin of this API, `.matchedTransitionSource()` paired with `.navigationTransition(.zoom)`, animates a push from one view to another instead of a change inside

a single view, and we used it in the *NavigationStack and NavigationPath* section of *Chapter 11*.

Identity is what makes that pairing reliable, and `.id(piece.id)` on the card is a small line with a large effect. Without it the card is a single view whose `matchedGeometryEffect` identifier merely changes when the selection does, and SwiftUI reasonably reads that as one card moving to a new group, so a piece tapped mid-dismissal grows out of the previous card's position instead of its own tile. Giving each piece's card its own identity turns that into a removal and an insertion, and each gets its own spring, which is why two cards can be in flight at once without interfering.

Transitions cover the remaining case: a view entering or leaving the hierarchy, where there is no old value to interpolate from. `.transition()` names what that view should look like while it is absent, and SwiftUI runs it in whatever animation is in effect, which is why a transition with no `withAnimation` around the state change does nothing at all. The built-in transitions compose, with `.combined(with:)` running two at once and `.asymmetric(insertion:removal:)` choosing different ones for each direction, and for anything they cannot express we write a type conforming to `Transition`. `UnfoldTransition` receives a `TransitionPhase` and returns the content styled for it, and `phase.isIdentity` is true only when the view is fully present, which is why a single expression covers both directions. Because `scaleEffect(y:anchor:)` scales along one axis only, it d oes something none of the built-ins offer. Wrapping it in `AnyTransition` lets it sit alongside `.opacity` inside `.asymmetric()`, which still expects the older type-erased form.

See also

- `withAnimation(_:_:)` documentation: https://x.swiftuicookbook.com/uydio
- `Animation` documentation: https://x.swiftuicookbook.com/22d6o
- `animation(_:value:)` documentation: https://x.swiftuicookbook.com/m2s69
- `Transaction` documentation: https://x.swiftuicookbook.com/nr_da7
- `Transition` documentation: https://x.swiftuicookbook.com/bkka5
- `matchedGeometryEffect(id:in:properties:anchor:isSource:)` documentation: https://x.swiftuicookbook.com/9pw1i

Keyframe and Phase Animations

A single animation carries a value from where it is to where it should be, which covers most of what an interface needs. Motion that happens in stages is a different problem: a ball that drops, flattens against the floor, and bounces back, or a status icon that cycles through a handful of attention states while a long task runs. In UIKit, we reached for chained completion handlers, or for a `CAKeyframeAnimation` handed a list of values and key times. SwiftUI splits the problem in two, and choosing between the halves comes down to a single question: are the stages poses the view could hold, or is it one continuous timeline that happens to change direction?

In this recipe, we build a pour-over coffee timer. A badge at the top mimes a pour while the app sits idle, cycling through three poses with `PhaseAnimator`. A ring around the cup sweeps and thickens as the brew runs, drawn by a custom `Shape` whose two animated properties are wired up by the iOS 26 `@Animatable` macro, with `@AnimatableIgnored` marking the one property that has no meaningful midpoint. A drop falls into the cup and squashes on impact, choreographed by `KeyframeAnimator` across three independent `KeyframeTrack` timelines. Fi nally, the same choreography is handed to a `KeyframeTimeline` so a slider can step through it by hand, which is the fastest way to see what a set of keyframes actually contains.

Getting ready

Create a new SwiftUI iOS app named **BrewTimer** .

How to do it...

Each piece of motion in this app needs a type of its own. Rather than writing all of them up front and assembling at the end, we define each one immediately before the step that puts it on screen, so nothing sits unused for long and every new type shows up in the preview canvas within a step or two of being written. These are the steps:

1. Create `PourPhase.swift` in a `Models` folder. A phase is a pose the badge can hold, so the enum carries everything one pose needs: how far the mug is tipped, how large it is, and the animation that carries the badge into it. Note that `tipped` and `pouring` share an angle and differ only in scale, which will matter shortly:

```
import SwiftUI
enum PourPhase: CaseIterable {
  case upright, tipped, pouring
  var angle: Angle {
    switch self {
    case .upright: .degrees(0)
    case .tipped, .pouring: .degrees(-30)
    }
  }
  var scale: Double {
    switch self {
    case .upright, .tipped: 1
    case .pouring: 1.12
    }
  }
  var animation: Animation {
    switch self {
    case .upright: .snappy(duration: 0.4)
    case .tipped: .easeInOut(duration: 0.9)
    case .pouring: .bouncy(duration: 0.5)
    }
  }
}
```

2. Replace the `ContentView` struct with the following. The `.phaseAnimator()` modifier takes the sequence of phases, a closure that applies the current phase to the view, and a second closure that supplies the animation for each one. It loops on its own: there is no state driving it and nothing triggering it. The `.geometryGroup()` isolates the icon's position from that animation, so it does not sweep in from the corner while the preview canvas resolves its first layout. The `isBrewing` flag is the switch the brew will throw later, and for now it only chooses the label:

```
struct ContentView: View {
  @State private var isBrewing = false
```

```
  var body: some View {
    NavigationStack {
      VStack(spacing: 28) {
        pourBadge
      }
      .padding(24)
      .navigationTitle("BrewTimer")
    }
  }
  private var pourBadge: some View {
    HStack(spacing: 10) {
      Image(systemName: "mug.fill")
        .font(.title2)
        .phaseAnimator(PourPhase.allCases) { mug, phase in
          mug
            .rotationEffect(phase.angle, anchor: .bottomLe
            .scaleEffect(phase.scale)
        } animation: { phase in
          phase.animation
        }
        .geometryGroup()
      Text(isBrewing ? "Brewing" : "Ready to pour")
        .font(.headline)
    }
    .padding(.horizontal, 16)
    .padding(.vertical, 10)
    .background(.thinMaterial, in: .capsule)
  }
}
```

With the preview canvas in live mode, the badge should already be tipping and returning on a loop.

3. Create `BrewArc.swift` in a `Views` folder. The ring is a filled band rather than a stroked circle, which means `strokedPath()` bakes the thickness into the path itself and both the sweep and the width have to animate. The `@Animatable` macro writes the `animatableData` property that makes that possible, and

`@AnimatableIgnored` marks the one stored property it has to leave out:

```
import SwiftUI
@Animatable
struct BrewArc: Shape {
  var progress: Double
  var lineWidth: CGFloat
  @AnimatableIgnored var clockwise = false
  func path(in rect: CGRect) -> Path {
    let radius = min(rect.width, rect.height) / 2 - lineWi
    let center = CGPoint(x: rect.midX, y: rect.midY)
    let start = Angle.degrees(-90)
    let end = start + .degrees(360 * progress)
    var path = Path()
    path.addArc(center: center, radius: radius, startAngle
                endAngle: end, clockwise: clockwise)
    return path.strokedPath(StrokeStyle(lineWidth: lineWid
  }
}
#Preview {
  BrewArc(progress: 0.7, lineWidth: 16)
    .fill(.orange.gradient)
    .frame(width: 220, height: 220)
    .padding()
}
```

4. Back in `ContentView.swift`, add the brew stage: a track ring at full sweep, the progress ring drawn on top of it, and the cup in the middle. Both rings read the same `ringWidth`, so the whole band thickens together as the brew runs:

```
struct ContentView: View {
  @State private var isBrewing = false
  @State private var progress = 0.0
  @State private var ringWidth = 10.0
  var body: some View {
```

```
      NavigationStack {
        VStack(spacing: 28) {
          pourBadge
          brewStage
        }
        // ... the padding and the navigation title
      }
    }
    // ... the pourBadge computed property
    private var brewStage: some View {
      ZStack {
        BrewArc(progress: 1, lineWidth: ringWidth)
          .fill(.quaternary)
        BrewArc(progress: progress, lineWidth: ringWidth)
          .fill(.orange.gradient)
        Image(systemName: "cup.and.heat.waves.fill")
          .font(.system(size: 68))
          .foregroundStyle(.brown)
      }
      .frame(width: 250, height: 250)
    }
}
```

5. Add the button that starts a brew, along with the two functions that run the cycle. `startBrew()` animates the ring to full over six seconds, and its completion hands off to `clearRing()`, which waits a beat so the finished ring registers before emptying it, then releases the controls. Chaining the two this way means the whole cycle runs without a timer:

```
struct ContentView: View {
  // ... the 3 state properties
  private let brewDuration = 6.0
  var body: some View {
    NavigationStack {
      VStack(spacing: 28) {
        pourBadge
        brewStage
```

```
            brewButton
        }
        // ... the padding and the navigation title
      }
    }
    // ... the pourBadge and brewStage computed properties
    private var brewButton: some View {
      Button("Brew", systemImage: "drop.fill") {
        startBrew()
      }
      .buttonStyle(.borderedProminent)
      .controlSize(.large)
      .disabled(isBrewing)
    }
    private func startBrew() {
      isBrewing = true
      withAnimation(.easeInOut(duration: brewDuration)) {
        progress = 1
        ringWidth = 20
      } completion: {
        clearRing()
      }
    }
    private func clearRing() {
      withAnimation(.easeInOut(duration: 0.6).delay(1.5)) {
        progress = 0
        ringWidth = 10
      } completion: {
        isBrewing = false
      }
    }
}
```

Back in the preview canvas, tap **Brew** and watch the ring sweep and thicken at the same time, hold when it reaches full, and then empty itself before the button becomes tappable again.

6. Create `PourValues.swift` in the `Models` folder. This is the value the drop animation interpolates: its vertical offset, how much it is squashed, and how visible it is. Holding all three in one struct is what lets a single animator drive them together:

```
import SwiftUI
struct PourValues {
  var drop = -80.0
  var squash = 1.0
  var opacity = 0.0
}
```

7. Add the choreography to the same file as an extension. Each `KeyframeTrack` targets one property through a key path and runs on a schedule of its own, so the drop can finish falling while the squash is still waiting its turn. The `@KeyframesBuilder` attribute is what lets a plain function return the tracks, which is how the same definition reaches two different consumers later:

```
extension PourValues {
  @KeyframesBuilder<PourValues>
  static func pour() -> some Keyframes<PourValues> {
    KeyframeTrack(\.drop) {
      MoveKeyframe(-80)
      CubicKeyframe(-30, duration: 0.55)
      LinearKeyframe(-30, duration: 0.45)
    }
    KeyframeTrack(\.squash) {
      LinearKeyframe(1, duration: 0.55)
      SpringKeyframe(0.35, duration: 0.15, spring: .snappy
      SpringKeyframe(1, duration: 0.3, spring: .bouncy)
    }
    KeyframeTrack(\.opacity) {
      LinearKeyframe(1, duration: 0.1)
      LinearKeyframe(1, duration: 0.6)
      LinearKeyframe(0, duration: 0.3)
```

```
    }
  }
}
```

8. Create `DropletView.swift` in the `Views` folder. It takes a `PourValues` and draws it: `drop` becomes a vertical offset, `squash` flattens the circle while stretching it sideways, and `opacity` fades it out. Nothing here knows where those values came from, which is what lets the animator in the next step and the slider at the end of the recipe drive this same view:

```
import SwiftUI
struct DropletView: View {
  let values: PourValues
  var body: some View {
    Circle()
      .fill(.blue.gradient)
      .frame(width: 20, height: 20)
      .scaleEffect(x: 2 - values.squash, y: values.squash)
      .opacity(values.opacity)
      .offset(y: values.drop)
  }
}
#Preview {
  DropletView(values: PourValues(drop: 0, squash: 1, opaci
    .padding()
}
```

9. Back in `ContentView.swift`, add a `droplet` property and place it last in the `ZStack` inside `brewStage`, so it draws above the rings and the cup. `KeyframeAnimator` takes a starting value, a closure that builds the view from the current value, and the keyframes to run. Without a `trigger:` argument it repeats for as long as it

stays in the view tree, so the `if isBrewing` check is what starts and stops the pouring:

```
struct ContentView: View {
  // ... the 3 state properties, brewDuration, body and po
  private var brewStage: some View {
    ZStack {
      // ... the two BrewArc rings and the cup image
      droplet
    }
    .frame(width: 250, height: 250)
  }
  @ViewBuilder
  private var droplet: some View {
    if isBrewing {
      KeyframeAnimator(initialValue: PourValues()) { value
        DropletView(values: values)
      } keyframes: { _ in
        PourValues.pour()
      }
    }
  }
  // ... the brewButton property, startBrew() and clearRin
}
```

10. Finish with the practice slider. A `KeyframeTimeline` holds the same choreography but plays nothing on its own, and `value(progress:)` samples it anywhere between 0 and 1. Feeding that into the same `DropletView` gives the drop a second driver, so the `else` branch shows whichever pose the slider is pointing at:

```
struct ContentView: View {
  // ... the isBrewing, progress and ringWidth state prope
  @State private var scrub = 0.0
  private let brewDuration = 6.0
  private let timeline = KeyframeTimeline(initialValue: Po
```

```
      PourValues.pour()
    }
    var body: some View {
      NavigationStack {
        VStack(spacing: 28) {
          pourBadge
          brewStage
          scrubber
          brewButton
        }
        // ... the padding and the navigation title
      }
    }
    // ... the pourBadge and brewStage computed properties
    @ViewBuilder
    private var droplet: some View {
      if isBrewing {
        // ... the KeyframeAnimator from the previous step
      } else {
        DropletView(values: timeline.value(progress: scrub))
      }
    }
    private var scrubber: some View {
      VStack(spacing: 4) {
        Slider(value: $scrub, in: 0...1)
        Text("Drag to step the drop through the pour")
          .font(.caption)
          .foregroundStyle(.secondary)
      }
      .disabled(isBrewing)
      .opacity(isBrewing ? 0.3 : 1)
    }
    // ... the brewButton property, startBrew() and clearRin
  }
```

11. Use the preview canvas in live mode to exercise all four kinds of motion. Watch the badge tip, swell, and snap back on a loop before anything else happens. Tap **Brew** and follow the ring as it sweeps and thickens while the drop falls into the cup over and

over. Let the ring fill, hold, and clear itself, then drag the slider slowly and watch that same drop move through th e same choreography one frame at a time. For the complete `ContentView`, see the book's GitHub repository.

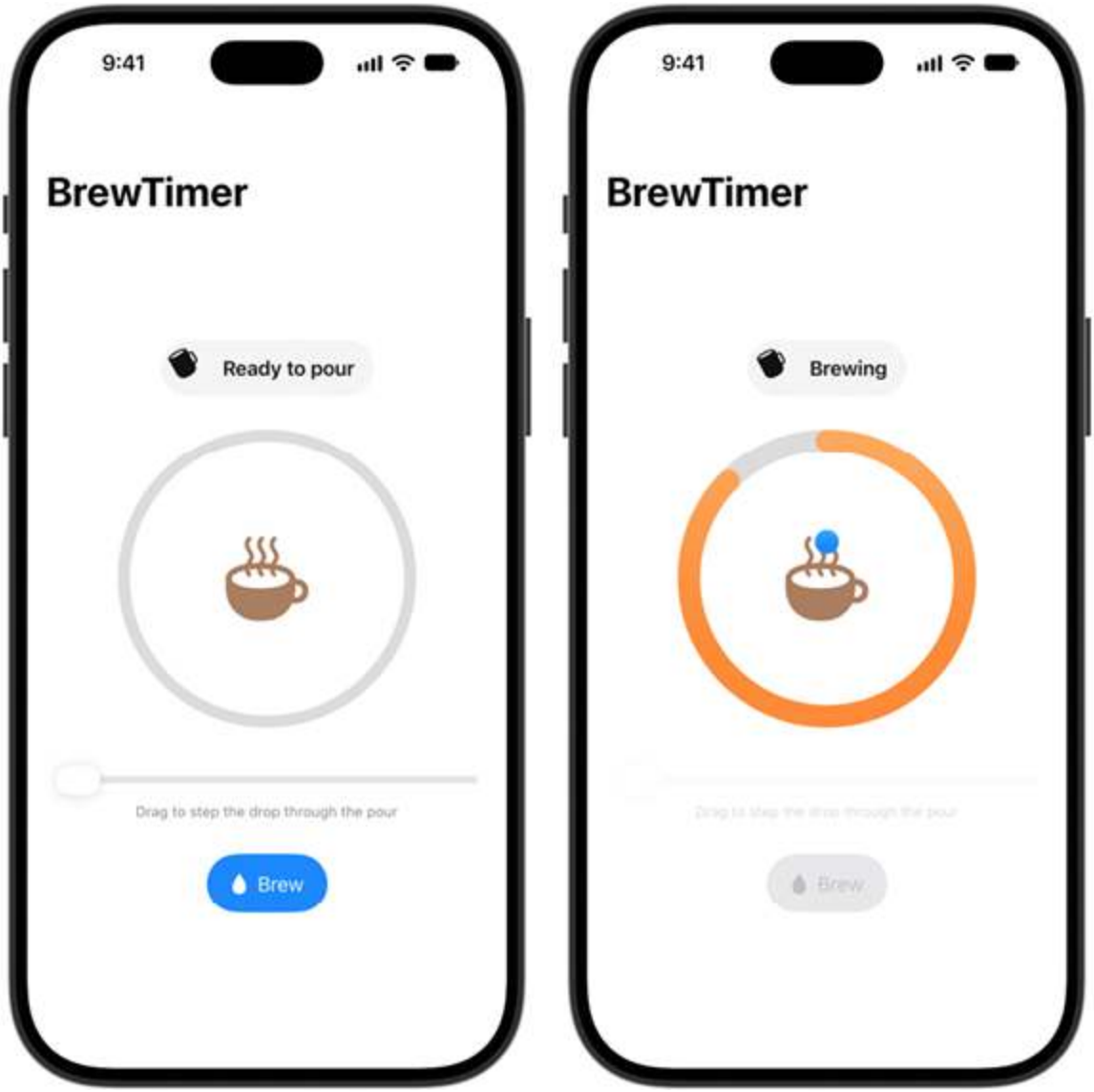


Figure 12.4: The badge and slider while idle, and the ring mid-brew with the drop falling

How it works...

The badge never touches state. `PhaseAnimator` walks its sequence by itself, animating into each phase in turn and starting the next only once the previous one has finished. That settling between phases is the defining property of the API, and it is also the thing to design around: a phase should be a pose the view could hold, not a waypoint along a path. Our `tipped` and `pouring` phases share an angle for exactly that reason, because a third phase parked halfway back to upright would stop the rotation mid-swing and read as a stutter rather than a beat. The `animation:` closure is handed the phase being **entered** , so `upright` 's snappy curve is what brings the mug home and `tipped` 's slower ease is what tips it over. There is also a `trigger:` form that runs the sequence once per change of a value, which is what to reach for when the motion should answer a tap instead of looping.

Where a phase animator knows only about poses, `KeyframeAnimator` interpolates a value continuously along a timeline we lay out ourselves. That value is our `PourValues` struct, and each `KeyframeTrack` drives one of its properties through a key path on an independent schedule. Our drop finishes falling at 0.55 seconds, which is the exact moment the squash track starts compressing, while the opacity track is still holding at full and will not begin fading until 0.7. The four keyframe types are the vocabulary for getting between values: `MoveKeyframe` jumps without animating, which is what snaps the drop back up to the spout at the start of every repeat; `LinearKeyframe` travels at a constant rate; `CubicKeyframe` accelerates and decelerates, which is what makes the fall read as gravity; and `SpringKeyframe` overshoots and settles, which is what gives the squash its rebound.

`BrewArc` is where the ring's two moving parts meet. SwiftUI interpolates a `Shape` through its `animatableData` property, and a shape that declares none gets `EmptyAnimatableData`, so it jumps between values instead of animating them. Writing that property by hand for two values means composing an `AnimatablePair` and unpacking it again in the setter, and that boilerplate is what the `@Animatable` macro removes: it collects the stored properties that can be interpolated and synthesizes `animatableData` for all of them. Marking `clockwise` with `@AnimatableIgnored` is not a stylistic choice, because a `Bool` has no meaningful midpoint; leave the attribute off and the macro reports **Cannot automatically synthesize 'animatableData'** and the file stops compiling. Custom shapes and `Path` are covered in the *Shapes, Paths, and Drawing* section.

The slider is that same choreography with the clock taken away. A `KeyframeTimeline` resolves the tracks once and then answers questions about them: `value(progress:)` maps 0 through 1 onto the whole sequence, `value(time:)` takes seconds instead, and `duration` reports how long it runs. Because `pour()` returns the keyframes rather than applying them, the animator and the timeline consume one definition and there is no second copy to drift. The brew cycle sequences itself a third way: `withAnimation` accepts a `completion:` closure that fires when its animation finishes, and that completion still fires correctly through a `.delay()`, so hanging `clearRing()` off the end of the six-second sweep runs the hold and the reset in order. The `Animation` curves themselves, and `withAnimation` in general, are covered in the *Animations, Transitions, and Matched Geometry* section.

See also

- `PhaseAnimator` documentation: https://x.swiftuicookbook.com/fb6ub
- `phaseAnimator(_:content:animation:)` documentation: https://x.swiftuicookbook.com/j7bfi
- `KeyframeAnimator` documentation: https://x.swiftuicookbook.com/513bp
- `KeyframeTrack` documentation: https://x.swiftuicookbook.com/78fra
- `KeyframeTimeline` documentation: https://x.swiftuicookbook.com/x779n
- `Animatable()` macro documentation: https://x.swiftuicookbook.com/o1qce
- Wind your way through advanced animations in SwiftUI (WWDC 2023): https://x.swiftuicookbook.com/h34fs

Glass and Material Effects

Translucent chrome is not new. `UIVisualEffectView` has been blurring whatever sits behind a toolbar since iOS 8, and SwiftUI has had `Material` since iOS 15. What arrived in iOS 26 is different in kind rather than in degree. Liquid Glass does not only blur what is behind it; it bends light around its own edges, picks up color from whatever it passes over, and merges with neighboring glass when the two come close enough. That last property is the one that changes how we arrange controls, because a glass control is no longer an independent rectangle that happens to be see-through.

In this recipe, we build an artwork viewer whose entire interface floats over the picture. Each artwork is a `MeshGradient`, which gives us the rich, multicolored surface glass needs in order to show anything at all, and `.backgroundExtensionEffect()` carries it up under the navigation bar instead of letting it stop at the safe area. The controls live inside a `GlassEffectContainer`: a pair of chevrons fused into a single shape by `.glassEffectUnion()`, and a palette button that morphs into three tint buttons through `.glassEffectID()`. Those buttons drive `Glass.tint(_:)` across every control on screen, while `Glass.interactive(_:)` makes each one respond to a press. The caption card below them deliberately stays on `.reg ularMaterial`, so both treatments sit on screen together and the difference between them is something we can look at rather than take on trust.

Getting ready

Create a new SwiftUI iOS app named **GlassGallery**.

How to do it...

Glass shows almost nothing without a busy surface behind it, so we build the artwork first and only then start floating controls over it. As in the previous recipe, each new type arrives in the step just before the one that puts it on screen. These are the steps:

1. Create `Artwork.swift` in a `Models` folder. Each artwork carries a three-color palette rather than the nine colors a mesh needs, and `meshColors` rotates those three into a 3x3 grid. That keeps the data small while still producing a surface with enough variation for glass to pick up:

```
import SwiftUI
struct Artwork: Identifiable {
  let id: String
  let title: String
  let artist: String
  let palette: [Color]
  var meshColors: [Color] {
    [palette[0], palette[1], palette[2],
     palette[1], palette[2], palette[0],
     palette[2], palette[0], palette[1]]
  }
}
extension Artwork {
  static let all: [Artwork] = [
    Artwork(id: "dusk", title: "Dusk Over Kerry", artist:
            palette: [.orange, .pink, .purple]),
    // ... the remaining three artworks
  ]
}
```

For the complete `all` array, see the book's GitHub repository.

2. Create `ArtworkBackdropView.swift` in a `Views` folder. `MeshGradient` interpolates its colors across a grid of control points, so the nine points below map onto the nine colors from the previous step. The `.backgroundExtensionEffect()` is what lets the artwork continue past the safe area: it mirrors and blurs the edges of the view into the space under the navigation bar, so glass floating up there has something to refract:

```
import SwiftUI
struct ArtworkBackdropView: View {
  let artwork: Artwork
  private let points: [SIMD2<Float>] = [
    [0, 0], [0.5, 0], [1, 0],
    [0, 0.5], [0.5, 0.5], [1, 0.5],
```

```
    [0, 1], [0.5, 1], [1, 1]
  ]
  var body: some View {
    MeshGradient(width: 3, height: 3, points: points,
                 colors: artwork.meshColors)
      .backgroundExtensionEffect()
  }
}
#Preview {
  ArtworkBackdropView(artwork: Artwork.all[0])
}
```

3. Replace the `ContentView` struct with the following. The caption card is the recipe's control group: it uses `.regularMaterial`, the treatment we would have reached for before iOS 26, and it stays that way to the end so we always have something to compare the glass against:

```
struct ContentView: View {
  @State private var index = 0
  private var artwork: Artwork { Artwork.all[index] }
  var body: some View {
    NavigationStack {
      ZStack(alignment: .bottom) {
        ArtworkBackdropView(artwork: artwork)
        captionCard
          .padding(20)
      }
      .navigationTitle("GlassGallery")
      .navigationBarTitleDisplayMode(.inline)
    }
  }
  private var captionCard: some View {
    VStack(alignment: .leading, spacing: 4) {
      Text(artwork.title)
        .font(.headline)
      Text(artwork.artist)
        .font(.subheadline)
        .foregroundStyle(.secondary)
```

```
    }
    .frame(maxWidth: .infinity, alignment: .leading)
    .padding(16)
    .background(.regularMaterial, in: .rect(cornerRadius:
  }
}
```

In the preview canvas, notice how the artwork runs all the way up behind the navigation title rather than stopping below it.

4. Add the controls that step between artworks. `glassButton` is a small factory so every control gets the same treatment from one place, and `.glassEffect()` with no argument applies the standard `.regular` glass in its default capsule shape. The step changes three places. In `body`, wrap the caption card in a `VStack` with `controls` above it, which moves the padding up to the pair. Then add `controls` and `glassButton(_:action:)` before `captionCard`, and `step(_:)` after it:

```
struct ContentView: View {
  // ... the index property and artwork
  var body: some View {
    NavigationStack {
      ZStack(alignment: .bottom) {
        ArtworkBackdropView(artwork: artwork)
        VStack(spacing: 20) {
          controls
          captionCard
        }
        .padding(20)
      }
      // ... the navigation title and display mode
    }
  }
  private var controls: some View {
    HStack(spacing: 24) {
```

```
        glassButton("chevron.left") { step(-1) }
        glassButton("chevron.right") { step(1) }
      }
    }
    private func glassButton(_ symbol: String,
                             action: @escaping () -> Void) -
      Button(action: action) {
        Image(systemName: symbol)
          .font(.title3)
          .frame(width: 28, height: 28)
          .padding(12)
      }
      .buttonStyle(.plain)
      .glassEffect()
    }
    // ... the captionCard computed property
    private func step(_ delta: Int) {
      withAnimation(.smooth) {
        index = (index + delta + Artwork.all.count) % Artwor
      }
    }
}
```

Two separate glass circles should now float above the card. Step through the artworks and watch the glass take its color from whichever part of the mesh is behind it.

5. Now fuse those two circles into one shape. A `GlassEffectContainer` is what allows its glass children to interact at all, and `spacing:` sets how close two shapes must be before they start to merge. `.glassEffectUnion()` goes further and joins shapes unconditionally: give two views the same union `id` in a shared `@Namespace` and they render as a single piece of glass no matter how far apart they sit:

```
struct ContentView: View {
  // ... the index property
  @Namespace private var glass
  // ... the artwork computed property
  // ... the body
  private var controls: some View {
    GlassEffectContainer(spacing: 24) {
      HStack(spacing: 24) {
        glassButton("chevron.left") { step(-1) }
          .glassEffectUnion(id: "stepper", namespace: glas
        glassButton("chevron.right") { step(1) }
          .glassEffectUnion(id: "stepper", namespace: glas
      }
    }
  }
  // ... glassButton, captionCard and step()
}
```

The two circles are now a single capsule with both chevrons inside it.

6. Create `GlassTint.swift` in the `Models` folder. The `color` is optional because no tint at all is a real choice rather than a missing one, and `Glass.tint(_:)` accepts an optional `Color` for exactly that reason. Both tints carry an alpha instead of being fully opaque, which turns out to matter for more than appearance:

```
import SwiftUI
enum GlassTint: String, CaseIterable, Identifiable {
  case none = "None"
  case warm = "Warm"
  case cool = "Cool"
  var id: String { rawValue }
  var color: Color? {
    switch self {
    case .none: nil
    case .warm: .orange.opacity(0.35)
```

```
    case .cool: .cyan.opacity(0.35)
    }
  }
  var symbol: String {
    switch self {
    case .none: "circle.slash"
    case .warm: "sun.max.fill"
    case .cool: "snowflake"
    }
  }
}
```

7. Route every button through a configured `Glass` value. `Glass` is a value type with chainable methods, so `.regular.tint(...).interactive()` builds up one description that `style` hands to every control. The `.interactive()` part is what makes the glass flex and brighten under a finger:

```
struct ContentView: View {
  // ... the index property
  @State private var tint: GlassTint = .none
  // ... the glass namespace and the artwork property
  private var style: Glass { .regular.tint(tint.color).int
  // ... the body and the controls computed property
  private func glassButton(_ symbol: String,
                           action: @escaping () -> Void) -
    Button(action: action) {
      Image(systemName: symbol)
        .font(.title3)
        .frame(width: 28, height: 28)
        .padding(12)
    }
    .buttonStyle(.plain)
    .glassEffect(style)
  }
  // ... the captionCard property and step()
}
```

8. Finish the row of controls with the tint buttons. When `showsTints` flips, the single palette button is replaced by three, and because each shape carries a `.glassEffectID()` in the same namespace, the container morphs one piece of glass into three rather than fading one set out and another in:

```
struct ContentView: View {
  // ... the index property
  @State private var showsTints = false
  // ... the tint property, glass namespace, artwork and s
  // ... the body
  private var controls: some View {
    GlassEffectContainer(spacing: 24) {
      HStack(spacing: 24) {
        // ... the two stepper buttons
        if showsTints {
          ForEach(GlassTint.allCases) { choice in
            glassButton(choice.symbol) { pick(choice) }
              .glassEffectID(choice.id, in: glass)
          }
        } else {
          glassButton("paintpalette") { toggleTints() }
            .glassEffectID("palette", in: glass)
        }
      }
    }
  }
  // ... glassButton, captionCard and step()
  private func toggleTints() {
    withAnimation(.smooth) {
      showsTints = true
    }
  }
  private func pick(_ choice: GlassTint) {
    withAnimation(.smooth) {
      tint = choice
      showsTints = false
    }


  }
```

```
    }
}
```

9. Exercise the finished app in the preview canvas using live mode. Tap the palette icon and watch one piece of glass flow into three, then pick the sun icon and notice that all the glass controls change their tint while the caption card underneath does not. Over the blue artworks, a warm tint reads closer to olive than to orange, because the tint mixes with what is behind the control instead of painting over it. Step through the four artworks and watch the glass recolor itself from the mesh behind it, and press and hold a chevron to feel the interactive flex. For the c omplete `ContentView`, see the book's GitHub repository.



Figure 12.5: The same controls untinted and warm-tinted; the tint mixes with the artwork behind the glass, while the material caption card below is identical in both

How it works…

The two treatments in our app answer different questions. `.regularMaterial` on the caption card asks what is behind this rectangle, blurs it, and blends the result toward the system background, which is why the card reads as the same neutral panel

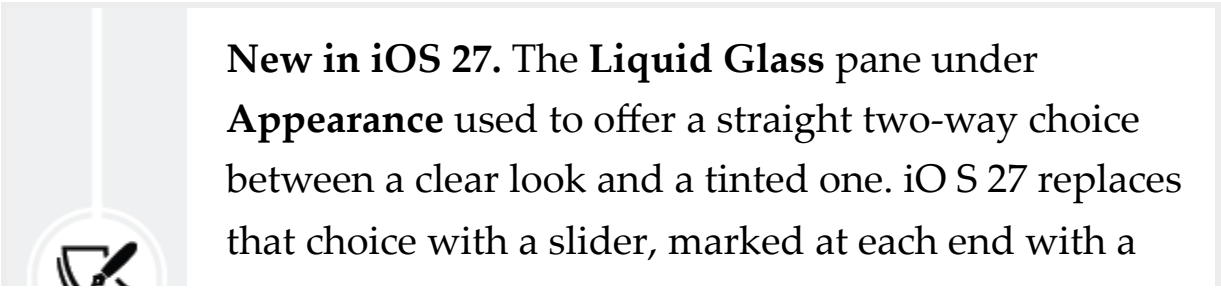
over every artwork. `.glassEffect()` asks a richer question: it refracts the mesh through the shape's edges, tints itself from what it passes over, and casts a specular highlight along its rim. Materials come in a fixed set from `.ultraThinMaterial` through `.ultraThickMaterial`, chosen by how much of the backdrop should survive. Glass is configured instead of chosen: `Glass` is a value type whose `.regular` and `.clear` starting points take chainable modifiers, so `.regular.tint(tint.color).interactive()` in our `style` property is one description built in three steps and handed to every control.

`GlassEffectContainer` is the piece that surprises people, because without it the glass modifiers still render but the shapes cannot see one another. Inside a container they can, and there are two ways to make that happen. The `spacing:` argument sets a proximity threshold: shapes closer than that distance begin to flow together on their own, which is what produces the liquid merging effect as controls move. `.glassEffectUnion()` is the explicit form, joining any shapes that share an `id` within a `@Namespace` regardless of distance, which is how our two chevrons became one capsule while the palette button beside them stayed separate. `.glassEffectID()` solves a third problem, identity across a change: our palette button and the three tint buttons that replace it are different views, and the shared namespace is what tells the container to morph the glass rather than cross-fade it. If that sounds familiar, it is the same idea as `.matchedGeometryEffect()` from the *Animations, Transitions, and Matched Geometry* section, applied to glass shapes instead of view frames.

Glass near the navigation bar has a problem: the artwork stops at the safe area, so there is nothing up there to refract but the window background. `.backgroundExtensionEffect()` solves it by mirroring and

blurring the view's own edges outward into that space, leaving the original content untouched inside the safe area. Apple's own examples put it on the detail column of a `NavigationSplitView`, but nothing about it is iPad-only: on our single-column iPhone layout it fills the status bar and navigation bar region with the artwork, which is exactly the surface the controls need. One habit worth forming early: glass is only as legible as what sits behind it, so a control that looks perfect over a dark corner of one artwork can wash out entirely over a pale one. Testing across the full set is not optional polish, it is how we find out whether the design works at all.

How transparent that glass ends up looking is not entirely our decision. Under **Settings**, **Appearance**, **Liquid Glass**, the person using the device picks their preferred look, and the **Reduce Transparency** and **Increase Contrast** switches under **Accessibility**, **Display & Text Size** override that pane. Our app reads none of this and should not try to, because the preference reaches `.glassEffect()` on its own while `.regularMaterial` on the caption card ignores it completely. It does explain why our two tints carry an alpha rather than being solid colors. A fully opaque tint fills the glass, which stops it sampling the artwork underneath and, more to the point, largely overrides whatever they picked there. Tinting with a translucent color leaves both the artwork and their preference visible through the control.



New in iOS 27. The **Liquid Glass** pane under **Appearance** used to offer a straight two-way choice between a clear look and a tinted one. iO S 27 replaces that choice with a slider, marked at each end with a



glass symbol rather than a caption, and puts a live sample above it so the effect can be judged while the slider moves. The accessibility switches are unchanged.

See also

- `glassEffect(_:in:)` documentation: https://x.swiftuicookbook.com/sy9r7
- `Glass` documentation: https://x.swiftuicookbook.com/2262h
- `GlassEffectContainer` documentation: https://x.swiftuicookbook.com/susvq
- `glassEffectUnion(id:namespace:)` documentation: https://x.swiftuicookbook.com/41lt0
- `backgroundExtensionEffect()` documentation: https://x.swiftuicookbook.com/myqwi
- `Material` documentation: https://x.swiftuicookbook.com/n393c
- Build a SwiftUI app with the new design (WWDC 2025): https://x.swiftuicookbook.com/wliha

13

Swift Charts

Every app I have worked on eventually grew a screen whose job was to make numbers legible at a glance: a sales trend, a sensor log, a week of workouts. Tables answer the questions we already know to ask, while a chart surfaces the pattern we did not know was there, which is why those screens earn their place. Swift Charts treats this as a first-class problem: built on SwiftUI's declarative grammar, it lets us describe each data series as marks inside a `Chart` view and derives the scales, axes, and legends from the data itself. The five recipes in this chapter climb from that grammar to interaction, to plots defined by functions rather than data, and finally into the third dimension.

We begin with a climate browser that draws the same year of rainfall and temperatures for three cities five different ways, meeting `Chart` and the mark vocabulary: `BarMark`, `LineMark`, `PointMark`, `AreaMark`, and `RectangleMark`, with `.foregroundStyle(by:)` telling the cities apart by color and `.position(by:)` setting their bars side by side. The same app then stops using defaults: we rebuild both axes with `AxisMarks`, `AxisValueLabel`, `AxisGridLine`, and `AxisTick`, pin the value range with `.chartYScale()`, choose the city colors with `.chartForegroundStyleScale()`, move the legend with `.chartLegend()`, tint the plot area with `.chartPlotStyle()`, and mark the annual mean and the wettest month with labeled `RuleMark` annotations.

An activity browser follows, spreading four months of daily step counts across two charts: `.chartXSelection()` lets a press pick out a day whose numbers appear in an `.annotation()` that its overflow resolution keeps inside the plot, while `.chartScrollableAxes()` , `.chartXVisibleDomain()` , and `.chartScrollPosition()` scroll a thirty-day window across the full period. Then the data itself becomes optional: the function plots take a closure instead of an array, with `LinePlot` drawing a sine curve that follows its amplitude and frequency sliders live, its parametric form drawing a spiral that no function of x could produce, `AreaPlot` filling the region under the curve, and the vectorized `PointPlot` reading 400 sensor readings through key paths instead of building a mark per element. The chapter closes in three dimensions with `Chart3D` : a `SurfacePlot` samples a terrain function into a rotatable surface colored by elevation, `PointMark` gains a `z` argument that pins survey stations onto it, and a `Chart3DPose` binding with `.chart3DCameraProjection()` hands the camera to the user.

By the end of this chapter, you should be able to choose the right mark for a comparison, take over exactly the parts of a chart where the defaults do not fit your needs, decide when interactivity earns its complexity, and judge when a dataset deserves a third axis rather than a second chart.

In this chapter, we will cover the following recipes:

- Basic Charts
- Customizing Charts
- Interactive Swift Charts
- Vectorized and Function Plots
- 3D Charts

Technical requirements

The code in this chapter is based on Xcode 27.0 and iOS 27.0. You can download and install the latest version of Xcode from the App Store. All the code examples for this chapter can be found on GitHub at https://x.swiftuicookbook.com/ch13 .

Basic Charts

Before Swift Charts, putting a chart in an iOS app meant one of two unappealing choices: pull in a third-party library and inherit its release schedule, or draw the chart by hand with Core Graphics and own every tick, label, and scale calculation forever. Both options existed because the iOS SDK included no charting API. Swift Charts removed that limitation by making a chart a view we declare the same way we declare a `List` or a `VStack` .

In this recipe, we build a climate browser for three cities with genuinely different weather, and then draw the same twelve months of data five different ways. Rainfall becomes bars, temperatures become curved lines, the two are plotted against each other as a scatter, and the year appears again as filled areas and as shaded high-to-low ranges. Working from one dataset is what makes the comparison useful: each chart type presents the same numbers in a different way, and switching between them shows which comparisons each one makes easy.

One piece of vocabulary is worth settling first, because it runs through the whole chapter. A mark is a single drawn element of a chart: one bar, one line, one point. Swift Charts borrows the term from the grammar of graphics, the vocabulary that visualization tools such as Tableau and Vega-Lite share. In code it names the types we write inside a chart, such as `BarMark` and `LineMark` .

Getting ready

Create a new SwiftUI iOS app named **CityClimate** .

How to do it...

We will build the data layer first, then draw a single chart directly in `ContentView` so there is something on screen within two steps. Once we have a chart worth keeping, we move it into its own view and grow it one mark type at a time, which keeps every intermediate state runnable. The picker that ties them together comes last, because it can only be written once every chart it selects exists. These are the steps:

1. Create `ClimateRecord.swift` in a `Models` folder. Each record is one city in one month, which is the shape Swift Charts wants: one element per mark it will draw. The month is stored as an `Int` so the data literals stay short, and `date` converts it on demand, because charts plot dates on a proper time axis and integers on a numeric one:

```
import Foundation
struct ClimateRecord: Identifiable {
  let city: String
  let month: Int
  let rainfall: Double
  let high: Double
  let low: Double
  var id: String { "\(city)-\(month)" }
  var date: Date {
    DateComponents(calendar: .current, year: 2025, month: month)
  }
}
extension ClimateRecord {
  static let all: [ClimateRecord] = [
    ClimateRecord(city: "Seattle", month: 1, rainfall: 140, high
    // ... the remaining 35 records
  ]
}
```

Seattle, Phoenix, and Miami are in the data because they disagree with each other: wet and cool, dry and hot, wet and hot. For the complete `all` array, see the book's GitHub repository.

2. Replace the `ContentView` struct with the following, where the whole chart is four lines of content. `Chart` takes our collection directly, the way `List` does, and the closure returns one mark per element. Note the `import Charts` above it, which the Xcode template does not give us:

```
import SwiftUI
import Charts
struct ContentView: View {
  var body: some View {
    Chart(ClimateRecord.all) { record in
      BarMark(
        x: .value("Month", record.date, unit: .month),
        y: .value("Rainfall", record.rainfall)
      )
    }
    .padding()
  }
}
```

   Look at the preview canvas. There are twelve bars, not the thirty-six that three cities across twelve months would suggest. All three cities are in the data, and the chart is stacking them into one bar per month, because nothing has told it yet that the city is a dimension worth separating.

3. Give the city a visual role by adding `.foregroundStyle(by:)` to the mark. This is the first of the `by:` modifiers, and it does two jobs at once: it colors the stacked segments that were already there, and it generates the legend at the bottom:

```
struct ContentView: View {
  var body: some View {
```

```
    Chart(ClimateRecord.all) { record in
      BarMark(
        x: .value("Month", record.date, unit: .month),
        y: .value("Rainfall", record.rainfall)
      )
      .foregroundStyle(by: .value("City", record.city))
    }
    .padding()
  }
}
```

4. Stacked segments are hard to compare across cities, since only the bottom one starts at zero. Add `.position(by:)` to move each city sideways into its own bar instead:

```
struct ContentView: View {
  var body: some View {
    Chart(ClimateRecord.all) { record in
      // ... the BarMark with its x and y values
      .foregroundStyle(by: .value("City", record.city))
      .position(by: .value("City", record.city))
    }
    .padding()
  }
}
```

5. The chart is worth keeping now, so move it somewhere it can grow. Create `ClimateChartView.swift` in a `Views` folder, with the bar chart as a computed property rather than as the `body` directly. That looks like indirection for one chart, but every mark type we add from here becomes a sibling property, and `body` becomes the one place that chooses between them:

```
import SwiftUI
import Charts
struct ClimateChartView: View {
  var body: some View {
    rainfallChart
  }
```

```
  private var rainfallChart: some View {
    Chart(ClimateRecord.all) { record in
      BarMark(
        x: .value("Month", record.date, unit: .month),
        y: .value("Rainfall", record.rainfall)
      )
      .foregroundStyle(by: .value("City", record.city))
      .position(by: .value("City", record.city))
    }
  }
}
#Preview {
  ClimateChartView()
    .padding()
}
```

6. Back in `ContentView.swift`, replace the `ContentView` struct with a shell that hosts the chart. The `import Charts` line goes away with the chart itself, since `ContentView` no longer names a Swift Charts type:

```
import SwiftUI
struct ContentView: View {
  var body: some View {
    NavigationStack {
      VStack {
        ClimateChartView()
      }
      .padding()
      .navigationTitle("City Climate")
    }
  }
}
```

7. In `ClimateChartView.swift`, add a temperature chart and point `body` at it instead of `rainfallChart` so we can see what we are building. `LineMark` connects its points in order instead of standing them on the axis, `.symbol(by:)` gives each city a distinct marker so the three series stay apart without relying on color, and `.interpolationMethod(.catmullRom)`

curves the segments through the points rather than joining them with straight lines:

```
struct ClimateChartView: View {
  var body: some View {
    temperatureChart
  }
  // ... the rainfallChart property
  private var temperatureChart: some View {
    Chart(ClimateRecord.all) { record in
      LineMark(
        x: .value("Month", record.date, unit: .month),
        y: .value("High", record.high)
      )
      .foregroundStyle(by: .value("City", record.city))
      .symbol(by: .value("City", record.city))
      .interpolationMethod(.catmullRom)
    }
  }
}
```

8. Three lines with no reference points make it hard to say which city is actually comfortable. Replace the `temperatureChart` property with the following, which puts a shaded band behind the lines for the range most people find pleasant. `RectangleMark` given only a `yStart` and a `yEnd` spans the full width of the plot, and adding it means restructuring the chart: the data-driven `Chart(_:)` initializer draws one mark per record, which would stack thirty-six identical rectangles, so we switch to the empty `Chart` initializer and put the records in an explicit `ForEach`:

```
  private var temperatureChart: some View {
    Chart {
      RectangleMark(
        yStart: .value("Comfort low", 18),
        yEnd: .value("Comfort high", 26)
      )
      .foregroundStyle(.green.opacity(0.15))
```

```
        ForEach(ClimateRecord.all) { record in
          LineMark(
            x: .value("Month", record.date, unit: .month),
            y: .value("High", record.high)
          )
          .foregroundStyle(by: .value("City", record.city))
          .symbol(by: .value("City", record.city))
          .interpolationMethod(.catmullRom)
        }
      }
    }
```

9. So far every chart has put time on the x-axis. Add one that does not, plotting rainfall against temperature to ask whether the hot months are the wet ones. `PointMark` places a single marker per record, and dropping the month entirely turns twelve ordered readings per city into an unordered cloud:

```
struct ClimateChartView: View {
  var body: some View {
    scatterChart
  }
  // ... the rainfallChart and temperatureChart properties
  private var scatterChart: some View {
    Chart(ClimateRecord.all) { record in
      PointMark(
        x: .value("High", record.high),
        y: .value("Rainfall", record.rainfall)
      )
      .foregroundStyle(by: .value("City", record.city))
      .symbol(by: .value("City", record.city))
    }
  }
}
```

The three cities land in three separate clusters, and each answers the question differently: Seattle's wettest months are its coldest, Miami's are its hottest, and Phoenix stays dry at every temperature. That

reversal between Seattle and Miami is the point, and a chart of any one city on its own could not have shown it.

10. Add an area chart of the same rainfall data the bars used. `AreaMark` fills everything under its line, and with a `by:` style scale in play it stacks the cities on top of each other, so the outline traces the three-city total rather than any one of them:

```
struct ClimateChartView: View {
  var body: some View {
    areaChart
  }
  // ... the rainfallChart, temperatureChart and scatterChart pr
  private var areaChart: some View {
    Chart(ClimateRecord.all) { record in
      AreaMark(
        x: .value("Month", record.date, unit: .month),
        y: .value("Rainfall", record.rainfall)
      )
      .foregroundStyle(by: .value("City", record.city))
    }
  }
}
```

11. Finish the set with the one chart that uses both temperature columns. Giving `AreaMark` a `yStart` and a `yEnd` instead of a single `y` floats the band between the daily low and the daily high, and the reduced opacity lets the three cities overlap without hiding each other:

```
struct ClimateChartView: View {
  var body: some View {
    rangeChart
  }
  // ... the four existing chart properties
  private var rangeChart: some View {
    Chart(ClimateRecord.all) { record in
      AreaMark(
        x: .value("Month", record.date, unit: .month),
        yStart: .value("Low", record.low),
```

```
            yEnd: .value("High", record.high)
        )
        .foregroundStyle(by: .value("City", record.city))
        .opacity(0.6)
      }
    }
  }
```

12. Now that every chart exists, we can finally name them. Create `ChartStyle.swift` in the `Models` folder. The raw values are the picker's segment labels, which is why they are abbreviated, and `Identifiable` is what lets `ForEach` iterate the cases directly:

```
import Foundation
enum ChartStyle: String, CaseIterable, Identifiable {
  case rainfall = "Rain"
  case temperature = "Temp"
  case scatter = "Scatter"
  case area = "Area"
  case range = "Range"
  var id: Self { self }
}
```

13. In `ClimateChartView.swift`, accept a style and switch on it. This is why each chart went into its own property: `body` becomes a five-line routing table, and the compiler now guarantees that every case has a chart to show. The property defaults to `.rainfall`, so the view draws the rainfall chart unless a caller asks for another one, which also means the `ClimateChartView()` call already sitting in `ContentView` keeps working. The preview changes too: passing `ChartStyle.allCases` to the `arguments:` parameter, which we encountered in the *Trait Variations and Device Simulations* section of *Chapter 3*, renders all five charts in the preview canvas at once:

```
struct ClimateChartView: View {
  var style: ChartStyle = .rainfall
```

```
  var body: some View {
    switch style {
    case .rainfall: rainfallChart
    case .temperature: temperatureChart
    case .scatter: scatterChart
    case .area: areaChart
    case .range: rangeChart
    }
  }
  // ... the five chart properties
}
#Preview(arguments: ChartStyle.allCases) { style in
  ClimateChartView(style: style)
    .padding()
}
```

14. Replace the `ContentView` struct with the finished version, which holds the selected style in a `@State` property and passes it to `ClimateChartView` through the `style` property we added in the previous step. The chart takes no explicit height, so it expands into whatever the `VStack` has left after the picker, and a segmented `Picker` keeps all five choices visible at once:

```
import SwiftUI
struct ContentView: View {
  @State private var style: ChartStyle = .rainfall
  var body: some View {
    NavigationStack {
      VStack {
        ClimateChartView(style: style)
        Picker("Chart style", selection: $style) {
          ForEach(ChartStyle.allCases) { option in
            Text(option.rawValue).tag(option)
          }
        }
        .pickerStyle(.segmented)
      }
      .padding()
      .navigationTitle("City Climate")
    }
  }
}
```

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

15. With the preview canvas in live mode, move through all five segments and watch the same thirty-six records take five different shapes. Notice how **Rain** and **Area** plot identical numbers while showing different things: the grouped bars give each city its own baseline so the cities can be compared against each other, while the stacked areas add the three together into a single outline. **Temp** is where the comfort band earns its place, with Seattle reaching it only in midsummer while Phoenix crosses it twice on the way to and from a much hotter peak, and **Scatter** is the only one where Phoenix and Miami stop overlapping. For the complete app code, see the book's GitHub repository.

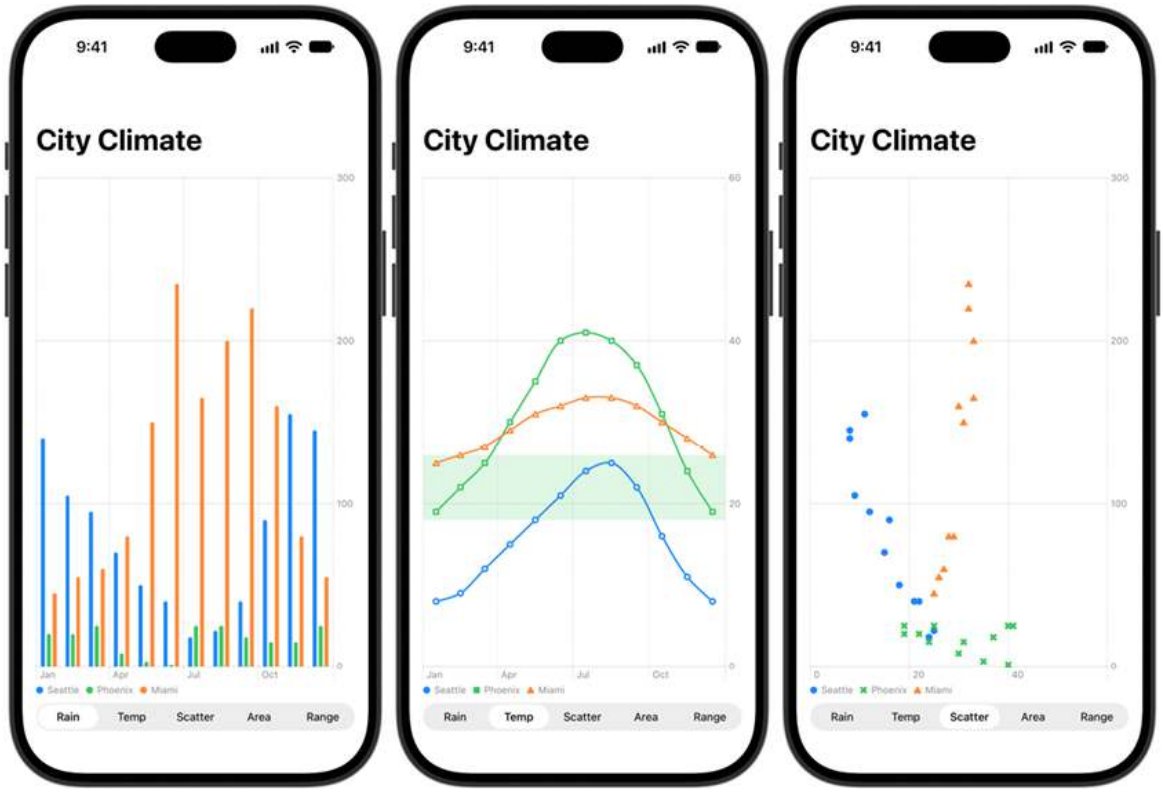


Figure 13.1: The rainfall, temperature, and scatter charts drawn from t he same records

How it works…

Marks are to a `Chart` what rows are to a `List`: its content. That parallel is worth taking literally, because it explains the two initializers we used. `Chart(ClimateRecord.all)` is the data-driven form, and like `List(_:)` it runs its closure once per element, which is why a `RectangleMark` written inside it would have been drawn thirty-six times. The empty `Chart { }` form takes static content instead, so our comfort band is stated once and the records go through an explicit `ForEach`. Reach for the data-driven form when every mark comes from the data, and the explicit form as soon as one does not.

Our axes and legend get their readable labels from the `.value(_:_:)` calls: each one builds a `PlottableValue` pairing a display name with a value, and Swift Charts uses that name in the axis label, the legend, and the accessibility description, which is why they read as prose rather than as property names. The type of the value decides the axis: our `date` produces a time axis that thins itself to Jan, Apr, Jul, and Oct rather than crowding twelve labels onto a phone, while `rainfall` produces a numeric one starting at zero. The `unit: .month` argument tells the framework the granularity the value really has, so records are binned by month instead of by their exact timestamp.

The `by:` modifiers are the part that repays study, because they do not set an appearance, they declare a grouping. `.foregroundStyle(by: .value("City", record.city))` says the city is a dimension of the data, and the framework responds by building a scale over the distinct cities, assigning each a color, generating a legend, and (for bars and areas) stacking the groups. Our step 2 chart showed twelve bars for thirty-six records precisely because no such dimension existed yet, so the framework had nothing to separate the overlapping values by and summed them. `.position(by:)` takes the same grouping and spends it on horizontal space rather than

color, which is the whole difference between a stacked bar chart and a grouped one. `.symbol(by:)` spends it on marker shape, and pairing it with `.foregroundStyle(by:)` is a habit worth keeping: a chart that encodes a dimension twice still reads in grayscale, and the printed version of this book is the obvious example.

Choosing between the mark types is choosing what to claim about the numbers. `BarMark` measures from zero and compares magnitudes, so it suits rainfall totals. `LineMark` asserts that consecutive points are connected, which is true across months and would be meaningless across cities; `.interpolationMethod(.catmullRom)` curves those connections, and it is worth remembering that the curve is a drawing choice and not additional data. `AreaMark` fills to a baseline and, with a grouping scale, stacks, so our area chart is a statement about combined rainfall rather than three separate cities. `PointMark` makes no claim about order at all, which is what let it drop the month and compare two measurements directly. `RectangleMark` covers a region rather than a value, which suits a threshold band, and it is also the mark to reach for when building a heatmap. Swift Charts has one more mark we did not need here, `SectorMark`, which draws the wedges of pie and donut charts; it is the right tool for parts of a whole at a single moment, and the wrong one for anything with an axis.

What we have not touched is that almost everything on screen was chosen for us: the axis ranges, the tick positions, the date format, the legend placement, and the colors. Those defaults are good enough that a chart is useful within two steps, which is the framework's real achievement. They are also just defaults, and the moment a chart needs a specific unit in its labels or a marked average, we have to take them over. That is the subject of the *Customiz ing Charts* section.

See also

- `Chart` documentation: https://x.swiftuicookbook.com/yklpv
- `BarMark` documentation: https://x.swiftuicookbook.com/lvg9z
- `AreaMark` documentation: https://x.swiftuicookbook.com/hj81c
- `SectorMark` documentation: https://x.swiftuicookbook.com/ugkdg
- *Creating a chart using Swift Charts* article: https://x.swiftuicookbook.com/zr0yn
- Hello Swift Charts (WWDC 2022): https://x.swiftuicookbook.com/1t89j

Customizing Charts

Defaults are a good place to start and a poor place to stop. This recipe continues the app built in the previous recipe, so we recommend working through that one first. The chart we finish here is its rainfall chart, from the *Basic Charts* section, which draws a year of data for three cities and reads well enough at a glance but cannot answer the first question anyone actually asks of it: which months are unusually wet? Its numbers carry no units, so a value of 200 could be millimeters or inches, and it labels four months out of twelve, so finding September means counting.

We finish that chart while leaving the other four on their defaults. That asymmetry is the point rather than an oversight: the picker turns into a before-and-after control, so the same thirty-six records can be seen customized and untouched a tap apart. We take over both axes, pin the scale so it stops moving with the data, choose the three city colors instead of accepting the ones we were given, move the legend, tint the plot area, and finally mark the annual mean and the wettest month with labe led reference marks.

Getting ready

Since the chart we customize here is the one built in the *Basic Charts* section, we have already provided it in a starter project so you can focus on the customization APIs. Download or clone the code for this chapter from GitHub as explained in the *Technical requirements* section, and open the **CityClimate** project located in the **Starter** fo lder of this recipe.

How to do it...

The axes come first, because until the values can be read accurately there is no way to judge any of the choices that follow. The scale and the colors come next, then the chrome around the plot, and last the reference marks that call out individual values. Every step until the final two edits the `rainfallChart` property and leaves the other four charts untouched. These are the steps:

1. Open `ClimateChartView.swift` and add a `.chartXAxis` modifier to the `rainfallChart` property. `AxisMarks(values:)` decides where marks are placed, and `.stride(by: .month)` asks for one per month rather than letting the framework thin them to whatever fits. The narrow date format renders a single letter per month, which is what allows all twelve to sit side by side on a phone:

```
struct ClimateChartView: View {
  // ... the style property and the body
  private var rainfallChart: some View {
    Chart(ClimateRecord.all) { record in
      // ... the BarMark with its foregroundStyle and position m
    }
    .chartXAxis {
      AxisMarks(values: .stride(by: .month)) { _ in
        AxisGridLine()
        AxisTick()
        AxisValueLabel(format: .dateTime.month(.narrow))
      }
```

```
    }
  }
  // ... the other four charts
}
```

Naming `AxisGridLine` and `AxisTick` explicitly is not optional here. The moment we supply our own content closure, we are responsible for everything the axis draws, so anything we leave out disappears. The ticks are the easiest part to miss: each one is the short stub that extends a gridline below the plot, beside its month letter, and commenting out `AxisTick()` removes them.

2. Now take over the y-axis, where the missing unit lives. Asking for a mark every 50 millimeters would crowd the labels, so we branch on the value: multiples of 100 get a full mark with a labeled value, and everything between gets a faint dashed line only. This is the conditional axis pattern, and it is how a chart gets fine gridlines without a wall of numbers:

```
struct ClimateChartView: View {
  // ... the style property and the body
  private var rainfallChart: some View {
    Chart(ClimateRecord.all) { record in
      // ... the BarMark with its foregroundStyle and position m
    }
    // ... the chartXAxis modifier
    .chartYAxis {
      AxisMarks(values: .stride(by: 50)) { value in
        if let mm = value.as(Double.self),
            mm.truncatingRemainder(dividingBy: 100) == 0 {
          AxisGridLine()
          AxisTick()
          AxisValueLabel("\(Int(mm)) mm")
        } else {
          AxisGridLine(stroke: StrokeStyle(lineWidth: 0.5, dash
        }
      }
    }
```

```
    }
    // ... the other four charts
  }
```

`value.as(Double.self)` is how we get the plotted value back out of an `AxisValue`, and it is optional because the axis cannot know in advance which type we plotted.

3. Pin the y-axis range with `.chartYScale(domain:)`. Until now the chart has sized itself to the data, which is convenient while exploring and unhelpful in a finished view, because the same bar height means a different number whenever the data changes. A fixed domain also leaves headroom above the tallest bar for the annotation we add later:

```
struct ClimateChartView: View {
  // ... the style property and the body
  private var rainfallChart: some View {
    Chart(ClimateRecord.all) { record in
      // ... the BarMark with its foregroundStyle and position m
    }
    // ... the chartXAxis and chartYAxis modifiers
    .chartYScale(domain: 0...260)
  }
  // ... the other four charts
}
```

4. Assign the city colors deliberately with `.chartForegroundStyleScale()`. So far the colors have come from the order the cities happen to appear in the data, which means an edit to the array could silently repaint the chart. Passing a dictionary ties each city to a color that will not move:

```
struct ClimateChartView: View {
  // ... the style property and the body
  private var rainfallChart: some View {
    Chart(ClimateRecord.all) { record in
```

```
      // ... the BarMark with its foregroundStyle and position m
    }
    // ... the axis and scale modifiers
    .chartForegroundStyleScale([
      "Seattle": .teal,
      "Phoenix": .orange,
      "Miami": .indigo
    ])
  }
  // ... the other four charts
}
```

5. Finish the chrome around the plot. Moving the legend to the top puts the color key above the data instead of below it, where it competes with the axis labels, and a faint plot background separates the drawing area from the rest of the view:

```
struct ClimateChartView: View {
  // ... the style property and the body
  private var rainfallChart: some View {
    Chart(ClimateRecord.all) { record in
      // ... the BarMark with its foregroundStyle and position m
    }
    // ... the axis, scale, and foreground style modifiers
    .chartLegend(position: .top, alignment: .leading, spacing: 1
    .chartPlotStyle { plotArea in
      plotArea.background(.quaternary.opacity(0.25))
    }
  }
  // ... the other four charts
}
```

6. The chart can now be read accurately, but it still says nothing about the data. Open `ClimateRecord.swift` and add two summary values at the bottom of the file, which the next steps draw on the chart. Both are computed from `all`, so neither can fall out of step with the records:

```
extension ClimateRecord {
  static var meanRainfall: Double {
    all.map(\.rainfall).reduce(0, +) / Double(all.count)
  }
  static var wettest: ClimateRecord {
    all.max { $0.rainfall < $1.rainfall } ?? all[0]
  }
}
```

7. Back in `ClimateChartView.swift`, draw the mean as a dashed reference line. `RuleMark` given only a `y` value spans the full width of the plot, and because it is a single mark rather than one per record, the chart has to be restructured the same way the comfort band did in the previous section: the records move into an explicit `ForEach` inside the empty `Chart` initializer:

```
struct ClimateChartView: View {
  // ... the style property and the body
  private var rainfallChart: some View {
    Chart {
      ForEach(ClimateRecord.all) { record in
        // ... the BarMark with its foregroundStyle and positior
      }
      RuleMark(y: .value("Average", ClimateRecord.meanRainfall)
        .foregroundStyle(.secondary)
        .lineStyle(StrokeStyle(lineWidth: 1, dash: [4]))
    }
    // ... the axis, scale, legend, and plot style modifiers
  }
  // ... the other four charts
}
```

8. A dashed line with no label is a puzzle rather than information, so create `AnnotationLabel.swift` in the `Views` folder. Building the label as its own view keeps the chart code readable and gives the two annotations we are about to write a shared appearance. The translucent background matters more than it looks: an annotation sits

on top of the plot, so without it the text competes with whatever it covers:

```
import SwiftUI
struct AnnotationLabel: View {
  let text: String
  init(_ text: String) {
    self.text = text
  }
  var body: some View {
    Text(text)
      .font(.caption2)
      .foregroundStyle(.secondary)
      .padding(.horizontal, 4)
      .background(.background.opacity(0.7), in: .rect(cornerRadi
  }
}
#Preview {
  AnnotationLabel("Average 74 mm")
}
```

9. Attach the label to the rule with `.annotation()`, which places any view we like at a mark's position. Aligning it to the leading edge keeps it clear of the bars in the middle of the year:

```
struct ClimateChartView: View {
  // ... the style property and the body
  private var rainfallChart: some View {
    Chart {
      ForEach(ClimateRecord.all) { record in
        // ... the BarMark with its foregroundStyle and positior
      }
      RuleMark(y: .value("Average", ClimateRecord.meanRainfall))
        .foregroundStyle(.secondary)
        .lineStyle(StrokeStyle(lineWidth: 1, dash: [4]))
        .annotation(position: .top, alignment: .leading, spacing
          AnnotationLabel("Average \(Int(ClimateRecord.meanRain
        }
    }
    // ... the axis, scale, legend, and plot style modifiers
  }
```

```
  // ... the other four charts
}
```

10. Label the wettest month the same way, this time on a bar rather than a rule. The annotation closure runs for all thirty-six records, so an `if` inside it is what makes exactly one label appear. This is worth remembering as the general pattern for calling out a single data point without filtering the data or adding a second mark:

```
struct ClimateChartView: View {
  // ... the style property and the body
  private var rainfallChart: some View {
    Chart {
      ForEach(ClimateRecord.all) { record in
        BarMark(
          x: .value("Month", record.date, unit: .month),
          y: .value("Rainfall", record.rainfall)
        )
        .foregroundStyle(by: .value("City", record.city))
        .position(by: .value("City", record.city))
        .annotation(position: .top, spacing: 2) {
          if record.id == ClimateRecord.wettest.id {
            AnnotationLabel("\(Int(record.rainfall)) mm")
          }
        }
      }
      // ... the RuleMark with its annotation
    }
    // ... the axis, scale, legend, and plot style modifiers
  }
  // ... the other four charts
}
```

11. With the preview canvas in live mode, look at what the chart can say now that it could not say when we opened the starter project: twelve month labels instead of four, millimeters instead of bare numbers, a range that holds still when the data changes, colors that are ours rather than assigned, and two marks naming the average and the

peak. Switching to any other segment shows the four charts we left alone, still drawing the defaults this one started from. For the complete app code, see the book's GitHub repository.



Figure 13.2: The rainfall chart before and after the customizations in this recipe

12. The other four charts are still on defaults, which makes them a ready-made exercise. Try giving **Temp** a fixed `.chartYScale(domain:)` so the comfort band always sits at the same height, and an annotation on the band naming it. **Scatter** needs different treatment from everything here, because its x-axis holds temperature rather than dates, so month formats do not apply and degree labels do. **Area** plots the same

millimeters as **Rain** and can take the same y-axis. **Range** can mark its widest month with an annotation. Then look at what those four have in common: the axis work has to be written per chart, because each one plots different types, but `.chartForegroundStyleScale()`, `.chartLegend()`, and `.chartPlotStyle()` care about none of that. Lift those three off `rainfallChart` and onto the `switch` in `body`, and all five cha rts inherit them at once.

How it works…

Every modifier in this recipe hangs off the `Chart` rather than off a mark, and that placement is the whole mental model. A mark modifier such as `.foregroundStyle(by:)` describes one drawn element; a chart modifier such as `.chartYScale(domain:)` describes the space all the marks are drawn into. It also explains why none of our work leaked into the other four charts even though they live in the same file: each computed property builds its own `Chart`, so each carries its own axes, scales, and legend.

Supplying an axis content closure is a replacement rather than an addition, which is the detail that trips people up first. The default axis is a package of gridlines, ticks, and labels, and the moment we write `.chartXAxis { }` we own all three. That is why both of our closures name `AxisGridLine` and `AxisTick` alongside `AxisValueLabel`, and why leaving one out is how an axis quietly loses its ticks. The values passed to `AxisMarks` are a request rather than a command, though: `.stride(by: .month)` gives us twelve because twelve narrow labels fit, and the same request on a narrower view would still be thinned.

The `AxisValue` handed to the content closure carries the plotted value in whatever type produced it, which is why reading it back means asking for a type with `value.as(Double.self)` and handling the optional. Our y-axis uses that value twice over: once to build the label text with its unit, and

once to decide whether the mark is a labeled major or a faint dashed minor. Branching inside `AxisMarks` is how a chart gets denser gridlines than it has labels, and it costs nothing at runtime because the closure runs a handful of times, not once per record.

Scales are the other half of accuracy. Without `.chartYScale(domain:)` the y-axis is derived from the data, so a bar's height means something only relative to the other bars currently on screen, and a chart that is redrawn with new records silently rescales underneath the same layout. `.chartForegroundStyleScale()` fixes the same class of problem for color: the default assignment follows the order values are encountered, so it is stable only as long as nobody edits the data. Both modifiers trade a little flexibility for a chart that keeps its meaning when the numbers change.

Annotations are ordinary SwiftUI views positioned relative to a mark, which is why `AnnotationLabel` is a plain `View` with no chart-specific API in it at all. The difference between our two uses is worth holding onto. The one on `RuleMark` runs once, because there is one rule. The one on `BarMark` runs for every record, since it is attached inside the `ForEach`, so the `if` comparing against `ClimateRecord.wettest` is what reduces thirty-six potential labels to the single one on Miami's June bar. Reaching for a filtered collection or a second mark type would have worked too, and it would have put presentation logic into the data instead of leaving it in the view where it belongs.

See also

- `AxisMarks` documentation: https://x.swiftuicookbook.com/bjb9j
- `AxisValueLabel` documentation: https://x.swiftuicookbook.com/sqt8e
- `RuleMark` documentation: https://x.swiftuicookbook.com/n6iza

- `AnnotationPosition` documentation: https://x.swiftuicookbook.com/4nv35
- *Customizing axes in Swift Charts* article: https://x.swiftuicookbook.com/2jt3m
- Swift Charts: Raise the bar (WWDC 2022): https://x.swiftuicookbook.com/567db

Interactive Swift Charts

A chart printed in a textbook or pasted into a presentation has to say everything at once, because looking at it is the only thing anyone can do. A chart in an app does not: it can hold far more data than it shows and move through it on demand, and it can report the exact value of any point the user asks about. That interactivity is the whole subject of this recipe.

In this recipe, we build an activity browser for four months of daily step counts. One month is visible on the screen at any time. Pressing and holding a day highlights its bar and displays an annotation with that day's step count and active minutes, placed so the whole annotation stays inside the chart no matter which day is picked. A second chart underneath holds all four months of data and shows the same thirty days as the upper chart. Dragging the lower chart sideways moves the thirty-day range across the whole period, and because both charts draw the same range of dates, the upper chart changes its data to match as the lower chart scrolls. No day is highlighted while this is happening, since the selection clears as soon as the finger lifts from the upper chart; the highlight returns when a day is selected again, and then it appears in both charts at once.

Drawing two charts is a decision about the data: step counts run into the thousands while active minutes stay in the tens, so the two cannot share a y-axis without flattening the smaller of them. Selecting a day and scrolling

the data are both drag gestures, and a chart acts on only one of them: with `.chartXSelection()` and `.chartScrollableAxes()` applied to the same chart, selection takes the drag and the chart becomes very difficult to scroll. The result is an unreliable feature: the chart scrolls once in a while and refuses the same gesture the next time, reading the drag as a long press and showing the annotation instead. Handing the selection to the upper chart and the scrolling to the lower one is w hat makes both of them work.

Getting ready

Create a new SwiftUI iOS app named **ActivityTrends** .

How to do it...

We start with a repeatable data set so every selection and scroll in this recipe can be checked against the same bars. We continue with the steps chart and its selection, the annotation card it shows, and finally the active minutes chart that drives the horizontal scrolling. These are the steps:

1. The app needs four months of daily step counts and active minutes, which a real app would read from the HealthKit store or from a server. Here the data set is generated randomly with `Double.random` . Using it on its own would produce a different chart on every launch, so neither the values printed in this recipe nor the bars drawn in its figures would match what we see in the actual app. The file therefore starts with a custom random number generator we control: `SeededGenerator` conforms to `RandomNumberGenerator` , so it works with the standard random APIs, and a fixed seed makes its sequence repeatable. It is the same generator that produces the data for the 400 stars in the *Canvas and Custom Rendering* section of *Chapter 12* . Create `DailyActivity.swift` in a `Models` folder with the following code:

```
import Foundation
struct SeededGenerator: RandomNumberGenerator {
  private var state: UInt64
  init(seed: UInt64) {
    state = seed
  }
  mutating func next() -> UInt64 {
    state = state &* 6364136223846793005 &+ 1442695040888963407
    return state
  }
}
```

2. Add the model and its data to the same file. Real activity is erratic rather than periodic, so each day is drawn at random inside a band around an average, and the average itself drops at weekends. That gives a chart with a weekly rhythm and no two weeks alike. The generator is created once and threaded through the whole sequence, which is what creates a deterministic result:

```
struct DailyActivity: Identifiable {
  let date: Date
  let steps: Int
  let activeMinutes: Int
  var id: Date { date }
}
extension DailyActivity {
  static let lastDay =
    DateComponents(calendar: .current, year: 2026, month: 6, day
    ?? .now
  static let windowLength: TimeInterval = 3600 * 24 * 30
  static var lastMonthStart: Date {
    lastDay.addingTimeInterval(-windowLength)
  }
  static let all: [DailyActivity] = {
    var generator = SeededGenerator(seed: 1972)
    return (0..<120).reversed().map { offset in
      let date =
        Calendar.current.date(byAdding: .day, value: -offset, to
        ?? lastDay
      let weekday = Calendar.current.component(.weekday, from: d
```

```
        let isWeekend = weekday == 1 || weekday == 7
        let average = isWeekend ? 6000.0 : 10000.0
        let range = 0.85...1.15
        let steps = average * Double.random(in: range, using: &ger
        let minutes = steps / 150 * Double.random(in: range, using
        return DailyActivity(date: date, steps: Int(steps),
                             activeMinutes: Int(minutes))
      }
    }()
}
```

Note the fixed `lastDay` rather than `Date.now`. Deriving the data from today's date would mean the chart quietly changes every time we run the app, which is fine in an app and useless in a book.

3. Create `StepsChartView.swift` in a `Views` folder. This is the steps chart, and it draws one month rather than all four. That is a design decision rather than a technical limit: 120 bars across a phone screen would be too thin to tell apart or to press on, and this recipe is built around picking out a single day. Two details make the shorter range work: a filter that keeps only the days the scale covers, and a half-open upper bound on that range:

```
import SwiftUI
import Charts
struct StepsChartView: View {
  let windowStart: Date
  private var windowEnd: Date {
    windowStart.addingTimeInterval(DailyActivity.windowLength)
  }
  private var visible: [DailyActivity] {
    DailyActivity.all.filter { $0.date >= windowStart && $0.date
  }
  var body: some View {
    Chart {
      ForEach(visible) { activity in
        BarMark(
          x: .value("Day", activity.date, unit: .day),
          y: .value("Steps", activity.steps)
```

```
        )
        .foregroundStyle(.teal)
      }
    }
    .chartXScale(domain: windowStart...windowEnd)
  }
}
#Preview {
  StepsChartView(windowStart: DailyActivity.lastMonthStart)
    .padding(.top, 100)
    .padding([.horizontal, .bottom])
}
```

The preview pads the top by 100 points rather than using `.padding()` alone. The card added in a later step is drawn above the chart, and without that room it would be cut off by the top of the screen.

4. The chart draws the data now, but it is not interactive yet, so the next addition is selection. `.chartXSelection(value:)` does most of the work: it installs the gesture recognizer, converts the touch position into a `Date`, and writes that date into a binding, leaving no coordinate math to do. The `selected` property turns the date back into an activity record, and dimming every other bar is what makes the choice visible on screen. Still in `StepsChartView.swift`, replace the `StepsChartView` struct and the `#Preview` below it with the following:

```
struct StepsChartView: View {
  @Binding var selectedDate: Date?
  let windowStart: Date
  // ... the windowEnd and visible properties
  private var selected: DailyActivity? {
    guard let selectedDate else { return nil }
    return DailyActivity.all.first {
      Calendar.current.isDate($0.date, inSameDayAs: selectedDat
    }
  }
  var body: some View {
    Chart {
      ForEach(visible) { activity in
```

```
        BarMark(
          x: .value("Day", activity.date, unit: .day),
          y: .value("Steps", activity.steps)
        )
        .foregroundStyle(.teal)
        .opacity(selected == nil || selected?.id == activity.id
      }
    }
    .chartXScale(domain: windowStart...windowEnd)
    .chartXSelection(value: $selectedDate)
  }
}
#Preview {
  @Previewable @State var date: Date?
  StepsChartView(selectedDate: $date, windowStart: DailyActivity
    .padding(.top, 100)
    .padding([.horizontal, .bottom])
}
```

With the preview canvas in live mode, press and hold a bar and drag the finger left and right. The bar under the finger keeps its full color while all the others dim, and lifting the finger clears the selection again.

5. With the chart finished, it can go on screen. The selected day and the first date of the range both live in `ContentView`, the parent of both chart views, because both charts need them to stay in sync. The steps chart receives the date as a plain value, since it only reads it and never changes it, and receives the selection as a binding, since pressing a bar does change that. Replace the `ContentView` struct with the following:

```
struct ContentView: View {
  @State private var selectedDate: Date?
  @State private var windowStart = DailyActivity.lastMonthStart
  var body: some View {
    NavigationStack {
      VStack(alignment: .leading, spacing: 16) {
        Text("Steps")
```

```
            .font(.headline)
          StepsChartView(selectedDate: $selectedDate, windowStart:
        }
        .padding()
        .navigationTitle("Activity Trends")
      }
    }
}
```

6. Leaving one bar at full color while the others dim shows which day is selected, but says nothing about what that day is worth. A small card can carry the numbers, and building it as its own view keeps the chart code readable. It stacks three lines: the weekday and date, the step count, and the active minutes. The step count is given the most weight because it is the value the chart is about. The background and shadow are not just decoration. The card will be drawn on top of the chart, so without an opaque background the bars would show through the text. Create `SelectionCard.swift` in the `Views` folder:

```
import SwiftUI
struct SelectionCard: View {
  let activity: DailyActivity
  var body: some View {
    VStack(alignment: .leading, spacing: 2) {
      Text(activity.date, format: .dateTime.weekday(.abbreviated
        .font(.caption2)
        .foregroundStyle(.secondary)
      Text("\(activity.steps.formatted()) steps")
        .font(.caption)
        .fontWeight(.semibold)
      Text("\(activity.activeMinutes.formatted()) active minutes
        .font(.caption2)
        .foregroundStyle(.secondary)
    }
    .padding(8)
    .background(.background, in: .rect(cornerRadius: 8))
    .shadow(radius: 2)
  }
}
```

```
#Preview {
  SelectionCard(activity: DailyActivity.all[60])
}
```

7. Put the card on the steps chart. A `RuleMark` at the selected date draws the vertical line above the chosen bar, and the card is attached to that mark as an annotation, which is what positions it over the right day. The `overflowResolution` argument is the part worth remembering: without it, a card centered over a bar near either edge would hang off the side of the chart. With `x: .fit(to: .chart)` the framework slides the card inward until it fits, while the rule stays exactly where it was. Two smaller details finish the mark: `y: .disabled` leaves the vertical placement alone, and `.zIndex(-1)` keeps the rule behind the bars rather than drawing a line across them:

```
struct StepsChartView: View {
  // ... the two stored properties and the three computed ones
  var body: some View {
    Chart {
      ForEach(visible) { activity in
        // ... the BarMark with its foregroundStyle and opacity
      }
      if let selected {
        RuleMark(x: .value("Selected", selected.date, unit: .day
          .foregroundStyle(.secondary)
          .lineStyle(StrokeStyle(lineWidth: 1, dash: [4]))
          .zIndex(-1)
          .annotation(
            position: .top,
            spacing: 4,
            overflowResolution: .init(x: .fit(to: .chart), y: .
          ) {
            SelectionCard(activity: selected)
          }
      }
    }
    // ... the chartXScale and chartXSelection modifiers
  }
}
```

Back in the preview canvas, press and hold near the middle of the chart and the card appears above the rule. Drag all the way to the left and right edges and watch the card slide inward to stay whole while the rule stays on the selected bar.

8. Let's make a small cosmetic improvement to the main view. The card is drawn above the steps chart, so a selection near the left edge slides it directly onto the **Steps** heading. Fading the heading while a selection exists solves that without moving anything, since an opacity change leaves the layout identical. In `ContentView.swift`, add two modifiers to the heading:

```
struct ContentView: View {
  // ... the two state properties
  var body: some View {
    NavigationStack {
      VStack(alignment: .leading, spacing: 16) {
        Text("Steps")
          .font(.headline)
          .opacity(selectedDate == nil ? 1 : 0)
          .animation(.default, value: selectedDate)
        // ... the StepsChartView
      }
      .padding()
      .navigationTitle("Activity Trends")
    }
  }
}
```

In the preview canvas, press and hold anywhere on the chart: the **Steps** heading fades out while the card is up and returns as soon as the finger lifts. Try a bar near the left edge to see why it earns its place, since that is where the card slides across and would otherwise sit on top of the heading.

9. Let's work on the active minutes chart. It takes the opposite approach to the steps chart: it holds all 120 days and shows a moving 30-day range of data, with `.chartScrollableAxes()` enabling the drag, `.chartXVisibleDomain()` setting how many days are visible, and `.chartScrollPosition()` reporting the first date currently shown. It reads the selection to highlight the matching day, but never modifies it. Create `MinutesChartView.swift` in the `Views` folder:

```
import SwiftUI
import Charts
struct MinutesChartView: View {
  let selectedDate: Date?
  @Binding var windowStart: Date
  var body: some View {
    Chart {
      ForEach(DailyActivity.all) { activity in
        BarMark(
          x: .value("Day", activity.date, unit: .day),
          y: .value("Active minutes", activity.activeMinutes)
        )
        .foregroundStyle(.orange)
        .opacity(isSelected(activity) ? 1 : 0.4)
      }
    }
    .chartXVisibleDomain(length: DailyActivity.windowLength)
    .chartScrollableAxes(.horizontal)
    .chartScrollPosition(x: $windowStart)
    .chartYAxis {
      AxisMarks(values: .automatic(desiredCount: 3))
    }
  }
  private func isSelected(_ activity: DailyActivity) -> Bool {
    guard let selectedDate else { return true }
    return Calendar.current.isDate(activity.date, inSameDayAs: s
  }
}
#Preview {
  @Previewable @State var start = DailyActivity.lastMonthStart
  MinutesChartView(selectedDate: nil, windowStart: $start)
    .frame(height: 120)
    .padding()
}
```



Swift Charts also offers `.chartScrollTargetBehavior()`, which snaps scrolling to whole days or months and looks like a natural addition here. Adding it silently breaks the starting position: it overrides the initial value of the binding passed to `.chartScrollPosition()` and opens the chart at the very beginning of the data instead. If a scrollable chart ignores where you told it to start, that modifier is the first thing to remove.

10. Add the second chart to `ContentView`. The two charts share `windowStart`, but only the scrolling one modifies it, which is why the steps chart receives a plain value and the active minutes chart receives a binding. Dragging the lower chart now moves the upper one:

```
struct ContentView: View {
  @State private var selectedDate: Date?
  @State private var windowStart = DailyActivity.lastMonthStart
  var body: some View {
    NavigationStack {
      VStack(alignment: .leading, spacing: 16) {
        // ... the Steps heading and the StepsChartView
        Text("Active minutes")
          .font(.headline)
        MinutesChartView(selectedDate: selectedDate, windowStar
          .frame(height: 120)
      }
      .padding()
      .navigationTitle("Activity Trends")
    }
  }
}
```

11. Give `ContentView` an initializer so it can start with a day already selected. This is worth doing for any view whose interesting state only appears mid-gesture: without it, the selected state can only be seen by holding a finger on the chart, which is impossible to inspect in the preview canvas. Note that `windowStart` gives up the initial value it had, because of how the `State()` macro behaves when a property that already carries a value is assigned in an initializer. We cover that macro in the *State and Binding Basics* section of *Chapter 5*. Add the initializer, drop that initial value, and replace the `#Preview` at the bottom with three:

```
struct ContentView: View {
  @State private var selectedDate: Date?
  @State private var windowStart: Date
  init(selectedDate: Date? = nil) {
    self.selectedDate = selectedDate
    self.windowStart = DailyActivity.lastMonthStart
  }
  // ... the body
}
#Preview {
  ContentView()
}
#Preview("Mid-window selection") {
  ContentView(selectedDate: DailyActivity.all[103].date)
}
#Preview("Selection at right edge") {
  ContentView(selectedDate: DailyActivity.all[118].date)
}
```

12. With the preview canvas in live mode, press and hold a bar in the upper chart and watch the annotation name the day while the matching bar lights up below. Keep holding and drag sideways to run it across the month, then let go to clear the selection. Then drag the lower chart and see both charts move together to a different month. Pick a day at the far left or right and notice the card sliding

inward to stay whole while its rule stays put. For the complete app code, see the book's GitHub repository.



Figure 13.3: The app at rest, a day selected in the middle of the range, and a day selected at the right edge where the card shifts inward to stay on screen

How it works...

Two quantities need two charts, since step counts in the thousands and active minutes in the tens cannot share a y-axis. What the gesture conflict decides is narrower than that: which of the two interactions each chart carries. `.chartXSelection(value:)` installs its own recognizers, a tap and a press-and-drag, and `.chartScrollableAxes()` wants that same drag for scrolling. Putting both on one chart does not fail loudly; it produces a chart that selects readily and scrolls almost never, which is worse than a clean failure, because it reads as a layout problem rather than a gesture one. I split the roles only after watching a chart that carried both refuse to

scroll: with the chart being read fixed and the chart being browsed scrolling, both interactions feel immediate. The same reasoning applies to `.chartGesture()`, which hands a `ChartProxy` to a gesture we write ourselves and would let us drive selection from a custom drag: it belongs on a chart that is not already scrolling, and here `.chartXSelection()` covers the same ground with one line.

Selection is a `Date`, not a bar. `.chartXSelection(value:)` converts the touch position into a value on the x scale, which is the same quantity the marks are plotted against, so it can report a day whose bar was never precisely hit. That is why `selected` searches with `isDate(_:inSameDayAs:)` rather than comparing dates for equality; an exact match would almost always fail. It also explains why the lower chart can highlight the same day without any selection modifier at all. It receives the date and compares, and nothing about highlighting requires owning the gesture.

The two charts frame their data in opposite ways. The steps chart fixes its x scale with `.chartXScale(domain:)` and filters its data to match, so it draws exactly what it shows. Marks drawn outside the scale still render, spilling past the plot and onto the axis labels, which is what the `visible` filter prevents. And the filter's upper bound has to be strict, or the last day in the range lands on the edge of the scale and half its bar is drawn outside the plot. The active minutes chart keeps every record and moves a 30-day range across them with `.chartXVisibleDomain()` and `.chartScrollableAxes()`, reporting where that range currently starts through `.chartScrollPosition()`. Because that position is a binding, scrolling the active minutes chart changes the value of `windowStart` in the parent view, which then flows back down as a plain value and changes which days the steps chart draws. `ContentView` owns that date, the active minutes chart writes to it through a binding, and the steps chart receives a copy of it, which is the ordinary SwiftUI arrangement of state, binding, and value applied to a chart's scale.

Annotations are placed relative to a mark, so an annotation on a mark near the edge would naturally hang off the side. `overflowResolution` is what turns that from a layout problem into a solved one: `x: .fit(to: .chart)` shifts the annotation horizontally until it sits inside the chart, and `y: .disabled` leaves the vertical placement untouched. What it does not do is move the mark, which is precisely the behavior we want, since the rule has to keep pointing at the selected day while the card drifts. It also does not know about anything outside the chart, which is why the `Steps` heading needed fading: the card can be resolved into the chart's bounds and still land on top of a view above it.

We used the x axis throughout because time is what this data is organized by, but `.chartYSelection(value:)` is the matching modifier for the other axis, reporting a value on the y scale instead. It is worth knowing about and worth using sparingly on the same chart as x selection, since a single tap has to resolve to one axis or the other, and a chart that tries to read both from one gesture will make a choice the user did not intend.

See also

- `chartXSelection(value:)` documentation: https://x.swiftuicookbook.com/wiqhr
- `chartScrollableAxes(_:)` documentation: https://x.swiftuicookbook.com/f6eq6
- `chartScrollPosition(x:)` documentation: https://x.swiftuicookbook.com/tkknv
- `AnnotationOverflowResolution` documentation: https://x.swiftuicookbook.com/h4lgt
- `RandomNumberGenerator` documentation: https://x.swiftuicookbook.com/2yb74

- Explore pie charts and interactivity in Swift Charts (WWDC 2023): https://x.swiftuicookbook.com/tx6hi

Vectorized and Function Plots

Every chart we have built in this chapter started from an array of records, with a `ForEach` turning each element into a mark. A mathematical curve does not arrive as an array. To draw the graph of a formula with marks, we would first have to convert the formula into data: pick a range, evaluate the function at some number of points, and connect the results with line segments. That point count is a rendering concern disguised as a data concern, because the right number depends on the size and resolution of the chart, which the code generating the array knows nothing about. The function plots introduced with iOS 18 put the sampling where it belongs: `LinePlot` and `AreaPlot` accept the function itself as a closure, and the framework evaluates it wherever it needs values to draw a smooth curve.

In this recipe, we build a plotting playground with three tabs. The first tab draws a sine wave whose amplitude and frequency are set by sliders; since the chart holds a function rather than data, moving a slider changes the closure and the curve follows, with nothing to regenerate. The second tab uses the parametric form of `LinePlot` to draw a spiral, a curve that no function of x can produce. The third tab covers the other half of what iOS 18 added to Swift Charts, vectorized plots: it generates 400 noisy sensor readings scattered around the same sine wave and hands the whole array to a single `PointPlot` , which reads values through key paths instead of building a mark per element, and then lays the exact curve over the noise with th e same `LinePlot` the recipe started with.

Getting ready

Create a new SwiftUI iOS app named **FunctionLab** .

How to do it...

The app is three independent views, one per tab, plus a data file for the scatter tab. We build the three tab views first and assemble the `TabView` last, because `ContentView` references all three views and can only compile once they exist; in this order, the project compiles after every step. These are the steps:

1. Let's start with the function tab. Create `FunctionPlotView.swift` in a `Views` folder. The chart contains a single `LinePlot` whose closure receives an x value and returns the matching y; there is no data array anywhere in the file. The two string arguments name the plotted quantities, the same role the first argument of `.value(_:_:)` plays for a mark. The x scale runs from 0 to 2π, which frames exactly one period of the sine:

```
import SwiftUI
import Charts
struct FunctionPlotView: View {
  var body: some View {
    NavigationStack {
      Chart {
        LinePlot(x: "x", y: "y") { x in
          2 * sin(x)
        }
        .foregroundStyle(.blue)
        .lineStyle(StrokeStyle(lineWidth: 2))
      }
      .chartXScale(domain: 0...2*Double.pi)
      .chartYScale(domain: -3.5...3.5)
      .padding()
      .navigationTitle("Function")
    }
  }
}
#Preview {
```

```
    FunctionPlotView()
}
```

The preview canvas shows one complete wave, rising to 2 and dipping to -2.

2. The point of holding a function rather than data is that the function can change at any moment. Add `amplitude` and `frequency` state properties driven by two sliders, and wrap the chart in a `VStack` to give the sliders a place below it. The closure now computes `amplitude * sin(frequency * x)`, and it reads the two properties through a capture list: `LinePlot` declares its closure `Sendable`, and the compiler flags any direct reference from it to main-actor state such as `@State` properties, so `[amplitude, frequency]` stores plain copies inside the closure instead. Replace the `FunctionPlotView` struct with the following:

```
struct FunctionPlotView: View {
  @State private var amplitude = 2.0
  @State private var frequency = 1.0
  var body: some View {
    NavigationStack {
      VStack(alignment: .leading, spacing: 16) {
        Chart {
          LinePlot(x: "x", y: "y") { [amplitude, frequency] x in
            amplitude * sin(frequency * x)
          }
          .foregroundStyle(.blue)
          .lineStyle(StrokeStyle(lineWidth: 2))
        }
        .chartXScale(domain: 0...2*Double.pi)
        .chartYScale(domain: -3.5...3.5)
        HStack {
          Text("Amplitude")
          Slider(value: $amplitude, in: 0.5...3)
        }
        HStack {
          Text("Frequency")
          Slider(value: $frequency, in: 0.5...4)
```

```
        }
      }
      .padding()
      .navigationTitle("Function")
    }
  }
}
```

With the preview canvas in live mode, drag both sliders: **Amplitude** stretches the wave vertically, and **Frequency** packs more periods into the same 0 to 2π window.

3. `AreaPlot` is the second function plot. It takes the same closure but fills the region between the curve and zero instead of stroking the curve. Add a toggle and a conditional `AreaPlot` before the line, so it draws behind it, dimmed to 20 percent opacity to keep the line the stronger element; a chart builder accepts `if` statements just as a view builder does:

```
struct FunctionPlotView: View {
  @State private var amplitude = 2.0
  @State private var frequency = 1.0
  @State private var showsArea = false
  var body: some View {
    NavigationStack {
      VStack(alignment: .leading, spacing: 16) {
        Chart {
          if showsArea {
            AreaPlot(x: "x", y: "y") { [amplitude, frequency] x in
              amplitude * sin(frequency * x)
            }
            .foregroundStyle(.blue)
            .opacity(0.2)
          }
          // ... the LinePlot with its two modifiers
        }
        .chartXScale(domain: 0...2*Double.pi)
        .chartYScale(domain: -3.5...3.5)
        Toggle("Area under the curve", isOn: $showsArea)
        // ... the two slider HStacks
```

```
      }
      .padding()
      .navigationTitle("Function")
    }
  }
}
```

Turn **Area under the curve** on in the preview canvas, then move the sliders to see the fill track the curve.

4. The axes show numbers, but nothing names the function they belong to. Add a headline that prints the formula currently drawn, with both values rounded to one decimal; we assemble the string in a `formula` property rather than inline in the `body`:

```
struct FunctionPlotView: View {
  // ... the three state properties
  private var formula: String {
    let a = amplitude.formatted(.number.precision(.fractionLengt
    let f = frequency.formatted(.number.precision(.fractionLengt
    return "y = \(a) · sin(\(f) · x)"
  }
  var body: some View {
    NavigationStack {
      VStack(alignment: .leading, spacing: 16) {
        Text(formula)
          .font(.headline)
        // ... the Chart with its scale modifiers, the Toggle, a
      }
      .padding()
      .navigationTitle("Function")
    }
  }
}
```

Drag a slider and the headline updates together with the curve.

5. The second tab plots a curve that no function of x can draw: a spiral passes through the same x many times, once per turn. The parametric

`LinePlot` computes both coordinates from a third variable instead. Its closure receives `t` and returns the pair (x, y), and the required `domain` argument states the range of `t` to draw, here driven by a slider that sets the number of turns, with 2π of `t` per turn. The slider's label prints the current number of turns, and `.monospacedDigit()` gives every digit the same width, so the slider holds still while the number changes. Two layout details: `.aspectRatio(1, contentMode: .fit)` keeps the plot square, so the equal x and y scales render each turn round rather than elliptical, and the `Spacer` takes the leftover vertical space, since a square chart no longer expands to fill it. Create `ParametricPlotView.swift` in the `Views` folder:

```
import SwiftUI
import Charts
struct ParametricPlotView: View {
  @State private var turns = 3.0
  var body: some View {
    NavigationStack {
      VStack(alignment: .leading, spacing: 16) {
        Text("x = t · cos(t), y = t · sin(t)")
          .font(.headline)
        Chart {
          LinePlot(x: "x", y: "y", t: "t", domain: 0...(turns *
            let x = t * cos(t)
            let y = t * sin(t)
            return (x, y)
          }
          .foregroundStyle(.purple)
          .lineStyle(StrokeStyle(lineWidth: 2))
        }
        .chartXScale(domain: -26...26)
        .chartYScale(domain: -26...26)
        .aspectRatio(1, contentMode: .fit)
        HStack {
          Text("Turns: \(turns, format: .number.precision(.fract
            .monospacedDigit()
          Slider(value: $turns, in: 1...4)
        }
        Spacer()
      }
```

```
      .padding()
      .navigationTitle("Parametric")
    }
  }
}
#Preview {
  ParametricPlotView()
}
```

In the preview canvas, drag the **Turns** slider to the left: the count falls toward 1.0, the spiral unwinds toward a single loop, and each new slider value gives the plot a new domain to sample and draw.

6. The scatter tab plots measured data coming from two different sensors, data the project does not have, so we generate it. Like the activity data in the *Interactive Swift Charts* section, the values are random draws from the seeded `SeededGenerator`, so every launch produces the same 400 points and the figure in this recipe matches what we see on screen. `PlotSample` stores only the three values the chart will plot: the x position, the y reading, and the name of the sensor that produced it. Keeping them stored properties rather than computed ones follows Apple's guidance for vectorized plots, and *How it works…* explains what the framework does with them. Create `PlotSample.swift` in a `Models` folder:

```
import Foundation
struct SeededGenerator: RandomNumberGenerator {
  private var state: UInt64
  init(seed: UInt64) {
    state = seed
  }
  mutating func next() -> UInt64 {
    state = state &* 6364136223846793005 &+ 1442695040888963407
    return state
  }
}
struct PlotSample {
```

```
  let x: Double
  let y: Double
  let sensor: String
}
```

7. Add the sample data to the same file. Each reading is the function tab's default wave, `2 * sin(x)`, plus noise from a band whose width depends on the sensor, so one cloud hugs the curve and the other spreads around it; two sensors with visibly different noise give the vectorized styling a category worth coloring by. The outer loop appends all of one sensor's readings before the other's, which is Apple's other piece of guidance for vectorized plots: group the collection by the property used for styling:

```
extension PlotSample {
  static let all: [PlotSample] = {
    var generator = SeededGenerator(seed: 1972)
    let sensors: [(name: String, noise: Double)] = [
      ("Sensor A", 0.2),
      ("Sensor B", 0.6)
    ]
    var samples: [PlotSample] = []
    for sensor in sensors {
      for _ in 0..<200 {
        let x = Double.random(in: 0...10, using: &generator)
        let range = -sensor.noise...sensor.noise
        let noise = Double.random(in: range, using: &generator)
        samples.append(PlotSample(x: x, y: 2 * sin(x) + noise,
                                  sensor: sensor.name))
      }
    }
    return samples
  }()
}
```

8. Create `ScatterPlotView.swift` in the `Views` folder. `PointPlot` receives the entire array as its first argument, and where a mark pairs its label

with one element's value, the plot pairs it with a key path, `.value("x", \.x)`, that tells the framework where to find the value in every element. The styling follows the same pattern: `.foregroundStyle(by: .value("Sensor", \.sensor))` colors the points by sensor and builds the legend from the two names it finds, and `.symbolSize(16)` sets each dot's area to 16 square points, about four and a half points across, keeping the 400 readings visible as individual dots. Note the order of those two modifiers: the key path comes first, and swapping them stops the code from compiling, which *How it works…* explains:

```
import SwiftUI
import Charts
struct ScatterPlotView: View {
  var body: some View {
    NavigationStack {
      VStack(alignment: .leading, spacing: 16) {
        Text("y = 2 · sin(x) + noise")
          .font(.headline)
        Chart {
          PointPlot(
            PlotSample.all,
            x: .value("x", \.x),
            y: .value("y", \.y)
          )
          .foregroundStyle(by: .value("Sensor", \.sensor))
          .symbolSize(16)
        }
        .chartXScale(domain: 0...10)
        .chartYScale(domain: -3...3)
      }
      .padding()
      .navigationTitle("Scatter")
    }
  }
}
#Preview {
  ScatterPlotView()
}
```

The preview canvas shows the tight cloud and the loose one in two colors, with the legend naming **Sensor A** and **Sensor B** .

9. The clouds visibly follow a wave, and we know its exact formula, so let's draw it. A function plot and a vectorized plot are both chart content and can share a chart: add a `LinePlot` with the generator's own `2 * sin(x)` after the `PointPlot` , in `.orange` so its color stays apart from the two sensor colors:

```
struct ScatterPlotView: View {
  var body: some View {
    NavigationStack {
      VStack(alignment: .leading, spacing: 16) {
        Text("y = 2 · sin(x) + noise")
          .font(.headline)
        Chart {
          // ... the PointPlot with its two modifiers
          LinePlot(x: "x", y: "y") { x in
            2 * sin(x)
          }
          .foregroundStyle(.orange)
          .lineStyle(StrokeStyle(lineWidth: 2))
        }
        .chartXScale(domain: 0...10)
        .chartYScale(domain: -3...3)
      }
      .padding()
      .navigationTitle("Scatter")
    }
  }
}
```

The curve runs through the center of both clouds, which is exactly what it should do: every reading is that curve plus noise centered on zero.

10. With the three views finished, `ContentView` can name them, which is why we saved this step for last. Each `Tab` pairs a title and an SF Symbol with one of our views. We cover `TabView` and the `Tab` API in

depth in the *TabViews and Liquid Glass* section of *Chapter 11*. In `ContentView.swift`, replace the `ContentView` struct with the following:

```
struct ContentView: View {
  var body: some View {
    TabView {
      Tab("Function", systemImage: "function") {
        FunctionPlotView()
      }
      Tab("Parametric", systemImage: "scribble.variable") {
        ParametricPlotView()
      }
      Tab("Scatter", systemImage: "chart.dots.scatter") {
        ScatterPlotView()
      }
    }
  }
}
```

11. Use the preview canvas in live mode and work through the tabs: on **Function**, drag both sliders and flip the area toggle, checking that the headline always names the curve on screen; on **Parametric**, unwind the spiral to one turn and wind it back to four; on **Scatter**, compare the two clouds against the overlaid curve. For the complete app code, see the book's GitHub repository.

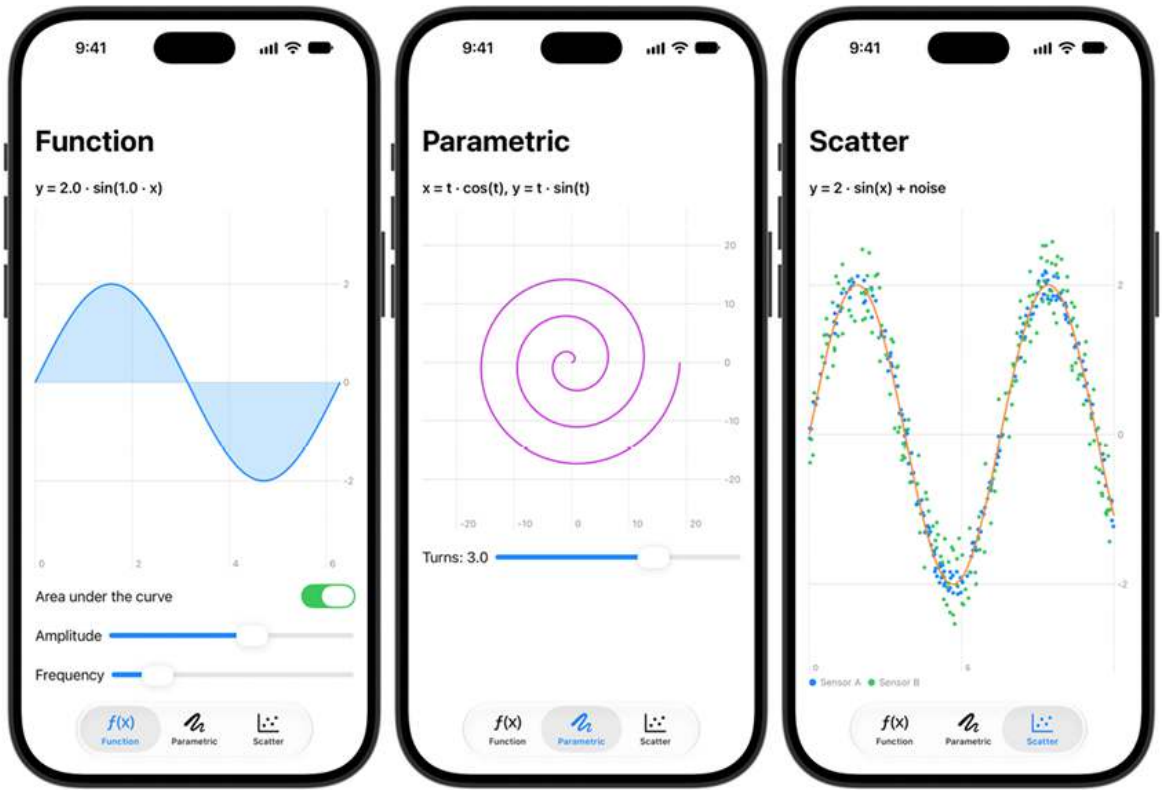


Figure 13.4: The sine wave with its area shaded, the parametric spiral, and the vectorized scatter plot with its function overlay

How it works...

A `ForEach` of marks starts from the data and asks how each element should be drawn; a function plot starts from the drawing and asks for values. When the chart needs the curve between 0 and 2π, it evaluates the closure at x positions of its own choosing, as many as a smooth result requires, and connects the results. The closure is declared `Sendable`, so the compiler flags any direct read of the view's main-actor state; the capture list copies `amplitude` and `frequency` into it at the moment `body` runs. Each slider change re-evaluates `body`, producing a new closure with fresh copies, and that is the entire update mechanism behind the live curve: no chart data changed, because there is none. When a function is undefined somewhere in the visible range, the closure returns `.nan` for those x values

and the plot leaves a gap instead of connecting across it. `AreaPlot` adds one form of its own: a closure returning a `(yStart, yEnd)` pair fills the region between two curves rather than between one curve and zero.

The parametric plot exists for curves where x cannot serve as the input, and our spiral is exactly that: both coordinates grow out of the angle `t`, and the `domain` argument tells the plot which range of `t` to draw. The same argument exists on the plain function plots, where it is optional and limits where the function is sampled independently of the x scale, useful when a function should only be drawn over part of the visible chart. In the parametric tab, binding the slider to the domain's upper bound is the whole interaction: every change hands the plot a new range of `t` to sample, and the spiral gains or loses turns accordingly.

`PointPlot` draws the same dots that a `ForEach` with one `PointMark` per element would; what changes is what we hand over. Given the whole collection and key paths, the framework reads x, y, and the sensor name directly out of each element without executing any of our code per point, which is what makes vectorized plots the right tool when points run to the hundreds or thousands and all of them share one mark type and styling. The two habits in our model file keep that path fast: plotted values live in stored properties, so a key-path read is a direct memory access rather than a computed-property call, and the array is grouped by sensor, so the renderer changes style once rather than potentially on every point. Vectorized plots style every element with the same modifiers; when elements need individual treatment, a different mark type per element, or a `zIndex` of their own, a `ForEach` of marks remains the right choice. The modifier order rule from step 8 is the type system enforcing that split: the key-path styling modifiers are defined on `VectorizedChartContent`, the protocol the plots conform to, while constant modifiers such as `.symbolSize(16)` exist on all chart content and return that general type.

Once a constant modifier runs, the vectorized type is gone and a key-path modifier can no longer attach, so key paths come first and constants last.

Plots and marks are all chart content, and they compose: our scatter chart holds a vectorized plot and a function plot side by side, and either could share the chart with plain marks as well. `PointPlot` is one of a full set: `BarPlot`, `LinePlot`, `AreaPlot`, `RulePlot`, `RectanglePlot`, and `SectorPlot` are the vectorized twins of the mark types from the *Basic Charts* section, all taking the same collection-plus-key-paths arguments. `LinePlot` and `AreaPlot` are the same types doing double duty, functi on plot in one initializer and vectorized plot in another.

See also

- `LinePlot` documentation: https://x.swiftuicookbook.com/jlhi6
- `AreaPlot` documentation: https://x.swiftuicookbook.com/161nb
- `PointPlot` documentation: https://x.swiftuicookbook.com/lu6yg
- `VectorizedChartContent` documentation: https://x.swiftuicookbook.com/3l3la
- Swift Charts: Vectorized and function plots (WWDC 2024): https://x.swiftuicookbook.com/2z9d8
- *Creating a data visualization dashboard with Swift Charts* article: https://x.swiftuicookbook.com/gb17b

3D Charts

All the charts we have built in this chapter live on a plane: two axes, with any further dimension squeezed into color, size, or a second mark. Some data is not suitable for plotting in two dimensions. It needs three. A stretch of terrain is the clearest example, because a position on the ground already takes two coordinates, and the measured value, the elevation, is a

third continuous dimension whose shape only reads as a shape when we can look at it from different sides. `Chart3D` , introduced with iOS 26, gives charts that third axis: marks accept a `z` value, the plot area becomes a cube, and the user explores it by rotating the scene.

In this recipe, we build an app that renders a mountain landscape as a colored, rotatable surface with six survey stations resting on it. The terrain side is a function, not data: `SurfacePlot` extends the function plots of the *Vectorized and Function Plots* section to two input variables, sampling a closure into a smooth two-dimensional surface that we color by elevation with the height-based surface style. The stations are data, plotted with `PointMark` and its new `z` argument. Around the chart, we wire the viewing controls: a `Chart3DPose` binding for drag rotation with a reset button, a toggle that switches the camera between orthographic and perspective projection, and the z-a xis scale and label modifiers that complete the axis system.

Getti ng ready

Create a new SwiftUI iOS app named **TerrainExplorer** .

How to do it...

The app is one view holding the chart and its controls, plus two small model files: the terrain function and the survey stations. We build the two models first because the view references both, which keeps the project compiling as we go. These are the steps:

1. Let's define the landscape. Create `Terrain.swift` in a `Models` folder. The whole terrain is a single function that takes a position on the ground and returns the elevation there. The product of a sine and a cosine gives smooth hills and valleys in both directions. The chart evaluates

this function from a `Sendable` closure, and under the project's default actor isolation an unmarked method belongs to the main actor, and the compiler flags a call to it from that closure, so we mark the function `nonisolated`:

```
import Foundation
enum Terrain {
  nonisolated static func elevation(x: Double, z: Double) -> Dou
    2 * sin(x / 2) * cos(z / 2)
  }
}
```

2. The stations are the app's data layer. Create `SurveyStation.swift` in the `Models` folder. Each station stores its position as an easting and a northing, the plain-number coordinates of a survey grid measuring distance east and north, and computes its elevation from the terrain function, so every station sits exactly on the surface. Note the argument order in that call, with the northing passed as `x` and the easting as `z`. That is deliberate, and we explain why in *How it works…*:

```
import Foundation
struct SurveyStation {
  let name: String
  let kind: String
  let easting: Double
  let northing: Double
  var elevation: Double {
    Terrain.elevation(x: northing, z: easting)
  }
}
extension SurveyStation {
  static let all: [SurveyStation] = [
    SurveyStation(name: "Summit One", kind: "Weather",
                  easting: 0.8, northing: 3.1),
    // ... the remaining 5 stations
```

```
  ]
}
```

The array holds six stations, three whose `kind` is `"Weather"` and three whose `kind` is `"Ranger"`, spread over the peaks, slopes, and valleys of our terrain. The full list is in the book's GitHub repository.

3. Now the chart. Create `TerrainView.swift` in a `Views` folder. `Chart3D` is the three-dimensional counterpart of `Chart`, and `SurfacePlot` is its function plot: the closure receives the two floor coordinates and returns the height of the surface at that point. The three string arguments name the plotted dimensions, and `.foregroundStyle(.heightBased)` colors the surface by height, cool colors low and warm colors high:

```
import SwiftUI
import Charts
struct TerrainView: View {
  var body: some View {
    NavigationStack {
      Chart3D {
        SurfacePlot(x: "Northing", y: "Elevation", z: "Easting") {
          Terrain.elevation(x: x, z: z)
        }
        .foregroundStyle(.heightBased)
      }
      .padding()
      .navigationTitle("Terrain")
    }
  }
}
#Preview {
  TerrainView()
}
```

The preview canvas shows a colorful but nearly flat ramp. Without explicit domains, the chart samples the function over a small default

window around zero, and our hills need more room than that.

4. Give the function its real range. All three scale modifiers exist in 3D, with `.chartZScale(domain:)` as the new member of the family. The floor becomes an 8 by 8 grid, and the y domain of -3 to 3 leaves margin around the hills, which rise to 2 and dip to -2. Add the three scales directly below the chart's closing brace:

```
struct TerrainView: View {
  var body: some View {
    NavigationStack {
      Chart3D {
        // ... the SurfacePlot and its .foregroundStyle
      }
      .chartXScale(domain: 0...8)
      .chartYScale(domain: -3...3)
      .chartZScale(domain: 0...8)
      .padding()
      .navigationTitle("Terrain")
    }
  }
}
```

   The hills appear, but the cube now overflows both edges of the screen, and the elevation axis is pushed out of view entirely.

5. Name the axes and size the chart. The axis-label modifiers mirror the scale family, z included. For the size, `.containerRelativeFrame(.vertical)`, which we first used to size carousel cards in the *Scroll Views* section of *Chapter 2*, computes the chart's height as a fraction of its container instead of a fixed point value, so the proportions hold on any device, and half the height leaves enough side room for the elevation axis. Insert the following between `.chartZScale` and `.padding()`:

```
struct TerrainView: View {
  var body: some View {
```

```
    NavigationStack {
      Chart3D {
        // ... the SurfacePlot and its .foregroundStyle
      }
      // ... the three scale modifiers
      .chartXAxisLabel("Northing")
      .chartYAxisLabel("Elevation")
      .chartZAxisLabel("Easting")
      .containerRelativeFrame(.vertical) { length, _ in
        length * 0.5
      }
      .padding()
      .navigationTitle("Terrain")
    }
  }
}
```

The cube now fits, with the floor axes named and numbered. Look closely at the elevation axis, though: the **Elevation** title prints in the middle of the axis, right on top of the tick label for 0.

6. The axis-content builders we use to customize flat charts work on the three-dimensional axes too, and they are the fix. Rebuild the elevation axis with `AxisMarks`, keeping every value's tick and grid line but skipping `AxisValueLabel` at zero, so the axis title has that row to itself. Add this between `.chartZScale` and `.chartXAxisLabel`:

```
struct TerrainView: View {
  var body: some View {
    NavigationStack {
      Chart3D {
        // ... the SurfacePlot and its .foregroundStyle
      }
      // ... the three scale modifiers
      .chartYAxis {
        AxisMarks { value in
          AxisTick()
          AxisGridLine()
          if let elevation = value.as(Double.self), elevation !=
            AxisValueLabel()
          }
```

```
        }
      }
      // ... the label modifiers and the containerRelativeFrame
      .padding()
      .navigationTitle("Terrain")
    }
  }
}
```

Elevation now sits alone in the zero row, while 2 and -2 keep their numbers.

7. Plot the stations on the surface. Inside a `Chart3D`, `PointMark` gains a `z` argument and renders as a small sphere. Each station plots at its grid position with the elevation it computed from the terrain function, so the spheres rest on the surface, and `.foregroundStyle(by:)` colors them by station kind. Add the `ForEach` after `.foregroundStyle(.heightBased)`:

```
struct TerrainView: View {
  var body: some View {
    NavigationStack {
      Chart3D {
        // ... the SurfacePlot and its .foregroundStyle
        ForEach(SurveyStation.all, id: \.name) { station in
          PointMark(
            x: .value("Northing", station.northing),
            y: .value("Elevation", station.elevation),
            z: .value("Easting", station.easting)
          )
          .foregroundStyle(by: .value("Type", station.kind))
        }
      }
      // ... the scale, axis, label, and frame modifiers
      .padding()
      .navigationTitle("Terrain")
    }
  }
}
```

Weather stations render blue and ranger stations green, the default palette's first two colors in the order the kinds appear in the data. No legend appears to name them. We come back to that in *How it works…* .

8. Comparing the bare surface with the measured points is worth a switch. Wrap the chart in a `VStack` so controls can sit below it, move `.padding()` and `.navigationTitle` from the chart to the stack, and gate the `ForEach` behind a new state property, since a `Chart3D` builder accepts `if` statements just as a view builder does. The file now maps like this:

```
struct TerrainView: View {
  @State private var showsStations = true
  var body: some View {
    NavigationStack {
      VStack(spacing: 16) {
        Chart3D {
          // ... the SurfacePlot and its .foregroundStyle
          if showsStations {
            // ... the ForEach with the station marks
          }
        }
        // ... the scale, axis, label, and frame modifiers
        Toggle("Show stations", isOn: $showsStations)
      }
      .padding()
      .navigationTitle("Terrain")
    }
  }
}
```

9. Rotation is what makes a 3D chart worth its pixels, and it takes one modifier. `Chart3DPose` describes the camera position with an azimuth, the rotation around the vertical axis, and an inclination, the tilt down from the horizon. Both are `Angle2D` values from the Spatial framework, so the file needs `import Spatial` . Passing a binding to

`.chart3DPose` lets the user rotate the chart by dragging, with every change written back into our state, which means a plain button can restore the starting angle. Add the import, the two pose properties, the modifier, and the reset button:

```
import SwiftUI
import Charts
import Spatial
struct TerrainView: View {
  private static let initialPose = Chart3DPose(
    azimuth: .degrees(30),
    inclination: .degrees(15)
  )
  @State private var pose = Self.initialPose
  @State private var showsStations = true
  var body: some View {
    NavigationStack {
      VStack(spacing: 16) {
        Chart3D {
          // ... the surface and the station marks
        }
        // ... the scale and axis modifiers
        .chart3DPose($pose)
        // ... the containerRelativeFrame
        Toggle("Show stations", isOn: $showsStations)
        Button("Reset view") {
          pose = Self.initialPose
        }
        .buttonStyle(.bordered)
      }
      .padding()
      .navigationTitle("Terrain")
    }
  }
}
```

10. The last control switches the camera's projection. Orthographic projection, the default, keeps equal sizes equal at any depth, which is what we want for reading values off the chart. Perspective projection shrinks distant geometry, which reads more like standing in the

landscape. Add a state property, the modifier below `.chart3DPose`, and a second toggle:

```
struct TerrainView: View {
  // ... the initialPose constant and the pose property
  @State private var usesPerspective = false
  @State private var showsStations = true
  var body: some View {
    NavigationStack {
      VStack(spacing: 16) {
        Chart3D {
          // ... the surface and the station marks
        }
        // ... the scale and axis modifiers
        .chart3DPose($pose)
        .chart3DCameraProjection(usesPerspective ? .perspective
        // ... the containerRelativeFrame
        Toggle("Show stations", isOn: $showsStations)
        Toggle("Perspective projection", isOn: $usesPerspective)
        // ... the Reset view button
      }
      .padding()
      .navigationTitle("Terrain")
    }
  }
}
```

11. The chart view is finished, so `ContentView` can show it. In `ContentView.swift`, replace the `ContentView` struct with the following:

```
struct ContentView: View {
  var body: some View {
    TerrainView()
  }
}
```

12. Use the preview canvas in live mode and drag the cube around slowly: the stations hidden behind the western ridge come into view as the far side rotates toward us, and the axis titles reposition

themselves along their edges as the cube turns. Toggle **Show stations** off to study the bare surface, switch **Perspective projection** on to watch the box edges converge toward the horizon, and press **Reset view** to snap back to the starting angle. For the complete `TerrainView`, see the book's GitHub repository.

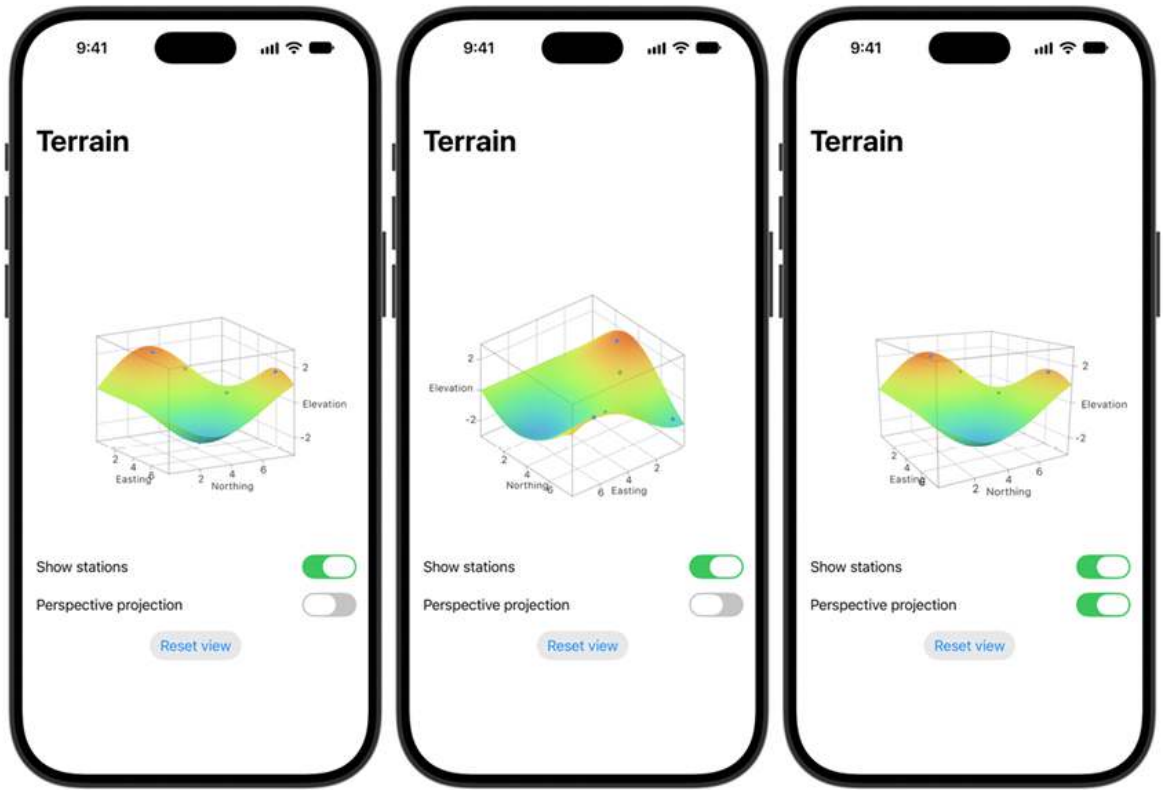


Figure 13.5: The terrain at the initial pose, rotated to bring the far stations into view, and with perspective projection on

How it works…

`Chart3D` keeps the architecture of its flat counterpart: a content builder full of marks and plots, scale modifiers that pin the domains, and axis builders that draw the frame. What changes is the geometry. Marks accept a `z` value, `PointMark` renders as a sphere, and `RuleMark` and `RectangleMark` extend the same way for reference lines and slabs in space. `SurfacePlot` carries the function-plot idea into two input variables: the framework

samples the closure over the floor window and stitches the results into a continuous surface, so sampling density stays a rendering concern, exactly as it was for `LinePlot`. The scale domains carry more weight here than they do in flat charts, because they set both the window the function is sampled over and the proportions of the cube itself: spans map to edge lengths, a domain that dwarfs the others stretches the cube toward a needle, and there is no separate aspect control. That is why our elevations share the single-digit range of the grid coordinates instead of using realistic meters. The height-based surface style has relatives worth knowing: `heightBased(yRange:)` pins its gradient to a fixed range, a gradient argument substitutes custom colors, and `.normalBased` shades by surface orientation instead of height.

The argument order in `Terrain.elevation(x: northing, z: easting)` looks backward until we spell out the scene's coordinate frame: x points right, y points up, and z points toward the viewer. Geographic coordinates form a right-handed triple of east, north, and up. Assigning east to x, up to y, and north to z swaps two members of that triple, and a single swap turns the frame into its mirror image. The rendered map then shows east where west belongs, and no rotation of the cube can undo it. Shifting the triple instead, north to x, up to y, east to z, is a rotation rather than a mirror, so the compass stays true: the camera starts on the eastern side looking west, with north receding to the right. When a 3D chart of map data looks subtly wrong, check the handedness of the axis assignment before touching the pose.

The z axis completes the axis system with modifiers named exactly like their x and y siblings, and the 2D axis DSL carries over with it, which is what step 6 leans on: `AxisMarks` hands us each axis value, and any component we do not emit is simply not drawn. That control matters more in 3D because the elevation axis annotations have hard limits. The axis title only renders in the value-label strip on the narrow side of the

cube, and nowhere else. `.chartYAxisLabel` honors only its string forms, ignoring the view-builder overload and the `position` argument. On top of that, a title of any length collides with the tick label at zero, which is why our axis drops that one number rather than abbreviating the title.

3D charts also drop interactive features we may take for granted, and the list is easier to learn here than the hard way. Selection is limited: `.chartXSelection(value:)` and its 3D companion `.chartZSelection(value:)` report the values under a tap, but no chart proxy exists to convert a touch location into data coordinates, and marks have no `.annotation()` modifier, so the tap-for-details pattern we built in the *Interactive Swift Charts* section cannot be ported as it is, and the drag gesture already belongs to rotation anyway. There is no legend either: `.foregroundStyle(by:)` assigns the default palette in the order the categories appear in the data, but `.chartLegend` and `.chartForegroundStyleScale` are silently ignored, so nothing on screen names the categories, and naming falls to the surrounding UI or, as in this recipe, to the text. Interaction in a 3D chart means changing the viewpoint, while identification still happens by color and position.

The pose binding is plain SwiftUI state: dragging writes new angles into `pose`, and setting `pose` moves the camera, which is all the **Reset view** button does. The change lands instantly: pose updates ignore `withAnimation`. Besides the initializer, `Chart3DPose` ships presets such as `.front`, `.top`, and `.default` for standard viewpoints. The projection choice comes down to reading versus feeling: orthographic keeps a far station the same size as a near one, so comparisons stay honest, while perspective converges the box edges like a photograph and suits a walkthrough imp ression. On iOS, the modifier's `.automatic` default resolves to orthographic.

See also

- `Chart3D` documentation: https://x.swiftuicookbook.com/oyewf

- `SurfacePlot` documentation: https://x.swiftuicookbook.com/hhesd
- `Chart3DPose` documentation: https://x.swiftuicookbook.com/2kj59
- Bring Swift Charts to the third dimension (WWDC 2025): https://x.swiftuicookbook.com/4l77w

14

Web Content and Rich Text

Somewhere in every app I have shipped, there is a screen that is not entirely the app's own: a help page pulled from the web, a note the user formats by hand, a document that arrives as Markdown or leaves as a PDF. For most of SwiftUI's life those screens meant a detour through UIKit, a `WKWebView` behind a `UIViewRepresentable`, a `UITextView` with an `NSAttributedString` and a delegate, and enough glue code to make either one look like it belonged. Starting with iOS 26, the first two have native SwiftUI counterparts, and the third was always closer than it looked. This chapter puts the three of them to work.

We begin by turning `WebView` and its observable `WebPage` into a small browser rather than embedding a single page, because a browser forces us to solve the problems production apps eventually meet. It loads URLs and app-generated HTML with `load(_:)` and `load(html:)`, reflects the page's title, progress, and `backForwardList` in its toolbar, and follows each navigation through the `navigations` sequence of `NavigationEvent` values. From there, we decide policy with `WebPage.NavigationDeciding`, serve local help pages through a `URLSchemeHandler`, build a table of contents with `callJavaScript(_:arguments:)`, and finish with `findNavigator(isPresented:)` and the back-forward swipe gestures.

The second recipe turns `TextEditor` into a rich text editor by binding it to an `AttributedString` and an `AttributedTextSelection`. Its formatting bar reads `typingAttributes(in:)` and `indices(in:)` to light up the right buttons, resolves the visible font traits through the `fontResolutionContext`, and applies edits with `transformAttributes(in:)` and `transform(updating:)`. An `AttributedTextFormattingDefinition` with an `AttributedTextValueConstraint` then defines exactly which attributes and values the editor accepts. In the final recipe, we compare `Text` with a `LocalizedStringKey` and `Text(verbatim:)` before rendering an article block by block from `AttributedString(markdown:)` and its `PresentationIntent` runs. We set that article in a bundled typeface registered under `UIAppFonts` and loaded with `Font.custom(_:size:relativeTo:)`, export it with `ImageRenderer`, display it through a `UIViewRepresentable` wrapper around PDFKit's `PDFView`, and share it with `ShareLink`.

By the end of this chapter, you should be able to embed web content that behaves like part of your app rather than a window onto someone else's, give the user a rich text editor with exactly the formatting you decide to support, and move formatted text between Markdown, your own typography, and PDF without leaving SwiftUI.

We are going to cover the following recipes in this chapter:

- WebView in SwiftUI
- Rich Text with AttributedString
- PDF, Markdown and Custom Fonts

Technical requirements

The code in this chapter is based on Xcode 27.0 and iOS 27.0. You can download and install the latest version of Xcode from the App Store. All the code examples for this chapter can be found on GitHub at https://x.swiftuicookbook.com/ch14 .

WebView in SwiftUI

Sooner or later most apps need to show a web page: a help article, a checkout flow, the terms of service nobody reads. For years that meant wrapping `WKWebView` in a `UIViewRepresentable`, adopting the navigation delegate, and copying the page title and loading progress back into SwiftUI state by hand. I have shipped that wrapper in more projects than I care to count, and it always grew a coordinator class that nobody wanted to own. Starting with iOS 26, WebKit ships its own SwiftUI view. `WebView` renders web content on the same engine as Safari, and `WebPage`, the observable class behind it, exposes the page's title, URL, progress, and history as plain properties that SwiftUI can read.

In this recipe, we build a small browser. It has an address bar, a page title, a progress bar, back and forward buttons, and a status line that follows each load. We then add a contents sheet built by running JavaScript inside the page, local help pages served through a custom URL scheme, an HTTPS-only navigation policy, and an error page generated from an HTML string when a load fails. Links the web view cannot display are handed to the appropriate system app.

The preview canvas handles most of this recipe, including following links and typing addresses. Handing a URL to another app is the one

thing it cannot do, so the final tour moves to Device Hub. We say which to use as we go .

Getting ready

Create a new SwiftUI iOS app named **MiniBrowser** . The app loads pages from the web, so the Mac needs an active internet connection, whether we run the app in the preview canvas or in Device H ub.

How to do it...

Our browser starts as a single view, then grows to include a contents sheet and five supporting model files: the page model, the local pages, the heading model, a URL scheme handler, and a navigation policy. We start with the one-line web view, move the page into an observable model so the view can read its title, progress, and history, and then add the pieces that turn a viewer into a browser. These are the steps:

1. Let's start with the smallest possible browser. Replace the `ContentView` struct with the following, where `WebView(url:)` takes an optional URL and loads it, so there is nothing to unwrap. Note the `import WebKit` above it, which the Xcode template does not give us; `WebView` and everything else in this recipe ship with WebKit:

```
import SwiftUI
import WebKit
struct ContentView: View {
  var body: some View {
    NavigationStack {
      WebView(url: URL(string: "https://en.wikipedia.org/wi
```

```
    }
  }
}
```

The preview canvas shows the Wikipedia article, scrollable and tappable, with no delegate in sight.

2. A URL is enough for a viewer, but a browser needs to read the page's state and control it, and that is the job of `WebPage`, an observable class from WebKit. Create `BrowserModel.swift` in a `Models` folder, replacing the template's `import Foundation` with `import WebKit`. The model is `@MainActor` because `WebPage` is, and our views read it on the main actor anyway. `load(address:)` turns whatever the user types into a URL, defaulting to HTTPS, and `load(_:)` starts the navigation and loops over the events it returns to keep a status string current:

```
import WebKit
@MainActor
@Observable
final class BrowserModel {
  let page = WebPage()
  private(set) var status = "Idle"
  func load(address: String) async {
    var text = address.trimmingCharacters(in: .whitespaces)
    if !text.contains("://") {
      text = "https://" + text
    }
    guard let url = URL(string: text) else {
      status = "Invalid address"
      return
    }
    await load(url)
  }
  func load(_ url: URL) async {
    do {
```

```
      for try await event in page.load(URLRequest(url: url)
        status = switch event {
        case .startedProvisionalNavigation: "Connecting"
        case .receivedServerRedirect: "Redirected"
        case .committed: "Loading content"
        case .finished: "Loaded"
        @unknown default: "Unknown event"
        }
      }
    } catch {
      status = "Failed"
    }
  }
}
```

3. Hand the page to the view. Replace the `ContentView` struct with the following. The model lives in `@State`, `WebView` now takes the page instead of a URL, the status line sits above it, and `.task` loads the home page when the view appears. Because `load` waits on the navigation events, it is asynchronous, which is exactly what `.task` is for:

```
struct ContentView: View {
  @State private var browser = BrowserModel()
  var body: some View {
    NavigationStack {
      VStack(spacing: 0) {
        Text(browser.status)
          .font(.caption)
          .foregroundStyle(.secondary)
          .padding(.vertical, 4)
        WebView(browser.page)
      }
      .task {
        await browser.load(address: "en.wikipedia.org/wiki/
      }
    }
```

```
    }
}
```

In the preview canvas, the status line runs through **Connecting** and **Loading content** before settling on **Loaded** .

4. A browser needs an address bar. Add an `address` state property and a `TextField` at the top of the stack, configured for URL entry so the keyboard neither capitalizes nor autocorrects what we type, and submit it to the model. The `.onChange` keeps the field in sync with `page.url` , so following a link updates the address too. Pages we generate from a string later in the recipe report `about:blank` as their URL, so we skip that scheme to keep the typed address visible:

```
struct ContentView: View {
  @State private var browser = BrowserModel()
  @State private var address = ""
  var body: some View {
    NavigationStack {
      VStack(spacing: 0) {
        TextField("Enter an address", text: $address)
          .textFieldStyle(.roundedBorder)
          .keyboardType(.URL)
          .textInputAutocapitalization(.never)
          .autocorrectionDisabled()
          .onSubmit {
            Task { await browser.load(address: address) }
          }
          .padding(.horizontal)
        // ... the status Text and the WebView
      }
      .onChange(of: browser.page.url) {
        if let url = browser.page.url, url.scheme != "about
          address = url.absoluteString
        }
      }
```

```
        .task {
          await browser.load(address: "en.wikipedia.org/wiki/
        }
      }
    }
}
```

5. `WebPage` publishes more than the URL. Its `title` becomes the navigation title, and while `isLoading` is true, `estimatedProgress` drives a progress bar that we insert between the status line and the web view. Because the page is observable, SwiftUI re-renders these views as the properties change, with no delegate callbacks to write:

```
struct ContentView: View {
  @State private var browser = BrowserModel()
  @State private var address = ""
  var body: some View {
    NavigationStack {
      VStack(spacing: 0) {
        // ... the address TextField and the status Text
        if browser.page.isLoading {
          ProgressView(value: browser.page.estimatedProgres
        }
        WebView(browser.page)
      }
      .navigationTitle(browser.page.title)
      .navigationBarTitleDisplayMode(.inline)
      // ... the onChange and task modifiers
    }
  }
}
```

6. History lives in `page.backForwardList`. There is no `goBack()` method; instead, we load the last item of `backList` or the first

item of `forwardList`, and the same two arrays tell us when to disable the buttons. Add them in a bottom toolbar, the placement we used in the *Toolbars and the Liquid Glass Design* section of *Chapter 4*, after the title modifiers:

```
struct ContentView: View {
  @State private var browser = BrowserModel()
  @State private var address = ""
  var body: some View {
    NavigationStack {
      VStack(spacing: 0) {
        // ... the address bar, status line, progress bar,
      }
      .navigationTitle(browser.page.title)
      .navigationBarTitleDisplayMode(.inline)
      .toolbar {
        ToolbarItemGroup(placement: .bottomBar) {
          Button("Back", systemImage: "chevron.backward") {
            if let item = browser.page.backForwardList.back
              browser.page.load(item)
            }
          }
          .disabled(browser.page.backForwardList.backList.i
          Button("Forward", systemImage: "chevron.forward")
            if let item = browser.page.backForwardList.forw
              browser.page.load(item)
            }
          }
          .disabled(browser.page.backForwardList.forwardLis
        }
      }
      // ... the onChange and task modifiers
    }
  }
}
```

In the preview canvas, follow a link in the article, and the back chevron lights up: the toolbar reads the history straight from

the page.

7. Not every page comes from the network. Create `LocalPages.swift` in the `Models` folder. It holds the HTML the app serves itself: an error page built from a URL and a reason, plus two help pages we serve later in the recipe. All of them go through `wrap(title:body:)`, which supplies the document shell, and two of its lines matter more than they look. The `<title>` is what `page.title` reports for the page, and the viewport meta tag makes the page lay out at phone width instead of as a shrunken desktop page. Type the error page and the template, and copy the two help pages from the book's GitHub repository:

```
import Foundation
enum LocalPages {
  static let help: [String: String] = [
    // ... the "index" and "navigating" pages, see the GitH
  ]
  static func error(for url: URL, reason: String) -> String
    wrap(title: "Page Unavailable", body: """
      <h1>Page Unavailable</h1>
      <p>MiniBrowser could not open <code>\(url.absoluteStr
      <p>\(reason)</p>
      """)
  }
  private static func wrap(title: String, body: String) ->
    """
    <html>
    <head>
    <title>\(title)</title>
    <meta name="viewport" content="width=device-width, init
    <style>body { font-family: -apple-system; padding: 24px
    </head>
    <body>\(body)</body>
    </html>
    """
```

```
    }
  }
```

8. Now the model can show that page when a load fails. In `BrowserModel.swift`, replace the `catch` clause of `load(_:)` with the two `catch` clauses below and add `showError(for:reason:)`. WebKit reports a failed load as `NavigationError.failedProvisionalNavigation` wrapping the underlying error, whose description is the readable cause, so we match that case first and keep a plain `catch` for anything else. `load(html:)` then renders our string as a page, with no server involved:

```
@MainActor
@Observable
final class BrowserModel {
  let page = WebPage()
  private(set) var status = "Idle"
  // ... load(address:)
  func load(_ url: URL) async {
    do {
      // ... the loop over the navigation events
    } catch WebPage.NavigationError.failedProvisionalNaviga
      showError(for: url, reason: cause.localizedDescriptio
    } catch {
      showError(for: url, reason: error.localizedDescriptio
    }
  }
  private func showError(for url: URL, reason: String) {
    status = "Failed"
    page.load(html: LocalPages.error(for: url, reason: reas
  }
}
```

In the preview canvas, type an address that does not exist, such as `nonexistent.invalid`, and the error page names the URL and

explains that the server could not be found.

9. The next feature reads the page itself. Create `Heading.swift` in the `Models` folder. A heading's identity is its position in the page's list of headings, which is also the index the scroll script needs in the next step:

```
import Foundation
struct Heading: Identifiable {
  let id: Int
  let text: String
}
```

10. Add the JavaScript. In `BrowserModel.swift`, add a `headings` property and the two methods below. `callJavaScript` runs a function body inside the page and hands back whatever the body returns, so the first script returns the text of every visible `h1` and `h2` as an array, which bridges to `[String]`. The second script scrolls to one of them, and the `arguments` dictionary passes the index into the script as a variable named `index`. Both build on the same `visibleHeadings` query, so the list and the scroll can never disagree about which heading is which:

```
@MainActor
@Observable
final class BrowserModel {
  let page = WebPage()
  private(set) var status = "Idle"
  private(set) var headings: [Heading] = []
  // ... load(address:), load(_:), and showError(for:reasor
  func extractHeadings() async {
    let script = "return \(visibleHeadings).map(h => h.text
    let titles = (try? await page.callJavaScript(script)) a
    headings = titles.enumerated().map {
      Heading(id: $0.offset, text: $0.element)
```

```
    }
  }
  func scroll(to heading: Heading) async {
    let script =
      "\(visibleHeadings)[index].scrollIntoView({ behavior:
    _ = try? await page.callJavaScript(script, arguments: |
  }
  private let visibleHeadings =
    "Array.from(document.querySelectorAll('h1, h2'))"
    + ".filter(h => h.checkVisibility())"
}
```

11. Create `HeadingsView.swift` in a `Views` folder. It lists the headings, reports the tapped one through a closure, and dismisses itself. The medium and large detents are the ones we used for sheets in the *Modal Sheets and Full-Screen Covers* section of *Chapter 11*:

```
import SwiftUI
struct HeadingsView: View {
  let headings: [Heading]
  let onSelect: (Heading) -> Void
  @Environment(\.dismiss) private var dismiss
  var body: some View {
    NavigationStack {
      List(headings) { heading in
        Button(heading.text) {
          onSelect(heading)
          dismiss()
        }
      }
      .overlay {
        if headings.isEmpty {
          ContentUnavailableView("No Headings", systemImage
        }
      }
      .navigationTitle("Contents")
      .navigationBarTitleDisplayMode(.inline)
      .toolbar {
        ToolbarItem(placement: .cancellationAction) {
```

```
          Button("Close", systemImage: "xmark") {
            dismiss()
          }
        }
      }
    }
    .presentationDetents([.medium, .large])
  }
}
#Preview {
  HeadingsView(headings: [Heading(id: 0, text: "WebKit"),
                          Heading(id: 1, text: "History")])
}
```

12. Wire the sheet into `ContentView`. A second toolbar group, pushed to the trailing edge by a flexible `ToolbarSpacer`, holds a contents button that extracts the headings and then presents the sheet, and tapping a heading in the sheet scrolls the page to it:

```
struct ContentView: View {
  @State private var browser = BrowserModel()
  @State private var address = ""
  @State private var isShowingContents = false
  var body: some View {
    NavigationStack {
      VStack(spacing: 0) {
        // ... the address bar, status line, progress bar,
      }
      .navigationTitle(browser.page.title)
      .navigationBarTitleDisplayMode(.inline)
      .toolbar {
        ToolbarItemGroup(placement: .bottomBar) {
          // ... the Back and Forward buttons
        }
        ToolbarSpacer(.flexible, placement: .bottomBar)
        ToolbarItemGroup(placement: .bottomBar) {
          Button("Contents", systemImage: "list.bullet") {
            Task {
              await browser.extractHeadings()
```

```
                isShowingContents = true
              }
            }
          }
        }
        .sheet(isPresented: $isShowingContents) {
          HeadingsView(headings: browser.headings) { heading
            Task { await browser.scroll(to: heading) }
          }
        }
        // ... the onChange and task modifiers
      }
    }
  }
```

In the preview canvas, with the Wikipedia article open, tap the list icon: the sheet lists the article's section headings, and picking one scrolls the page to it.

13. Our help pages need an address to live at, and `help://index` is one the web engine has never heard of. Create `HelpSchemeHandler.swift` in the `Models` folder, again importing WebKit instead of Foundation. A `URLSchemeHandler` answers a request with an asynchronous sequence: first a `URLResponse` describing the content, then the data. An `AsyncThrowingStream` is the simplest way to produce one, and the page name comes from the URL's host, so `help://navigating` serves the `navigating` entry and an unknown name gets our error page:

```
import WebKit
struct HelpSchemeHandler: URLSchemeHandler {
  func reply(for request: URLRequest) -> AsyncThrowingStrea
    URLSchemeTaskResult, any Error
  > {
    AsyncThrowingStream { continuation in
      guard let url = request.url else {
```

```
          continuation.finish(throwing: URLError(.badURL))
          return
        }
        let html =
          LocalPages.help[url.host() ?? ""]
          ?? LocalPages.error(
            for: url,
            reason: "There is no help page with that name."
          )
        let data = Data(html.utf8)
        let response = URLResponse(
          url: url,
          mimeType: "text/html",
          expectedContentLength: data.count,
          textEncodingName: "utf-8"
        )
        continuation.yield(.response(response))
        continuation.yield(.data(data))
        continuation.finish()
      }
    }
  }
```

14. A browser also decides which navigations to allow. Create `BrowserNavigationDecider.swift` in the `Models` folder, importing WebKit. `decidePolicy(for:preferences:)` runs before every navigation, with the request in `action` and the settings for that navigation in `preferences`. For web schemes we ask WebKit to upgrade the connection to HTTPS and to fail rather than fall back, which makes this an HTTPS-only browser. Our own `help` scheme and the `about` scheme of the pages we generate pass through untouched. Any other scheme, such as a Maps or mail link, is handed to the system and canceled, because the web engine cannot display it:

```
import WebKit
struct BrowserNavigationDecider: WebPage.NavigationDeciding
  func decidePolicy(
    for action: WebPage.NavigationAction,
    preferences: inout WebPage.NavigationPreferences
  ) async -> WKNavigationActionPolicy {
    guard let url = action.request.url, let scheme = url.sc
      return .cancel
    }
    switch scheme {
    case "http", "https":
      preferences.preferredHTTPSNavigationPolicy = .errorOr
      return .allow
    case "help", "about":
      return .allow
    default:
      await UIApplication.shared.open(url)
      return .cancel
    }
  }
}
```

15. Both pieces plug into the page through its configuration, and a `WebPage` takes its configuration and its navigation decider when it is created, so the property becomes a declaration plus an initializer. In `BrowserModel.swift`, replace `let page = WebPage()` with the following. The `URLScheme` initializer is failable because it validates the scheme name, so we unwrap the result before registering the handler, which routes every `help://` request to our struct:

```
@MainActor
@Observable
final class BrowserModel {
  let page: WebPage
  private(set) var status = "Idle"
```

```
  private(set) var headings: [Heading] = []
  init() {
    var configuration = WebPage.Configuration()
    if let scheme = URLScheme("help") {
      configuration.urlSchemeHandlers[scheme] = HelpSchemeH
    }
    page = WebPage(configuration: configuration,
                   navigationDecider: BrowserNavigationDeci
  }
  // ... the load, showError, extractHeadings, and scroll m
  // ... and the visibleHeadings query
}
```

16. Finish the view by adding help and find buttons, then enable the back and forward swipe gestures. The help button loads the help index through the same address path as everything else, and the find button is backed by `findNavigator`, which gives the web view the system's find-in-page bar:

```
struct ContentView: View {
  @State private var browser = BrowserModel()
  @State private var address = ""
  @State private var isShowingContents = false
  @State private var isFinding = false
  var body: some View {
    NavigationStack {
      VStack(spacing: 0) {
        // ... the address bar, status line, and progress b
        WebView(browser.page)
          .findNavigator(isPresented: $isFinding)
          .webViewBackForwardNavigationGestures(.enabled)
      }
      .navigationTitle(browser.page.title)
      .navigationBarTitleDisplayMode(.inline)
      .toolbar {
        // ... the Back and Forward group and the ToolbarSp
        ToolbarItemGroup(placement: .bottomBar) {
          // ... the Contents button
          Button("Find", systemImage: "magnifyingglass") {
```

```
                isFinding = true
            }
            Button("Help", systemImage: "questionmark.circle"
                Task { await browser.load(address: "help://inde
            }
          }
        }
        // ... the sheet, onChange, and task modifiers
      }
    }
  }
```



The back and forward swipes are edge gestures, and they are awkward to perform with a mouse or trackpad in Device Hub: the drag has to start at the very edge of the screen, and it often registers as a scroll instead. They work as expected on a device. In Device Hub, use the toolbar chevrons.

17. Run the app in Device Hub rather than the preview canvas, because this tour needs things the canvas cannot do: the Maps link hands a URL to another app, and the address bar's keyboard and the find bar are more dependable there. Tap the list icon and pick a heading to scroll to that section. Tap the magnifying glass icon and type a word from the article, such as `Safari`, into the find bar: the page highlights every match, and the bar's arrows step through them. Type `http://example.com` and watch the address bar change to `https://`, because our decider asked for the upgrade. Type `http://httpforever.com`, a site that serves plain HTTP only, and land on our error page with the TLS failure as

the reason. Tap the question mark icon for the help index, follow its link to the second page, and tap the Maps link there to watch the system open Maps instead of the browser. For the complete code, see the book's GitHub repository.

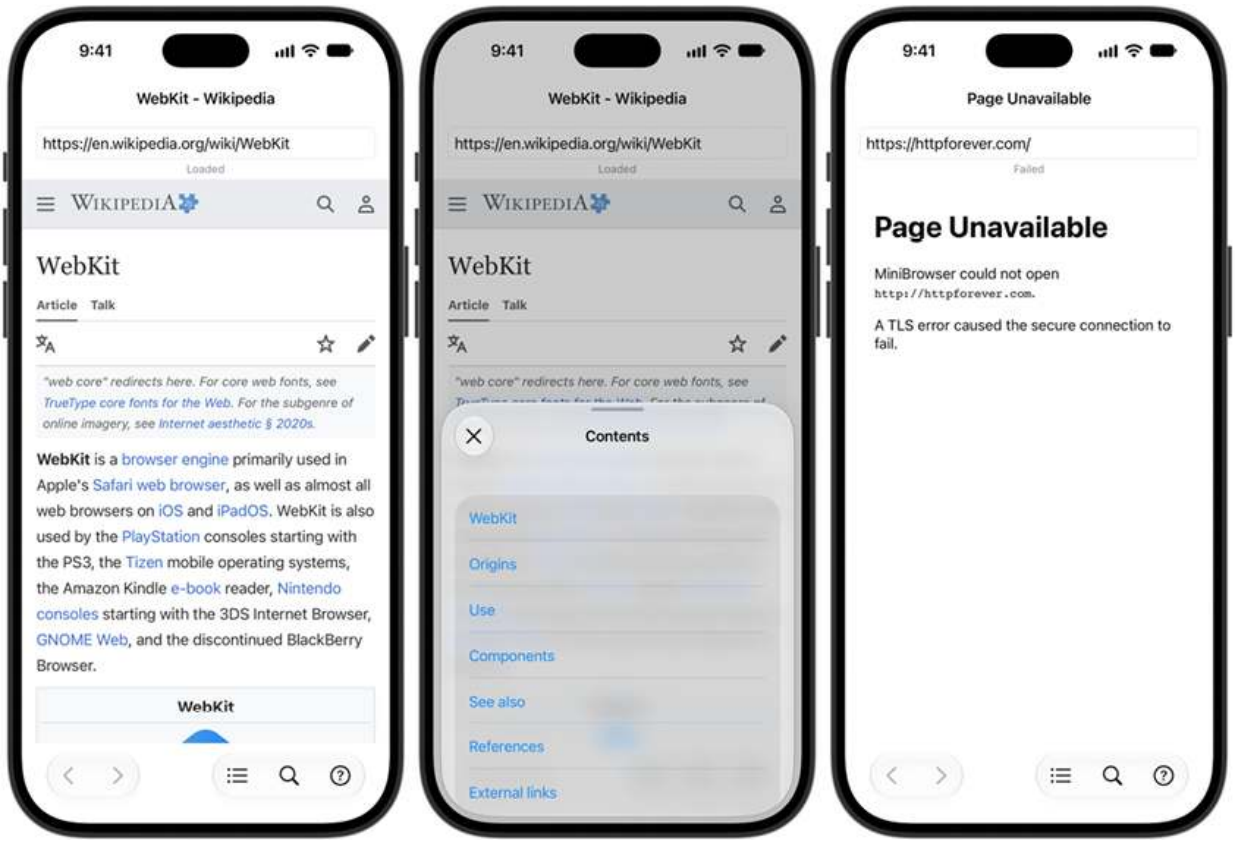


Figure 14.1: The Wikipedia article with the status line and bottom toolbar, the contents sheet, and the error page for an HTTP-only site

How it works...

The two initializers of `WebView` mark the boundary between a viewer and a browser. `WebView(url:)` creates and owns a page we never see, which is all a help article needs. `WebView(browser.page)` renders a `WebPage` we own. Because `WebPage` is `@Observable`, changes to `title`, `url`, `isLoading`, `estimatedProgress`, and `backForwardList` invalidate only the

views that read those properties. This is the same per-property tracking we studied in the *Observation Framework* section of *Chapter 5*. The same object is the control surface. Our browser already calls `load` with a `URLRequest`, an HTML string, and an item from the history list, and the same family accepts a plain `URL` or raw data with a MIME type, with `reload()` and `stopLoading()` completing the set; `hasOnlySecureContent` and `themeColor` are there too, ready for a lock icon or a tinted toolbar. In our browser, that one object replaces the delegate, the coordinator, and the state copying that used to sit between them.

Every `load` call returns the navigation as an asynchronous sequence of `NavigationEvent` values, which is what our status line consumes: `startedProvisionalNavigation` when the request goes out, `receivedServerRedirect` if the server bounces it, `committed` once content starts arriving, and `finished` when the page is complete, with a failure thrown as a `NavigationError` instead of delivered as an event. The page-wide `navigations` property publishes the same events for every navigation, including the links the user taps inside the page, but the per-load sequence also tells us that the URL we asked for is the one that failed, which is what the error page needs.

Watch the decider's timing, though: `decidePolicy(for:preferences:)` runs before the provisional navigation starts, so a `.cancel` produces no event and no error. The sequence does not finish either, so the loop stays suspended without ever running its body again, and the status line stays where it was. That is why our policy pairs `.cancel` with handing the URL to `UIApplication.shared.open`, the same hand-off to another app that `openURL` performs in the *Deep Linking and URL Handling* section of *Chapter 11*, and why the HTTPS-only rule goes

through `preferences` instead: an upgrade that fails is a real navigation failure, so it throws and lands on the error page. We scope that preference to `http` and `https` deliberately, because applying it to a custom scheme leaves an empty page that still reports `finished`. `preferredHTTPSNavigationPolicy` is one of several knobs on `NavigationPreferences`; `preferredContentMode` requests the mobile or desktop version of a site, and `allowsContentJavaScript` switches scripting off for a navigation.

`URLSchemeHandler` is how an app becomes the server for its own scheme. WebKit handles `http`, `https`, `file`, and `about` itself; for anything else registered in `Configuration.urlSchemeHandlers`, it calls `reply(for:)` and consumes the sequence we return. A plain `URLResponse` with the MIME type and encoding is enough for the engine to render our HTML. Because handlers and the navigation decider are part of the page's configuration, they are fixed for the page's lifetime, which is why `BrowserModel` builds them in its initializer rather than assigning them later. Under the project's default main-actor isolation, both `HelpSchemeHandler` and `BrowserNavigationDecider` conform to their protocols without any concurrency annotations, and the model itself is marked `@MainActor` to match the page it owns.

The JavaScript bridge has a few rules worth remembering. The string we pass to `callJavaScript` is treated as the body of a function, so a script that produces a value must return that value explicitly, and the result arrives as an optional `Any` that bridges to Swift types the way JSON does: an array of strings becomes `[String]` with one cast. The `arguments` dictionary turns each key into a variable the body can read, which is how the scroll script received `index` without any string interpolation of user data into code. We filter the headings through

`checkVisibility()` because pages hide markup the user never sees, and Wikipedia's own table-of-contents heading is a good example: it is in the document but not rendered on a phone, so scrolling to it would do nothing. The scroll script returns `undefined`, and the result is unused, which is what the `_ =` acknowledges. Both scripts run in the page's default content world; the optional `contentWorld` argument isolates our code from the page's own scripts when that matters.

Around the web view, `findNavigator` adds the system find bar the same way it does for any text view, and `webViewBackForwardNavigationGestures` belongs to a family of modifiers that also covers magnification, link previews, text selection, and scroll position. They configure the engine's behavior from SwiftUI without to uching the page.

See also

- `WebView` documentation: https://x.swiftuicookbook.com/z6w46
- `WebPage` documentation: https://x.swiftuicookbook.com/wp4g5
- `URLSchemeHandler` documentation: https://x.swiftuicookbook.com/fkrtz
- `WebPage.NavigationDeciding` documentation: https://x.swiftuicookbook.com/x3onh
- `callJavaScript(_:arguments:in:contentWorld:)` documentation: https://x.swiftuicookbook.com/4uio6
- Meet WebKit for SwiftUI (WWDC 2025): https://x.swiftuicookbook.com/mqdx7

Rich Text with AttributedString

Notes, comments, and messages all reach the point where plain text is not enough: a word needs to be bold, a warning needs to be red, a heading needs to stand out. In UIKit that meant a `UITextView`, an `NSAttributedString` with its dictionary-based attributes, a delegate to watch the selection, and a good deal of index arithmetic. I have written that editor more than once, and every version ended up with its formatting logic spread across the view controller. Starting with iOS 26, `TextEditor` binds directly to an `AttributedString`, the Swift-native value type for styled text, and the rest of the machinery (selection tracking, typing attributes, and formatting rules) comes with it.

In this recipe, we build a small note editor with deliberately limited formatting: bold, italic, underline, and four note colors, and nothing else. We start with the editor alone, which already supports formatting through the system's edit menu and keyboard shortcuts, and then build our own formatting bar on top of a selection binding. The bar derives each button's selected state from the selection's typing attributes, resolves fonts through the environment so that bold and italic mean what the user sees, and applies changes with the two transform methods `AttributedString` gains in SwiftUI, one for an insertion point and one for a range of text. We finish with a formatting definition: a declaration of which attributes the editor accepts at all, plus a constraint that snaps any text color outside our palette back to the default, whether it arrives from our bar, from the edit menu, or from a paste.

Everything in this recipe runs in the preview canvas, keyboard and selection included, so we stay there until the final step, where pasting text copied from another app is a j ob for Device Hub.

Getting ready

Create a new SwiftUI iOS a pp named **RichNotes** .

How to do it...

Our app is one editor view, a formatting bar as its own view, and three small model files: the sample note, the color palette, and the formatting definition. We build the editor first, because binding it to an `AttributedString` already gives us a working rich text editor, and then add our own controls and rules one at a time. These are the steps:

1. Let's start with something to edit. Create `SampleNote.swift` in a `Models` folder, replacing the template's `import Foundation` with `import SwiftUI` , because the `font` attribute we set belongs to SwiftUI's attribute scope. The note is a title line with a bold title font followed by a plain sentence, so the editor opens with two visibly different runs of text:

```
import SwiftUI
enum SampleNote {
  static var text: AttributedString {
    var title = AttributedString("Trip planning\n")
    title.font = .title2.bold()
    let body = AttributedString("Book the ferry, pack the r
      + " and confirm the hotel before Friday.")
    return title + body
```

```
  }
}
```

2. Replace the `ContentView` struct with the following. The only difference from a plain text editor is the type of the state: `TextEditor` bound to an `AttributedString` renders every run with its own attributes and preserves those attributes during editing:

```
struct ContentView: View {
  @State private var text = SampleNote.text
  var body: some View {
    NavigationStack {
      TextEditor(text: $text)
        .padding(.horizontal)
        .navigationTitle("Rich Notes")
        .navigationBarTitleDisplayMode(.inline)
    }
  }
}
```

   In the preview canvas, the title renders in its bold title font and the sentence in the body font. Select a word and the edit menu offers formatting commands. We did not write any custom formatting code to get them.

3. The system controls are a useful baseline, but a practical editor needs its own bar, so create `FormattingBar.swift` in a `Views` folder. The bar binds to the text and its `AttributedTextSelection` and reads `fontResolutionContext` from the environment. Its bold button uses `typingAttributes(in:)` to inspect the selection, `resolve(in:)` to check the rendered font, and `transformAttributes(in:)` to apply the change. I always give a control like this its own preview, so the

bar can be tried before it is wired into the app; here the preview pairs it with a small editor:

```
import SwiftUI
struct FormattingBar: View {
  @Binding var text: AttributedString
  @Binding var selection: AttributedTextSelection
  @Environment(\.fontResolutionContext) private var fontRes
  var body: some View {
    HStack {
      Button("Bold", systemImage: "bold") {
        toggleBold()
      }
      .tint(isBold ? Color.accentColor : Color.secondary)
    }
    .labelStyle(.iconOnly)
    .buttonStyle(.bordered)
    .padding(.horizontal)
    .padding(.vertical, 8)
    .background(.bar)
  }
  private var typingFont: Font.Resolved {
    let font = selection.typingAttributes(in: text).font ??
    return font.resolve(in: fontResolutionContext)
  }
  private var isBold: Bool {
    typingFont.isBold
  }
  private func toggleBold() {
    let makeBold = !isBold
    text.transformAttributes(in: &selection) { attributes i
      attributes.font = (attributes.font ?? .body).bold(mak
    }
  }
}
#Preview {
  @Previewable @State var text = SampleNote.text
  @Previewable @State var selection = AttributedTextSelecti
  VStack(spacing: 0) {
    TextEditor(text: $text, selection: $selection)
      .padding(.horizontal)
    FormattingBar(text: $text, selection: $selection)
```

```
    }
}
```

In the preview canvas, tap into the sentence, tap the bold icon, and type: the new characters are bold. Move the insertion point into the title and the icon lights up, because the title's font resolves as bold.

4. Give `ContentView` its own selection and the bar. Replace the `ContentView` struct with the following, where the editor now takes the `selection` binding as well, and `safeAreaInset(edge: .bottom)` places the bar below the editor and moves it up with the keyboard:

```
struct ContentView: View {
  @State private var text = SampleNote.text
  @State private var selection = AttributedTextSelection()
  var body: some View {
    NavigationStack {
      TextEditor(text: $text, selection: $selection)
        .padding(.horizontal)
        .safeAreaInset(edge: .bottom) {
          FormattingBar(text: $text, selection: $selection)
        }
        .navigationTitle("Rich Notes")
        .navigationBarTitleDisplayMode(.inline)
    }
  }
}
```

In the preview canvas, select the title and the first few words of the sentence, then tap the bold icon: the selected words jump to the title's size, and the title loses its bold. The bar treated the whole selection as if it were the title, and the next step fixes that.

5. Bold works at an insertion point, but a selection spanning different fonts requires run-by-run updates. In `FormattingBar.swift`, add an `update` method that uses `indices(in:)` to distinguish an insertion point from selected ranges, and route `toggleBold` through it. An insertion point still uses `transformAttributes(in:)`; for ranges, we update each run on a copy and replace the original text through `transform(updating:)`, which keeps the selection valid:

```
struct FormattingBar: View {
  // ... the properties, body, typingFont, and isBold
  private func toggleBold() {
    let makeBold = !isBold
    update { attributes in
      attributes.font = (attributes.font ?? .body).bold(mak
    }
  }
  private func update(_ change: (inout AttributeContainer)
    switch selection.indices(in: text) {
    case .insertionPoint:
      text.transformAttributes(in: &selection, body: change
    case .ranges(let ranges):
      var updated = text
      for run in updated[ranges].runs {
        var attributes = run.attributes
        change(&attributes)
        updated[run.range].setAttributes(attributes)
      }
      text.transform(updating: &selection) { text in
        text.replaceSubrange(text.startIndex..<text.endInde
      }
    }
  }
}
```

Select the whole note in the preview canvas and tap the bold icon twice: the title keeps its size, the sentence keeps its own, and the selection survives both taps.

6. Italic follows the same shape as bold, with `isItalic` from the resolved font and `italic(_:)` on the `Font`. Add the button after the bold one and the two members below:

```
struct FormattingBar: View {
  // ... the properties
  var body: some View {
    HStack {
      // ... the Bold button and its tint
      Button("Italic", systemImage: "italic") {
        toggleItalic()
      }
      .tint(isItalic ? Color.accentColor : Color.secondary)
    }
    // ... the label style, button style, padding, and back
  }
  // ... typingFont and isBold
  private var isItalic: Bool {
    typingFont.isItalic
  }
  // ... toggleBold()
  private func toggleItalic() {
    let makeItalic = !isItalic
    update { attributes in
      attributes.font = (attributes.font ?? .body).italic(m
    }
  }
  // ... update(_:)
}
```

7. Underline is not part of the font; it is an attribute of its own, and two frameworks define one by that name. UIKit's `underlineStyle` holds an `NSUnderlineStyle` that the SwiftUI editor ignores, so we

go through the `swiftUI` attribute scope explicitly, both when reading the state and when setting it. Add the button and the two members:

```
struct FormattingBar: View {
  // ... the properties
  var body: some View {
    HStack {
      // ... the Bold and Italic buttons
      Button("Underline", systemImage: "underline") {
        toggleUnderline()
      }
      .tint(isUnderlined ? Color.accentColor : Color.secon
    }
    // ... the label style, button style, padding, and back
  }
  // ... typingFont, isBold, and isItalic
  private var isUnderlined: Bool {
    selection.typingAttributes(in: text).swiftUI.underline
  }
  // ... toggleBold() and toggleItalic()
  private func toggleUnderline() {
    let underline = !isUnderlined
    update { attributes in
      attributes.swiftUI.underlineStyle = underline ? .sing
    }
  }
  // ... update(_:)
}
```

Back in the preview canvas, select a few words and tap the underline icon; tap it again to remove the underline.

8. Colors need a palette. Create `NotePalette.swift` in the `Models` folder, importing SwiftUI for `Color`. Four named cases keep the menu readable, and the `colors` array is what the formatting definition checks against later:

```
import SwiftUI
enum NotePalette: String, CaseIterable, Identifiable {
  case red, green, blue, purple
  var id: Self { self }
  var color: Color {
    switch self {
    case .red: .red
    case .green: .green
    case .blue: .blue
    case .purple: .purple
    }
  }
  static let colors = allCases.map(\.color)
}
```

9. Add a color menu at the trailing end of the bar. Each entry applies its color through the same `update` path, so a colored word keeps its bold, and the **Default** entry clears the attribute by writing `nil`:

```
struct FormattingBar: View {
  // ... the properties
  var body: some View {
    HStack {
      // ... the Bold, Italic, and Underline buttons
      Spacer()
      Menu("Color", systemImage: "paintpalette") {
        ForEach(NotePalette.allCases) { entry in
          Button(entry.rawValue.capitalized) {
            apply(entry.color)
          }
        }
        Button("Default") {
          apply(nil)
        }
      }
    }
    // ... the label style, button style, padding, and back
  }
```

```
  // ... typingFont, isBold, isItalic, isUnderlined, and th
  private func apply(_ color: Color?) {
    update { attributes in
      attributes.foregroundColor = color
    }
  }
  // ... update(_:)
}
```

In the preview canvas, select the word **ferry** , tap the bold icon, then pick **Red** from the palette icon: the word is red and still bold.

10. So far the editor accepts any attribute anyone hands it, including strikethrough from the edit menu and whatever formatting a paste brings along. An `AttributedTextFormattingDefinition` is where we say what the editor supports. Create `NoteFormatting.swift` in the `Models` folder, importing SwiftUI. Its `Scope` lists exactly three attributes, and its `body` , a builder of formatting rules, is empty for now:

```
import SwiftUI
struct NoteFormatting: AttributedTextFormattingDefinition {
  struct Scope: AttributeScope {
    let font: AttributeScopes.SwiftUIAttributes.FontAttribu
    let foregroundColor:
      AttributeScopes.SwiftUIAttributes.ForegroundColorAttr
    let underlineStyle:
      AttributeScopes.SwiftUIAttributes.UnderlineStyleAttri
  }
  var body: some AttributedTextFormattingDefinition<Scope>
  }
}
```

11. Attach the definition to the editor. In `ContentView.swift`, add the modifier right after the padding:

```
struct ContentView: View {
  @State private var text = SampleNote.text
  @State private var selection = AttributedTextSelection()
  var body: some View {
    NavigationStack {
      TextEditor(text: $text, selection: $selection)
        .padding(.horizontal)
        .attributedTextFormattingDefinition(NoteFormatting(
        // ... the safeAreaInset and the navigation title m
    }
  }
}
```

    Select a word in the preview canvas and open the edit menu: the formatting commands now offer only what the scope allows, and strikethrough is gone.

12. The scope decides which attributes exist; a constraint decides which values they may hold. Add a `PaletteColors` constraint to `NoteFormatting.swift` and list it in the definition's `body`. The constraint names the attribute it governs, and `constrain(_:)` receives a proxy for one run's attributes: a foreground color that is not in the palette becomes `nil`, which is the default text color:

```
struct NoteFormatting: AttributedTextFormattingDefinition {
  // ... the Scope
  var body: some AttributedTextFormattingDefinition<Scope>
    PaletteColors()
  }
}
struct PaletteColors: AttributedTextValueConstraint {
  typealias Scope = NoteFormatting.Scope
```

```
    typealias AttributeKey = AttributeScopes.SwiftUIAttribute
      .ForegroundColorAttribute
    func constrain(_ container: inout Attributes) {
      if let color = container.foregroundColor,
        !NotePalette.colors.contains(color)
      {
        container.foregroundColor = nil
      }
    }
  }
```

13. Run the app in Device Hub for this last check, because it needs another app: open Safari, select and copy a line that contains a link, switch back to **RichNotes** , and paste it into the note. The link disappears because `link` is outside the scope, while `PaletteColors` clears its foreground color because that color is outside our palette. Bold and italic remain because the scope allows the `font` attribute that carries them. Then keep going with the bar: select across the title and the sentence and toggle bold, color a word and un-bold it, and watch the icons light up as the insertion point moves through the formatting. With the Mac keyboard, *Cmd + B* , *Cmd + I* , and *Cmd + U* toggle the same styles, and the icons follow. The shortcuts only reach the app in Device Hub; in the preview canvas, Xcode takes them for the **Build** , **Profile** , and **Test** commands in its **Product** menu. For the complete code, see the book's GitHub repository.

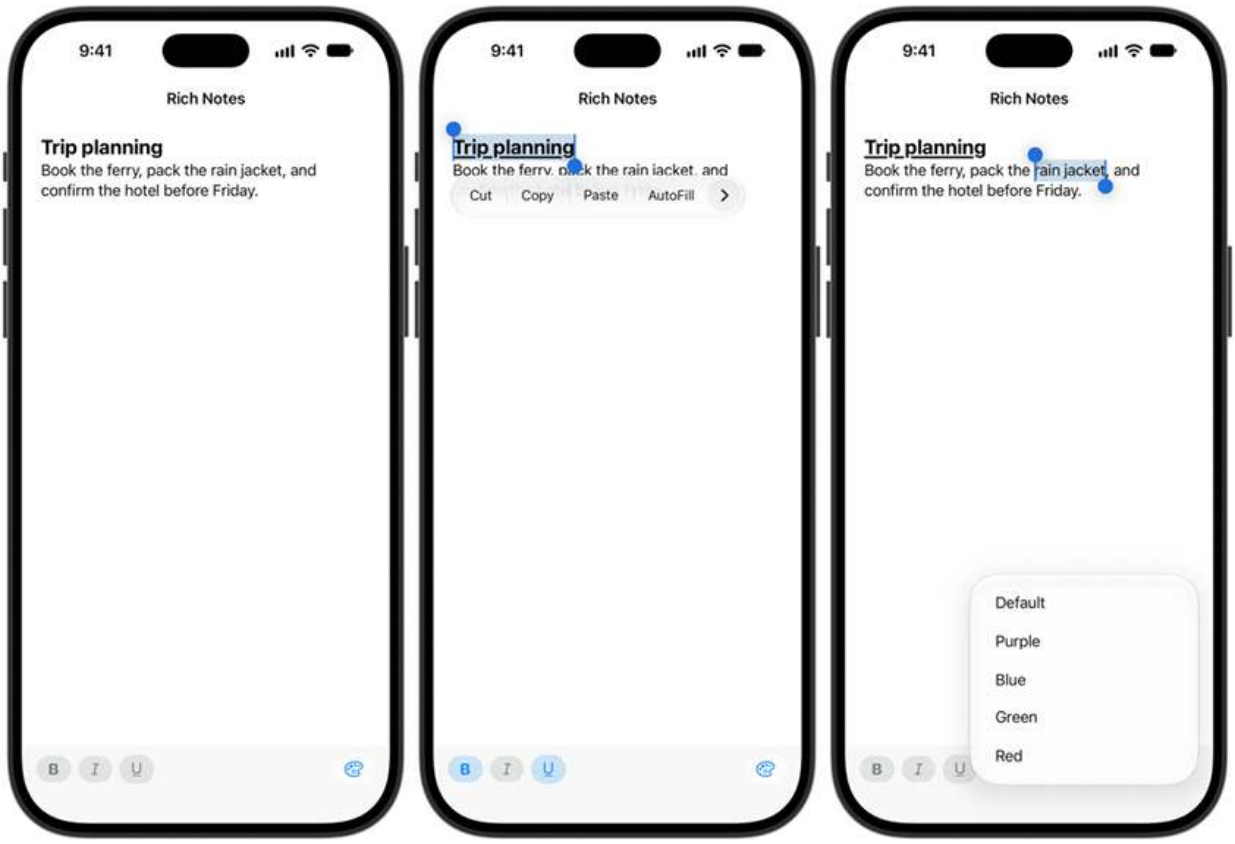


Figure 14.2: The note with the formatting bar, a mixed selection with the bold and underline ico ns lit, and the color menu

How it works…

`ContentView` in step 2 became a rich text editor because its binding is an `AttributedString` rather than a `String` . The string stores each run's font, color, and underline, and the system formatting commands update those attributes directly. `AttributedTextSelection` is how our code learns what the user is pointing at. Its `indices(in:)` reports either an insertion point or a set of ranges, and `typingAttributes(in:)` returns the attributes that the next typed character would get, which is also the honest source for a toolbar's state: a bold icon should light up when typing would produce bold, whether because a bold word is selected or because the caret sits at its end. Those attributes only hold

what was set explicitly, which is why the bar falls back to `.body` when a run has no font at all.

Deciding whether a font is bold sounds trivial until Dynamic Type, weights, and design variants enter the picture. A `Font` is a recipe, not a typeface, and `resolve(in:)` with the `fontResolutionContext` from the environment produces the concrete `Font.Resolved`, with `isBold`, `isItalic`, `pointSize`, and `weight` as the user actually sees them. Reading the context from the environment, the mechanism we covered in the *Reading and Defining Environment Values* section of *Chapter 6*, makes our bold and italic checks follow Dynamic Type and other font settings instead of inferring those traits from an unresolved `Font`. Toggling then means applying `bold(_:)` or `italic(_:)` to the run's own font, which is how the title keeps its size when the sentence next to it changes.

The two transform methods exist for the two shapes a selection can take. With an insertion point, there is no text to change, only the typing attributes, and `transformAttributes(in:)` updates them in the selection so the next keystroke picks them up. With a range, that same method treats the selection as one unit, seeding its closure from the first run and writing the whole container back, which flattens every difference inside the selection. Transforming the runs ourselves, from `text[ranges].runs`, keeps each run's other attributes intact, and working on a copy preserves the selection's relationship to the original value while we update every selected run. We then install that copy with one `replaceSubrange` call inside `transform(updating:)`, allowing SwiftUI to move the selection to the updated text. Mutating the original text one run at a time would invalidate the selection between edits.

A formatting definition is a contract between the editor and the app. The `Scope` is the allow list: attributes outside it never enter the string, the edit menu's formatting commands shrink to match, and a paste is filtered on the way in, which is what removed the link in step 13, while the constraint cleared its color. The `body` adds rules on top, each an `AttributedTextValueConstraint` bound to one attribute key. The editor runs the rules on every change, from typing, from the menu, from a paste, and from our own binding, so the constraint is the one place where the palette is enforced rather than a check scattered across every control. Inside `constrain(_:)` the proxy lets a rule read the other attributes in scope, so a constraint can also derive one attribute from another, such as coloring anything set in a heading font. The definition does not restrict what our own code writes into `text`, only what the editor keeps, which is exactly the boundary an app wants.

One naming trap deserves a mention because the compiler will not flag it. Foundation, UIKit, and SwiftUI each contribute an attribute scope to `AttributedString`, and UIKit's `underlineStyle` shares its name with SwiftUI's while holding an `NSUnderlineStyle`. A bare `.single` can resolve to either, and the UIKit attribute is invisible to the SwiftUI editor and outside our scope, so it silently disappears. Spelling `swiftUI.underlineStyle` removes the ambiguity; the font and color assignments need no such care because a `Font` or a `Color` value already selects the SwiftUI key.

Beyond editing, the same `AttributedString` is `Codable`, so saving a note is a `JSONEncoder` call away, and it can be created from Markdown as well, which is where the *PDF, Markdown and Cu stom Fonts* section picks up.

See also

- `TextEditor` `init(text:selection:)` documentation: https://x.swiftuicookbook.com/yxfxs
- `AttributedTextSelection` documentation: https://x.swiftuicookbook.com/0qn1x
- `AttributedTextFormattingDefinition` documentation: https://x.swiftuicookbook.com/z4sas
- `AttributedTextValueConstraint` documentation: https://x.swiftuicookbook.com/2lido
- *Building rich SwiftUI text experiences* article: https://x.swiftuicookbook.com/0tw9g
- Code-along: Cook up a rich text experience in SwiftUI with AttributedString (WWDC 2025): https://x.swiftuicookbook.com/b8gq1

PDF, Markdown and Custom Fonts

Release notes, help pages, and the occasional article all reach an app as text with a little formatting in it, and Markdown has become the way that text is written. For years, showing it meant either building an `NSAttributedString` by hand or converting the Markdown to HTML and handing it to a web view, and I have shipped both. Neither is needed for a few paragraphs anymore: `Text` parses inline Markdown on its own, and `AttributedString` parses a whole document, headings and lists included, into a form a SwiftUI view can lay out.

In this recipe, we build a small article viewer. We begin with `Text` and a Markdown string to see what `Text` parses, what it leaves

untouched, and why a string literal behaves differently from a `String` value. We then build a block-level renderer with `AttributedString(markdown:)`, set the article in the bundled Jost typeface, loaded with `Font.custom(_:size:relativeTo:)` so it still scales with Dynamic Type, and export the same view with `ImageRenderer`. A `UIViewRepresentable` wrapper around PDFKit's `PDFView` displays the PDF, and a `ShareLink` shares the file.

The preview canvas covers everything except sharing the file; the last step moves to Device Hub because saving the PDF ha nds the file to another app.

Getting ready

Create a new SwiftUI iOS app named **MarkdownPress**. The recipe also needs four Jost font files: `Jost-400-Book.ttf`, `Jost-400-BookItalic.ttf`, `Jost-700-Bold.ttf`, and `Jost-700-BoldItalic.ttf`. They are in the `ch14/assets/` folder of the book's GitHub repository, together with the typeface's license; download them now, since we add t hem to the project in step 8.

How to do it...

Our app has an article view, a reusable Markdown renderer, a PDFKit wrapper, and a PDF export view. Four supporting files provide the sample article, the block model, the Jost font extension, and the PDF exporter. We begin with `Text`, because seeing exactly where its Markdown support stops is the best argument for the block renderer that follows; once the renderer is set in Jost, we add the PDF export, which reuses that same view. These are the steps:

1. Let's start with the article. Create `SampleArticle.swift` in a `Models` folder and give it the Markdown source as a multiline string: a title, a paragraph with the inline styles we care about, a subheading, a short list, and a link. The blank lines are part of the Markdown, since they separate the paragraphs, and a line wrapped inside a paragraph is a soft break, which a Markdown parser turns into a space:

```
import Foundation
enum SampleArticle {
  static let markdown = """
    # Meet Jost

    Jost is a *geometric* sans-serif typeface designed by C
    published by **indestructible type** under the SIL Oper
    Its round letters and even strokes echo the German type
    1920s, which makes it a good fit for headings, captions
    articles like this one.

    ## Using it in an app

    Bundling a font takes three steps:

    - Add the `.ttf` files to the project.
    - List them under `UIAppFonts` in the `Info.plist` file
    - Load them with `Font.custom(_:size:relativeTo:)`.

    The typeface is free to use. See the
    [Jost project on GitHub](https://github.com/indestructi
    for the complete family and its license.
    """
}
```

2. Replace the `ContentView` struct with the following, which scrolls the article inside a navigation stack. The

`.frame(maxWidth:alignment:)` modifier keeps the text leading-aligned at the full width instead of centered in the scroll view:

```
struct ContentView: View {
  var body: some View {
    NavigationStack {
      ScrollView {
        Text(SampleArticle.markdown)
          .frame(maxWidth: .infinity, alignment: .leading)
          .padding()
      }
      .navigationTitle("MarkdownPress")
      .navigationBarTitleDisplayMode(.inline)
    }
  }
}
```

In the preview canvas, the article appears exactly as typed, asterisks, hash marks, and backticks included. `Text` received a `String`, and a `String` is displayed as is.

3. Wrap the same string in a `LocalizedStringKey`, the type a string literal becomes when we write it directly inside `Text`:

```
struct ContentView: View {
  var body: some View {
    NavigationStack {
      ScrollView {
        Text(LocalizedStringKey(SampleArticle.markdown))
          .frame(maxWidth: .infinity, alignment: .leading)
          .padding()
      }
      // ... the navigation title modifiers
    }
  }
}
```

Now the italic, bold, and code spans render and the link is tappable, but the hash marks and the dashes are still there, and every wrapped line still breaks where we typed it. `Text` parses inline Markdown only; headings, lists, and paragraphs are outside its vocabulary.

4. A viewer that can show its own source is handy while we work on the renderer, and it is where `Text(verbatim:)` belongs. Add a toggle to the toolbar and, when it is on, show the raw Markdown in a monospaced font:

```
struct ContentView: View {
  @State private var isShowingSource = false
  var body: some View {
    NavigationStack {
      ScrollView {
        Group {
          if isShowingSource {
            Text(verbatim: SampleArticle.markdown)
              .monospaced()
          } else {
            Text(LocalizedStringKey(SampleArticle.markdown)
          }
        }
        .frame(maxWidth: .infinity, alignment: .leading)
        .padding()
      }
      .navigationTitle("MarkdownPress")
      .navigationBarTitleDisplayMode(.inline)
      .toolbar {
        ToolbarItem(placement: .topBarLeading) {
          Toggle("Source",
                 systemImage: "chevron.left.forwardslash.ch
                 isOn: $isShowingSource)
        }
      }
    }
```

```
  }
}
```

The toggle renders as the code icon at the leading edge of the navigation bar and lights up while it is on. In the preview canvas, tap it to flip between the two renderings.

5. `Text` stops at inline Markdown, but `AttributedString` preserves a document's block structure through presentation intents. Create `MarkdownBlock.swift` in the `Models` folder. Its `blocks(from:)` method walks those intents one block at a time, while `kind(of:)` examines their components to distinguish headings and list items from ordinary paragraphs:

```
import Foundation
struct MarkdownBlock: Identifiable {
  enum Kind {
    case paragraph
    case heading(level: Int)
    case listItem
  }
  let id: Int
  let kind: Kind
  let text: AttributedString
  static func blocks(from markdown: String) -> [MarkdownBlc
    guard let article = try? AttributedString(markdown: mar
      return []
    }
    var blocks: [MarkdownBlock] = []
    for (intent, range) in article.runs[\.presentationInter
      let block = MarkdownBlock(id: blocks.count,
                                kind: kind(of: intent),
                                text: AttributedString(arti
      blocks.append(block)
    }
    return blocks
  }
```

```
  private static func kind(of intent: PresentationIntent?)
    for component in intent?.components ?? [] {
      switch component.kind {
      case .header(let level):
        return .heading(level: level)
      case .listItem:
        return .listItem
      default:
        continue
      }
    }
    return .paragraph
  }
}
```

6. With the blocks in hand, rendering is a stack of `Text` views. Create `MarkdownView.swift` in a `Views` folder. Headings get a bold title font sized by their level, list items get a bullet, and every block keeps its inline styles, because each `Text` takes the block's `AttributedString`:

```
import SwiftUI
struct MarkdownView: View {
  private let blocks: [MarkdownBlock]
  init(markdown: String) {
    blocks = MarkdownBlock.blocks(from: markdown)
  }
  var body: some View {
    VStack(alignment: .leading, spacing: 12) {
      ForEach(blocks) { block in
        switch block.kind {
        case .heading(let level):
          Text(block.text)
            .font(.system(level == 1 ? .title : .title2, we
        case .listItem:
          HStack(alignment: .firstTextBaseline) {
            Text("•")
            Text(block.text)
          }
```

```
          .padding(.leading)
        case .paragraph:
          Text(block.text)
        }
      }
    }
  }
}
#Preview {
  ScrollView {
    MarkdownView(markdown: SampleArticle.markdown)
      .padding()
  }
}
```

In the preview canvas, the title, the subheading, and the three bullets are laid out as blocks for the first time, and the wrapped lines have rejoined their paragraphs.

7. Back in `ContentView.swift`, put the renderer in place of the `LocalizedStringKey` text; the source toggle keeps working as before:

```
struct ContentView: View {
  @State private var isShowingSource = false
  var body: some View {
    NavigationStack {
      ScrollView {
        Group {
          if isShowingSource {
            Text(verbatim: SampleArticle.markdown)
              .monospaced()
          } else {
            MarkdownView(markdown: SampleArticle.markdown)
          }
        }
        // ... the frame and padding modifiers
      }
      // ... the navigation title and toolbar modifiers
    }
```

```
    }
}
```

8. The system font is fine, but a publication usually has a typeface of its own, and bundling one takes a resource and a registration. Drag the four Jost files downloaded in *Getting ready* into the Project navigator, inside a new `Fonts` folder, so they are copied into the project and bundled with the app. There is no code to write in this step or in the next one.
9. Register the files with the target. Select the project in the Project navigator, choose the **MarkdownPress** target, and open the **Info** tab. Under **Custom iOS Target Properties** , hover over any row and click the plus button that appears next to its key, then type `Fonts provided by application` as the new key; Xcode completes the name and sets its type to **Array** . Expand the new row with its disclosure triangle: Xcode has already added **Item 0** , so click the row's plus button three more times to add **Item 1** through **Item 3** , and enter one file name as the value of each: `Jost-400-Book.ttf` , `Jost-400-BookItalic.ttf` , `Jost-700-Bold.ttf` , and `Jost-700-BoldItalic.ttf` . The values are file names, extension included, not font names. Xcode adds an `Info.plist` file to the project the first time we add a custom row, because the settings it generates on its own cannot hold a list.



The names `Font.custom(_:size:relativeTo:)` expects are the PostScript names stored inside the font files, and they rarely match the file names: `Jost-400-Book.ttf` registers a font named `Jost-Book` in a



family named `Jost*`, asterisk included. When in doubt, I keep a temporary `onAppear` that prints `UIFont.familyNames` and `UIFont.fontNames(forFamilyName:)`, and read the names off the console.

10. Create `Font+Jost.swift` in the `Models` folder, replacing the template's `import Foundation` with `import SwiftUI`. A single `jost(_:)` function maps the text styles our article uses to the bundled faces, and `relativeTo:` ties each size to a Dynamic Type style so the typeface follows the user's text size setting. The point sizes are the system defaults for those styles, so Jost sits at the sizes the system font had:

```
import SwiftUI
extension Font {
  static func jost(_ style: Font.TextStyle) -> Font {
    switch style {
    case .title:
      .custom("Jost-Bold", size: 28, relativeTo: .title)
    case .title2:
      .custom("Jost-Bold", size: 22, relativeTo: .title2)
    default:
      .custom("Jost-Book", size: 17, relativeTo: .body)
    }
  }
}
```

11. Set the article in Jost. In `MarkdownView.swift`, the headings switch to the bold face and the whole stack gets the body face, which the list items and paragraphs inherit. The headings also drop their `weight` argument, because `jost(_:)` already returns `Jost-Bold` for both title styles:

```
struct MarkdownView: View {
  // ... the blocks property and the initializer
  var body: some View {
    VStack(alignment: .leading, spacing: 12) {
      ForEach(blocks) { block in
        switch block.kind {
        case .heading(let level):
          Text(block.text)
            .font(.jost(level == 1 ? .title : .title2))
        // ... the listItem and paragraph cases
        }
      }
    }
    .font(.jost(.body))
  }
}
```

In the preview canvas, the article is set in Jost, and the italic, bold, and code spans still render: SwiftUI picks the italic and bold faces of the family for the emphasized runs and switches to a monospaced font for the code spans, so we did not have to map them ourselves.

12. An article set in a typeface is worth exporting. Create `ArticleExporter.swift` in the `Models` folder, replacing the import with `import SwiftUI`. `ImageRenderer` lays out a `MarkdownView` at the width of a Letter page with generous margins, and its `render` method hands us the content size and a drawing closure, which we point at a PDF context:

```
import SwiftUI
enum ArticleExporter {
  static func pdf(for markdown: String) -> URL {
    let page = MarkdownView(markdown: markdown)
      .padding(48)
      .frame(width: 612)
```

```
    let renderer = ImageRenderer(content: page)
    let url = URL.temporaryDirectory.appending(path: "Artic
    renderer.render { size, render in
      var mediaBox = CGRect(origin: .zero, size: size)
      guard let context = CGContext(url as CFURL, mediaBox:
      else {
        return
      }
      context.beginPDFPage(nil)
      render(context)
      context.endPDFPage()
      context.closePDF()
    }
    return url
  }
}
```

13. SwiftUI has no PDF view of its own, so we wrap PDFKit's `PDFView` in a `UIViewRepresentable`, the bridging protocol from the *UIKit Interoperability Basics* section of *Chapter 1*. Create `PDFKitView.swift` in the `Views` folder and add `import PDFKit` below the SwiftUI import. The view is configured once in `makeUIView`, and `updateUIView` loads the document only when the URL changes; the preview renders the article on the spot:

```
import SwiftUI
import PDFKit
struct PDFKitView: UIViewRepresentable {
  let url: URL
  func makeUIView(context: Context) -> PDFView {
    let view = PDFView()
    view.autoScales = true
    return view
  }
  func updateUIView(_ view: PDFView, context: Context) {
    if view.document?.documentURL != url {
      view.document = PDFDocument(url: url)
    }
```

```
  }
}
#Preview {
  PDFKitView(url: ArticleExporter.pdf(for: SampleArticle.ma
}
```

The preview canvas shows the PDF page, scaled to fit the width, with the article in Jost.

14. Create `PDFExportView.swift` in the `Views` folder. The view renders the PDF when it appears, shows a progress view until the file exists, and offers a close button and a `ShareLink` for the file:

```
import SwiftUI
struct PDFExportView: View {
  let markdown: String
  @Environment(\.dismiss) private var dismiss
  @State private var pdfURL: URL?
  var body: some View {
    NavigationStack {
      Group {
        if let pdfURL {
          PDFKitView(url: pdfURL)
        } else {
          ProgressView()
        }
      }
      .navigationTitle("PDF Preview")
      .navigationBarTitleDisplayMode(.inline)
      .toolbar {
        ToolbarItem(placement: .cancellationAction) {
          Button("Close", systemImage: "xmark") {
            dismiss()
          }
        }
        ToolbarItem(placement: .primaryAction) {
          if let pdfURL {
            ShareLink(item: pdfURL)
          }
```

```
        }
      }
      .task {
        pdfURL = ArticleExporter.pdf(for: markdown)
      }
    }
  }
}
#Preview {
  PDFExportView(markdown: SampleArticle.markdown)
}
```

15. Finally, add an export button to `ContentView` and present the sheet from it:

```
struct ContentView: View {
  @State private var isShowingSource = false
  @State private var isExporting = false
  var body: some View {
    NavigationStack {
      ScrollView {
        // ... the Group with the source and the article
      }
      .navigationTitle("MarkdownPress")
      .navigationBarTitleDisplayMode(.inline)
      .toolbar {
        // ... the Source toggle
        ToolbarItem(placement: .primaryAction) {
          Button("Export", systemImage: "doc.richtext") {
            isExporting = true
          }
        }
      }
      .sheet(isPresented: $isExporting) {
        PDFExportView(markdown: SampleArticle.markdown)
      }
    }
  }
}
```

In the preview canvas, tap the document icon: the sheet opens with the article as a PDF page. The share icon even brings up the share sheet here, but its destinations do nothing in the preview canvas, because each one hands the file to another app; that part waits for Device Hub in the next step.

16. Run the app in Device Hub for this final pass, because the share sheet hands the PDF to other apps. Tap the code icon to switch between the rendered article and its source, then tap the document icon to export. In the sheet, tap the share icon, choose **Save to Files** , and tap **Save** to store the file in **On My iPhone** . Leave the app and open the exported file in the Files app to confirm that it preserves the Jost typesetting, because the fonts travel inside the PDF. Back in **MarkdownPress** , choose **Device** > **Accessibility** > **Increase Text Size** in Device Hub a few times, without leaving the app: the headings and the body grow together because of `relativeTo:` . For the complete code, see the book's GitHub repository.

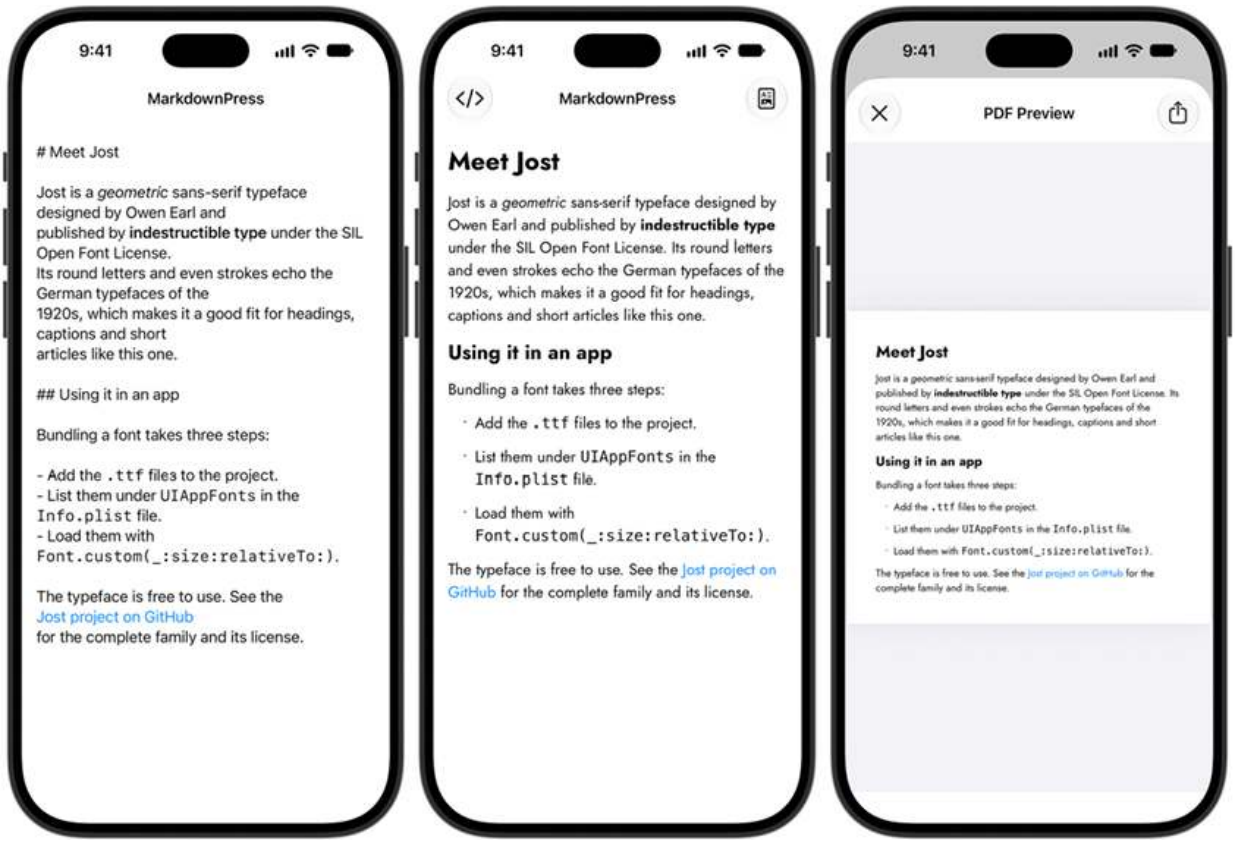


Figure 14.3: The inline-only rendering of step 3, the finished article set in Jost, and the PDF export view

How it works...

When we pass a string literal directly to `Text`, Swift selects the `LocalizedStringKey` initializer. That initializer looks the key up in the app's string tables and parses the inline Markdown in whatever it finds. Passing a `String` selects an initializer that neither localizes nor parses, which explains the different results in steps 2 and 3. `Text(verbatim:)` is the explicit form of that second path: it states in code that the text is meant to be shown as is, and it is the right choice for anything the user typed, where a stray asterisk should stay an asterisk. `Text` recognizes only inline constructs such as emphasis, code, strikethrough, and links, because a `Text` is a single run of styled

text with no notion of block layout; the parsing mode it uses preserves whitespace, which is what kept our wrapped lines broken in step 3.

Our renderer can work block by block because `AttributedString(markdown:)` moves the document's structure out of the characters and into attributes. A heading, a paragraph, or a list item becomes a run whose `presentationIntent` describes it, and the paragraph breaks themselves are gone, so `blocks(from:)` gets exactly one block each time it iterates `runs[\.presentationIntent]`. An intent is a stack of components, innermost first: a bullet's text carries a paragraph, then the list item with its ordinal, then the list, which is why `kind(of:)` scans all of them instead of checking only the first. Inside a block, the emphasis, code spans, and links remain as `inlinePresentationIntent` and `link` attributes, ready for each block's `Text` to render without any styling code of ours. Parsing also turned the soft line breaks we typed into spaces, and a document that fails to parse throws, which the model turns into an empty article rather than a crash. Passing `MarkdownParsingOptions` with `.inlineOnlyPreservingWhitespace` recovers the behavior `Text` showed in step 3; the default full parse is what a renderer wants.

The system registers custom fonts when the app launches. The `UIAppFonts` entry names the bundled files to load before our code runs, and from then on every API that takes a font name, `Font.custom` included, finds them by their PostScript names. Nothing checks those names at compile time, which is why the tip in step 9 matters: a misspelled name falls back to the system font without a warning. The `relativeTo:` argument is what keeps a custom font honest with Dynamic Type, since the size we pass becomes the value for the

default text size and scales with the style we name, the same way the system styles scale; the *Trait Variations and Device Simulations* section of *Chapter 3* shows how to preview those sizes side by side. Step 11 needed no special handling for the emphasized runs because the four files share one family: when `Text` needs a bold or italic variant of `Jost-Book`, the font system finds `Jost-Bold` and `Jost-BookItalic` in that family by their traits. Code spans fall outside the family altogether, and SwiftUI substitutes its monospaced font for them.

Nothing in `ArticleExporter` ever reaches the window. `ImageRenderer` takes a view value, lays it out at its ideal size, and draws it wherever we ask, which is what lets the export come from a `MarkdownView` that is never displayed and never depends on what is scrolled into view. The `render` method is the PDF path. Its closure receives the laid-out size and a drawing function, and everything we do with the `CGContext` between `beginPDFPage` and `endPDFPage` is ours: the article becomes one page as tall as its content, 612 points wide because that is the width of a Letter page, with the `padding` standing in for margins. The output is vector, so the text stays text, the fonts are embedded in the file, and the page looks the same everywhere. One detail is easy to miss: the renderer builds its own environment, so it draws the article in the light appearance even while the app is in dark mode.

`PDFKitView` follows the split we use for every UIKit wrapper: `makeUIView` creates and configures the `PDFView` once, `autoScales` included, and `updateUIView` only reacts to a changed URL, which the `documentURL` check enforces so that SwiftUI's frequent updates do not reload the document. Whether the URL points at our temporary file, a PDF in the bundle, or one downloaded from a server makes no difference to the wrapper. The sheet owns the render: `task` runs it once the sheet is

presented, the progress view covers the moment before the file exists, and `ContentView` only has to present the sheet, the pattern from the *Modal Sheets and Full-Screen Covers* section of *Chapter 11*. A `ShareLink` given a file URL shares the file itself rather than a link to it, which is why the share sheet offers to save it to Files or attach it to a message; we covered `ShareLink` in the *Toolbars and the Li quid Glass Design* section of *Chapter 4*.

See also

- `AttributedString init(markdown:options:baseURL:)` documentation: https://x.swiftuicookbook.com/uzbaj
- `PresentationIntent` documentation: https://x.swiftuicookbook.com/3oqtw
- *Instantiating attributed strings with Markdown syntax* article: https://x.swiftuicookbook.com/3t361
- `Font.custom(_:size:relativeTo:)` documentation: https://x.swiftuicookbook.com/7xymg
- `ImageRenderer` documentation: https://x.swiftuicookbook.com/cnqos
- `PDFView` documentation: https://x.swiftuicookbook.com/akaey

15

On-Device Intelligence in SwiftUI

The machine learning I added to iOS apps before Apple Intelligence followed one pattern: find or train a model for a single narrow task, bundle it with the app, and write the code that turns its output into something a view can show. Each model classified an image, detected text, or scored a sentence, and nothing more. Apple Intelligence reverses that arrangement. The operating system provides general-purpose models, and a SwiftUI app reaches them through two frameworks: Foundation Models, for a language model that runs on the device, and Image Playground, for creating images. The app ships no model at all, so the work that remains is ours: what to ask, how to shape the answer, and how the interface behaves while the model works.

The first three recipes build one cooking assistant, **PantryChef** , and each gives the model more to work with: the ingredients the user types, the app's own pantry data, and finally a photo of the fridge. The app checks `SystemLanguageModel` availability before it offers any controls, with a message for each reason the model can be unavailable. Prompts go through a `LanguageModelSession` , whose `isResponding` flag drives the interface, and a `LanguageModelError` tells us

when the guardrails block a request or the context window runs out. Guided generation then replaces the free-text answer with a `DishSuggestion` struct marked `@Generable`, whose `@Guide` constraints keep the cooking time and the lists in range. We also see how `prewarm()` loads a session ahead of its first request, how `supportsLocale()` reports whether the user's language is supported, and how `streamResponse(to:generating:)` delivers the dish as a `PartiallyGenerated` value while the model writes it.

The second recipe lets the model call our code. A type that conforms to `Tool` reads the user's pantry through `@Generable` arguments, and the session's `Transcript` shows each tool call and its output, so we can check what the model actually received. We also see why forcing a tool call through `GenerationOptions` needs an exit condition, such as a `DynamicProfile` that changes the session while it runs. The SDK ships tools of its own: `SpotlightSearchTool` searches the content an app has indexed, and `OCRTool` and `BarcodeReaderTool` read text and barcodes in images. The model itself is replaceable, too, because the `LanguageModel` protocol lets a provider package plug in a server model.

In the third recipe, the prompt carries an image: we attach the fridge photo with `Attachment`, offer the feature only when the model's `LanguageModelCapabilities` include `.vision`, and compare each request's token `usage` with the model's `contextSize`. Finally, we outline how `PhotosPicker` would supply the user's own photo, and how `PrivateCloudComputeLanguageModel` moves larger requests, with a deeper reasoning level, to Apple's servers.

The chapter ends with **DishArt**, a separate app, because Image Playground brings its own interface and needs no session. We

present the system's image-creation sheet with `imagePlaygroundSheet`, seed it with `ImagePlaygroundConcept` values, and show the button only where the `supportsImagePlayground` environment value allows it. `imagePlaygroundGenerationStyle(_:in:)` opens the sheet in **Any Style**, and `ImagePlaygroundOptions` sets the image size, turns off personalization, and switches the creation strategy so that the user refines the existing artwork instead of starting over. The recipe's closing notes cover concepts extracted from a longer passage, PencilKit drawings, and the style that hands generation to an external provider.

By the end of this chapter, you should be able to decide which parts of a feature belong to a model and which stay in your own code, hand the model your app's data and images and get typed results back, and build interfaces that stay honest about what the device can do while the model is working.

This chapter includes the following recipes:

- Generating Text and Structured Data On Device
- Letting the Model Call Your App's Code
- Multimodal Prompts
- Generating Images with Image Playground

Technical requirements

The code in this chapter is based on Xcode 27.0 and iOS 27.0. You can download and install the latest version of Xcode from the App Store. All the code examples for this chapter can be found on GitHub at https://x.swiftuicookbook.com/ch15.



The recipes in this chapter use Apple Intelligence. A simulated device, whether in Device Hub or behind the preview canvas, has no language model of its own and uses the Mac's. That Mac needs Apple silicon and macOS 27, with Apple Intelligence turned on in **System Settings** . Any Apple silicon Mac qualifies, but on one with 8 GB of memory, the model shares that memory with Xcode and Device Hub, so quit the apps you do not need while you work through the chapter. On an earlier version of macOS, the model reports itself as available, but every request fails. An iPhone that supports Apple Intelligence works as well, and the last recipe needs one, because Image Playground does not run on simulated devices. On either device, Apple Intelligence must support the language and region in use, and the Siri language must match the device language exactly: a Mac set to English (US) with Siri in English (UK) cannot turn on Apple Intelligence. Finally, the preview canvas can reach the model only after the app has run once in Device Hub. If the preview canvas crashes the first time it runs the model, relaunch it and try again.

Generating Text and Structured Data On Device

The first time I shipped a feature built on a language model, most of the work had nothing to do with the feature: an API key to keep out of the app binary, a network client, a privacy review, and a parser for whatever text came back. The Foundation Models framework removes most of that list. The model comes with the operating system and runs on the device, with no key and no network connection, so this recipe can spend its steps on the prompt and on what the app does with the answer.

In this recipe, we build a small cooking assistant: the user types the ingredients they have, and the on-device model suggests a dish. We check `SystemLanguageModel` availability before offering any controls, send the ingredients through a `LanguageModelSession` with `respond(to:)` , and use the session's `isResponding` flag to disable the button while the model works. Then we replace the free-text answer with a `DishSuggestion` struct marked `@Generable` , so the model fills in a name, a cooking time, the ingredients, and the steps as typed properties that we display directly in a form. The next two recipes extend the same app, so the structure we set up here carries through the chapter .

Getting ready

Create a new SwiftUI iOS app named **PantryChef** . The recipe needs a Mac or an iPhone set up for Apple Intelligence, as described in the *Technical requirements* sectio n.

How to do it...

The app has one view and two model files. We start with a plain-text answer, because seeing what the model returns on its own makes it clear what structured generation changes, and then swap it for a typed result without changing how the session is set up. These are the steps:

1. Create `Chef.swift` in a `Models` folder and add `import FoundationModels` below the template's `import Foundation`. `Chef` owns the `LanguageModelSession`, whose instructions describe the model's role and the shape of a good answer, and exposes the latest answer or error to the view, along with the session's `isResponding` flag. Before each request, `suggest(from:)` clears the previous answer and replaces the session with a new one, so no stale answer stays on screen while the model works, and every suggestion starts from an empty history:

```
import Foundation
import FoundationModels
@MainActor
@Observable
final class Chef {
  private(set) var answer: String?
  private(set) var errorMessage: String?
  private var session = LanguageModelSession(instructions:
  private static let instructions = """
    You are a home cooking assistant. Suggest one realisti
    ingredients listed and only common kitchen equipment.
    sentences. Never use markdown, bold text, headings, or
    """
  var isResponding: Bool { session.isResponding }
  func suggest(from ingredients: String) async {
    errorMessage = nil
    answer = nil
    session = LanguageModelSession(instructions: Self.inst
    let prompt = "I have these ingredients: \(ingredients)
```

```
    do {
      let response = try await session.respond(to: prompt)
      answer = response.content
    } catch {
      errorMessage = error.localizedDescription
    }
  }
}
```

2. Open `ContentView.swift` and replace the `ContentView` struct with a form that asks for ingredients. The button starts a `Task` because `suggest(from:)` is asynchronous and a button action is not. The answer and any error appear below the ingredients once they exist:

```
struct ContentView: View {
  @State private var chef = Chef()
  @State private var ingredients = "eggs, spinach, feta, b
  var body: some View {
    NavigationStack {
      Form {
        Section("What do you have?") {
          TextField("Ingredients", text: $ingredients, axi
          Button("Suggest a Dish", systemImage: "fork.knif
            Task { await chef.suggest(from: ingredients) }
          }
        }
        if let answer = chef.answer {
          Section("Suggestion") {
            Text(answer)
          }
        }
        if let message = chef.errorMessage {
          Text(message)
            .foregroundStyle(.red)
        }
      }
      .navigationTitle("PantryChef")
```

```
    }
  }
}
```

3. Run the app in Device Hub first, because the preview canvas finds the model only after the app has run once on the simulated device. Tap **Suggest a Dish** , and after a few seconds a short paragraph names a dish and explains how to make it. The model's output varies from run to run, so the answers quoted in this chapter are representative rather than exact. From here on, the preview canvas in live mode is the quickest place to try each change.
4. This step needs the software keyboard, which the preview canvas does not show, so run the app in Device Hub and turn off **Device** > **Keyboard** > **Simulate Hardware Keyboard** , which is on by default. Edit the ingredients and tap the button again: the keyboard stays up and covers the bottom of the form, which is where the answer appears. Track the text field's focus with a `@FocusState` property, which we used for keyboard navigation in the *Forms* section of *Chapter 4* , and clear it when the button is tapped, so the keyboard goes away as the request starts:

```
struct ContentView: View {
  @State private var chef = Chef()
  @State private var ingredients = "eggs, spinach, feta, b
  @FocusState private var isEditingIngredients: Bool
  var body: some View {
    NavigationStack {
      Form {
        Section("What do you have?") {
          TextField("Ingredients", text: $ingredients, axi
            .focused($isEditingIngredients)
```

```
            Button("Suggest a Dish", systemImage: "fork.knif
              isEditingIngredients = false
              Task { await chef.suggest(from: ingredients) }
            }
          }
          // ... the answer section and the error message
        }
        .navigationTitle("PantryChef")
      }
    }
}
```

5. Nothing tells the user that a request is running, and tapping again would start a second one. Disable the button while the session is responding or the text field is empty, and show a `ProgressView` below it in the meantime:

```
struct ContentView: View {
  // ... the chef, ingredients, and isEditingIngredients p
  var body: some View {
    NavigationStack {
      Form {
        Section("What do you have?") {
          TextField("Ingredients", text: $ingredients, axi
            .focused($isEditingIngredients)
          Button("Suggest a Dish", systemImage: "fork.knif
            isEditingIngredients = false
            Task { await chef.suggest(from: ingredients) }
          }
          .disabled(chef.isResponding || ingredients.isEmp
          if chef.isResponding {
            ProgressView("Thinking...")
          }
        }
        // ... the answer section and the error message
      }
      .navigationTitle("PantryChef")
    }
```

```
  }
}
```

6. A device that cannot run the model should not offer a button that always fails. Add `import FoundationModels` to `ContentView.swift`, keep a reference to `SystemLanguageModel.default`, and wrap the form in a `switch` over its `availability`, so the form appears only when the model is `.available`. Anything else gets a `ContentUnavailableView`, the system's standard empty-state view. A `switch` cannot take a modifier, so we place it in a `Group` and move the navigation title to the group, where it applies to whichever view the `switch` shows:

```
import SwiftUI
import FoundationModels
struct ContentView: View {
  @State private var chef = Chef()
  @State private var ingredients = "eggs, spinach, feta, b
  @FocusState private var isEditingIngredients: Bool
  private let model = SystemLanguageModel.default
  var body: some View {
    NavigationStack {
      Group {
        switch model.availability {
        case .available:
          Form {
            // ... the ingredients section, the answer, an
          }
        case .unavailable:
          ContentUnavailableView(
            "Apple Intelligence Unavailable",
            systemImage: "exclamationmark.triangle"
          )
        }
      }
      .navigationTitle("PantryChef")
```

```
    }
  }
}
```

7. The `.unavailable` case carries a reason, so we can explain what prevents generation. An unsupported device cannot run the model; Apple Intelligence may need to be enabled in **Settings** , or the model may still be downloading. Add a case for each above the general one. The general `.unavailable` case stays last, so the `switch` still covers any reason a later release adds:

```
struct ContentView: View {
  // ... the chef, ingredients, isEditingIngredients, and
  var body: some View {
    NavigationStack {
      Group {
        switch model.availability {
        case .available:
          // ... the form
        case .unavailable(.deviceNotEligible):
          ContentUnavailableView(
            "Device Not Supported",
            systemImage: "iphone.slash",
            description: Text("This device doesn't support
          )
        case .unavailable(.appleIntelligenceNotEnabled):
          ContentUnavailableView(
            "Apple Intelligence Is Off",
            systemImage: "apple.intelligence",
            description: Text("Turn it on in Settings to g
          )
        case .unavailable(.modelNotReady):
          ContentUnavailableView(
            "Model Not Ready",
            systemImage: "arrow.down.circle",
            description: Text("The model is downloading. T
          )
        case .unavailable:
```

```
            // ... the Apple Intelligence Unavailable view
          }
        }
        .navigationTitle("PantryChef")
      }
    }
  }
```

8. The answer reads well, but the app cannot do anything with it: the name, the time, and the ingredients are buried in a paragraph. Create `DishSuggestion.swift` in the `Models` folder and replace the template's `import Foundation` with `import FoundationModels`. The `@Generable` macro lets the model produce this type directly, and each `@Guide` tells the model what a property means and, where it helps, constrains it: the time stays between 5 and 120 minutes, and the ingredient and step lists stay short:

```
import FoundationModels
@Generable(
  description: "A dish the user can cook with the ingredie
)
struct DishSuggestion {
  @Guide(description: "A short, appetizing name for the di
  var name: String
  @Guide(description: "Total cooking time in minutes", .ra
  var minutes: Int
  @Guide(
    description: "Ingredients with amounts, for example 2
    .count(3...8)
  )
  var ingredients: [String]
  @Guide(
    description: "Cooking steps in order, one sentence eac
    .maximumCount(6)
  )
```

```
  var steps: [String]
}
```

9. In `Chef.swift`, add a `suggestion` property, clear it instead of `answer` at the start of `suggest(from:)`, and ask the session for a `DishSuggestion` instead of a string by passing the type as the `generating` argument. Keep the `answer` property for now so `ContentView` still compiles; we delete it in step 11:

```
final class Chef {
  private(set) var answer: String?
  private(set) var suggestion: DishSuggestion?
  private(set) var errorMessage: String?
  // ... the session and instructions properties, and isRe
  func suggest(from ingredients: String) async {
    errorMessage = nil
    suggestion = nil
    session = LanguageModelSession(instructions: Self.inst
    let prompt = "I have these ingredients: \(ingredients)
    do {
      let response = try await session.respond(
        to: prompt, generating: DishSuggestion.self
      )
      suggestion = response.content
    } catch {
      errorMessage = error.localizedDescription
    }
  }
}
```

10. Back in `ContentView.swift`, replace the answer section with a section for the dish. The dish name becomes the section header, and the rows display the cooking time and ingredients directly from the struct's properties, with no parsing in between:

```
struct ContentView: View {
  // ... the chef, ingredients, isEditingIngredients, and
  var body: some View {
    NavigationStack {
      Group {
        switch model.availability {
        case .available:
          Form {
            // ... the ingredients section
            if let dish = chef.suggestion {
              Section(dish.name) {
                LabeledContent("Ready in", value: "\(dish.
                ForEach(dish.ingredients, id: \.self) { Te
              }
            }
            // ... the error message
          }
        // ... the four unavailable cases
        }
      }
      .navigationTitle("PantryChef")
    }
  }
}
```

11. Open `Chef.swift` and delete the `answer` property, which nothing reads anymore.
12. In `ContentView.swift`, add a section for the steps after the dish section. The model returns them as a plain array, so the view numbers them itself with `enumerated()`:

```
struct ContentView: View {
  // ... the chef, ingredients, isEditingIngredients, and
  var body: some View {
    NavigationStack {
      Group {
        switch model.availability {
```

```
          case .available:
            Form {
              // ... the ingredients section
              if let dish = chef.suggestion {
                // ... the dish name section with the time a
                Section("Steps") {
                  ForEach(dish.steps.enumerated(), id: \.ele
                    Text("\(index + 1). \(step)")
                  }
                }
              }
              // ... the error message
            }
          // ... the four unavailable cases
          }
        }
        .navigationTitle("PantryChef")
      }
    }
}
```

13. A blocked request deserves a clearer message than the general one, because changing the ingredients may be all it takes. In `Chef.swift`, catch `LanguageModelError.guardrailViolation` ahead of the general `catch`:

```
final class Chef {
  // ... the suggestion, errorMessage, session, instructio
  func suggest(from ingredients: String) async {
    // ... clearing the old results, the new session, and
    do {
      // ... the respond call and storing the suggestion
    } catch LanguageModelError.guardrailViolation {
      errorMessage = "That request was blocked by the safe
    } catch {
      errorMessage = error.localizedDescription
    }
  }
}
```

14. With the preview canvas in live mode, try a few different ingredient lists: a handful of staples, a single vegetable, or a long list. Each tap starts from a fresh session, so no suggestion is influenced by the one before it, and the progress view shows only while the model is working. For the complete code, see the book's GitHub repository.



Figure 15.1: The free-text answer from step 3 and the finished stru ctured suggestion

How it works...

Everything in **PantryChef** starts from `SystemLanguageModel.default`, the model that Apple Intelligence installs with the operating system, and the prompts we send it never leave the device. We read `availability` directly in `body`, and that is enough to keep the view current: the model is `Observable`, so when the user turns Apple Intelligence on or the download finishes, SwiftUI evaluates the view again and the `switch` moves to the form. The session's `isResponding` is observable in the same way, which is why the button disables itself and the progress view appears without any extra state in the view.

`Chef` is marked `@MainActor` because the view reads it on the main actor; Xcode 27 isolates new projects to the main actor by default, and the explicit attribute keeps the class correct in a project that does not. The *Async/await with Remote APIs* section of *`Chapter 8`* explains why a model that drives views belongs on the main actor.

Each dish suggestion starts a separate conversation with the model, so we create a new `LanguageModelSession` for every request. A session's instructions frame every prompt, and its transcript records each prompt and response and goes back to the model with the next request, which is what makes follow-up questions work in a chat. Our suggestions do not build on each other, and the on-device model reads a limited number of tokens at a time, called its context window, which counts the instructions, the history, and the answer. `SystemLanguageModel.default.contextSize` returns the size of that window, and when we printed it on iOS 27, it was 8,192 tokens, both in Device Hub and on an iPhone. To see how a single shared session fills it, we tested a version of `Chef` that kept one session for every

request: after about a dozen suggestions, requests grew slower as the history grew, and then one failed with `LanguageModelError.contextSizeExceeded`. The *Multimodal Prompts* section later prints the context size together with the tokens that each request uses. The rule of thumb is: a session is one conversation, and independent requests get their own.

A view that keeps one session for a whole conversation is where `prewarm()` pays off, because it loads the session's resources before the first request. Our code replaces the session at the start of every request, and when we measured, prewarming one session did not speed up the next, so **PantryChef** never calls `prewarm()`.

Structured generation is where the framework does the most for us. The `@Generable` macro derives a schema from `DishSuggestion` at compile time, and at run time the session adds it to the prompt and constrains the model's output to match it while the output is being generated. What comes back is always a valid `DishSuggestion`, never text we parse and hope is well formed. The `@Guide` descriptions are prompts in their own right: the example in the ingredients guide is there because, in our first tests, the model listed bare ingredient names until the description showed it what an amount looks like. The constraints are enforced rather than suggested, so `.range(5...120)`, `.count(3...8)`, and `.maximumCount(6)` hold for every response. The model also generates the properties in the order we declare them, which is why `name` comes first and `steps` last.

Failures arrive as `LanguageModelError`, whose cases include a guardrail violation, a refusal, an exceeded context window, rate limiting, and an unsupported language or locale. It replaces

`LanguageModelSession.GenerationError`, which is deprecated in iOS 27, so examples written for iOS 26 that catch the session's error type need updating. We handle `.guardrailViolation` separately to explain that the request was blocked; other failures fall through to the general `catch` and show the framewo rk's own description.

There's more...

A suggestion takes a few seconds to arrive, and the user watches a progress view the whole time. `streamResponse(to:generating:)` delivers the result while it is being generated instead. Each element of the stream is a snapshot whose `content` is a `DishSuggestion.PartiallyGenerated`, a version of our struct that the macro generates with every property optional. In `Chef`, with a `draft` property of that type, the call replaces `respond` like this:

```
let stream = session.streamResponse(
  to: prompt, generating: DishSuggestion.self
)
for try await snapshot in stream {
  draft = snapshot.content
}
```

The properties fill in declaration order: the name arrives first, a few characters at a time, then the minutes, then the ingredients and the steps one element at a time. That makes the order of a `@Generable` type a design decision, because it is the order in which the user watches the result assemble.

The model also depends on the language. Apple Intelligence supports a set of languages, and

`SystemLanguageModel.default.supportsLocale()` reports whether the current locale is one of them, so an app can explain the limitation up front instead of failing with `LanguageModelError.unsupportedLanguageOrLocale` on the first request.

See also

- `SystemLanguageModel` documentation: https://x.swiftuicookbook.com/pw9n1
- `LanguageModelSession` documentation: https://x.swiftuicookbook.com/flufh
- `Generable` documentation: https://x.swiftuicookbook.com/bu8ll
- *Generating Swift data structures with guided generation* article: https://x.swiftuicookbook.com/15v5b
- *Managing the context window* article: https://x.swiftuicookbook.com/yhzjr
- What's new in the Foundation Models framework (WWDC 2026): https://x.swif tuicookbook.com/eec4x

Letting the Model Call Your App's Code

The model cannot see our pantry directly. In the previous recipe, it relied on the ingredients typed into the prompt and could suggest extras we did not have. Tool calling gives it a way to look up what is available. We describe a piece of our own code to the model, and

when the model needs what that code knows, it asks the framework to run it and reads the result before it answers.

In this recipe, we give our cooking assistant a pantry: a list of what is in the fridge and the cupboard that the user can edit, and a tool that reads it. We implement the `Tool` protocol and define a `@Generable` `Arguments` type so the model can choose a shelf. We register the tool with `LanguageModelSession(tools:instructions:)`, then display the call and its result from the session's `Transcript` so we can check which pantry da ta the model received.

Getting ready

This recipe continues the app from the *Generating Text and Structured Data On Device* section. We have already provided that finished app as a starter project, so download or clone the code for this chapter from GitHub as explained in the *Technical requirements* section, and open the **PantryChef** project located in the **Starter** folder of this recipe. If you completed the previous recipe, you can continue with yo ur own project instead.

How to do it...

We build the pantry first, so there is real data for the model to read, then the tool that reads it, and last the part of the interface that shows what the model did. These are the steps:

1. Create `Pantry.swift` in the `Models` folder. `PantryItem` keeps the quantity as text, such as `200 g`, because the model reads it as text anyway. The `Pantry` class holds each shelf as an array, and

`items(on:)` returns one shelf or both by name, which is the lookup the tool will use. `Pantry` is marked `@MainActor` because the list that edits it runs on the main actor. The full sample data is in the book's GitHub repository:

```
import Foundation
struct PantryItem: Identifiable, Equatable, Sendable {
  let id = UUID()
  var name: String
  var quantity: String
}
@MainActor
@Observable
final class Pantry {
  var fridge = [
    PantryItem(name: "Eggs", quantity: "6"),
    // ... the remaining four fridge items
  ]
  var cupboard = [
    PantryItem(name: "Bread", quantity: "1 loaf"),
    // ... the remaining four cupboard items
  ]
  func items(on shelf: String) -> [PantryItem] {
    switch shelf {
    case "Fridge": fridge
    case "Cupboard": cupboard
    default: fridge + cupboard
    }
  }
}
```

2. Create `PantryView.swift` in a `Views` folder. A `List` shows each shelf in its own section, and `.onDelete` lets the user swipe an item away. The view holds the `Pantry` as a plain reference, because `@Observable` tracks the arrays it reads without a property

wrapper, as we saw in the *Observation Framework* section of *Chapter 5*:

```
import SwiftUI
struct PantryView: View {
  let pantry: Pantry
  var body: some View {
    List {
      Section("Fridge") {
        ForEach(pantry.fridge) { item in
          LabeledContent(item.name, value: item.quantity)
        }
        .onDelete { pantry.fridge.remove(atOffsets: $0) }
      }
      Section("Cupboard") {
        ForEach(pantry.cupboard) { item in
          LabeledContent(item.name, value: item.quantity)
        }
        .onDelete { pantry.cupboard.remove(atOffsets: $0)
      }
    }
    .navigationTitle("Pantry")
  }
}
#Preview {
  NavigationStack {
    PantryView(pantry: Pantry())
  }
}
```

3. Open `Chef.swift` and give it the pantry. `Chef` owns a single `Pantry` instance, so the tool we write in step 5 reads the same pantry that the list edits:

```
final class Chef {
  let pantry = Pantry()
```

```
  // ... the existing properties and suggest(from:)
}
```

4. In `ContentView.swift`, add a toolbar button that opens the pantry. A `NavigationLink` pushes `PantryView` onto the navigation stack, and in the toolbar its label shows only the refrigerator symbol:

```
struct ContentView: View {
  // ... the chef, ingredients, isEditingIngredients, and
  var body: some View {
    NavigationStack {
      Group {
        switch model.availability {
        case .available:
          Form {
            // ... the ingredients section, the dish, and
          }
          .toolbar {
            NavigationLink {
              PantryView(pantry: chef.pantry)
            } label: {
              Label("Pantry", systemImage: "refrigerator")
            }
          }
        // ... the four unavailable cases
        }
      }
      .navigationTitle("PantryChef")
    }
  }
}
```

In the preview canvas, tap the refrigerator icon to open the pantry, and swipe a row to delete it. If the preview canvas cannot reach the model, run the app once in Device Hub, as explained in the *Technical requirements* section.

5. Create `PantryTool.swift` in the `Models` folder and replace the template's `import Foundation` with `import FoundationModels`. `PantryTool` conforms to `Tool`: the model reads its `name` and `description` to decide when to call it, and `Arguments` describes what it may pass in. Marking `Arguments` as `@Generable` means the model generates them the same way it generates a `DishSuggestion`, and the `.anyOf` guide limits `shelf` to three values. The tool returns the matching items as plain text:

```
import FoundationModels
struct PantryTool: Tool {
  let name = "checkPantry"
  let description = "Lists the food the user has at home,
  let pantry: Pantry
  @Generable
  struct Arguments {
    @Guide(description: "Where to look", .anyOf(["Fridge",
    var shelf: String
  }
  func call(arguments: Arguments) async throws -> String {
    let items = await pantry.items(on: arguments.shelf)
    guard !items.isEmpty else { return "Nothing on that sh
    return items.map { "\($0.name): \($0.quantity)" }.join
  }
}
```

6. Back in `Chef.swift`, hand the tool to the session. Create each session with `LanguageModelSession(tools:instructions:)`, rewrite the instructions so they tell the model to call `checkPantry` before it suggests a dish, and turn the prompt into a request. The ingredients now come from the pantry, so the text the user types says what they are in the mood for:

```
final class Chef {
  // ... the pantry, suggestion, errorMessage, and session
  private static let instructions = """
    You are a home cooking assistant. Call checkPantry to
    has, then suggest one realistic dish that uses only th
    common kitchen equipment. Answer in a few plain senter
    markdown, bold text, headings, or lists.
    """
  var isResponding: Bool { session.isResponding }
  func suggest(from request: String) async {
    errorMessage = nil
    suggestion = nil
    session = LanguageModelSession(
      tools: [PantryTool(pantry: pantry)], instructions: S
    )
    let prompt = "\(request). What can I cook with the foo
    // ... the do-catch that asks for a DishSuggestion
  }
}
```

7. In `ContentView.swift`, turn the ingredients field into a request field. Rename the two properties, and update the section title, the text field, and the button to match:

```
struct ContentView: View {
  @State private var chef = Chef()
  @State private var request = "Something quick for dinner
  @FocusState private var isEditingRequest: Bool
  private let model = SystemLanguageModel.default
  var body: some View {
    NavigationStack {
      Group {
        switch model.availability {
        case .available:
          Form {
            Section("What are you in the mood for?") {
              TextField("Request", text: $request, axis: .
                .focused($isEditingRequest)
```

```
                Button("Suggest a Dish", systemImage: "fork.
                  isEditingRequest = false
                  Task { await chef.suggest(from: request) }
                }
                .disabled(chef.isResponding || request.isEmp
                // ... the progress view
              }
              // ... the dish and the error message
            }
            // ... the toolbar
          // ... the four unavailable cases
          }
        }
        .navigationTitle("PantryChef")
      }
    }
}
```

8. With the preview canvas in live mode, tap **Suggest a Dish** . The suggestion alone does not show whether the model checked the pantry or which ingredients it found. Let's display the relevant transcript entries so we can inspect the call and its result.
9. Create `ToolActivity.swift` in the `Models` folder, again replacing `import Foundation` with `import FoundationModels` . Each value is one row of the activity list, built from one of two kinds of transcript entry. A `Transcript.ToolCall` holds the tool's name and the arguments the model generated, which `jsonString` prints as JSON, and a `Transcript.ToolOutput` holds what the tool returned as segments of text. A call and its output share the same `id` , so the prefixes keep the rows distinct:

```
import FoundationModels
struct ToolActivity: Identifiable {
  let id: String
```

```
  let title: String
  let detail: String
  init(call: Transcript.ToolCall) {
    id = "call-\(call.id)"
    title = "Called \(call.toolName)"
    detail = call.arguments.jsonString
  }
  init(output: Transcript.ToolOutput) {
    id = "output-\(output.id)"
    title = "\(output.toolName) returned"
    detail = output.segments
      .compactMap { segment in
        if case .text(let text) = segment { text.content }
      }
      .joined(separator: "\n")
  }
}
```

10. In `Chef.swift`, add an `activity` property that walks the session's transcript and turns tool calls and tool output into rows, skipping every other kind of entry:

```
final class Chef {
  // ... the pantry, suggestion, errorMessage, session, an
  var isResponding: Bool { session.isResponding }
  var activity: [ToolActivity] {
    session.transcript.flatMap { entry -> [ToolActivity] i
      switch entry {
      case .toolCalls(let calls): calls.map { ToolActivity
      case .toolOutput(let output): [ToolActivity(output:
      default: []
      }
    }
  }
  // ... suggest(from:)
}
```

11. In `ContentView.swift`, show the rows in a section between the request and the dish. The section appears only when the model has called the tool, which also tells us when it did not:

```
struct ContentView: View {
  // ... the chef, request, isEditingRequest, and model pr
  var body: some View {
    NavigationStack {
      Group {
        switch model.availability {
        case .available:
          Form {
            // ... the request section
            if !chef.activity.isEmpty {
              Section("What the model did") {
                ForEach(chef.activity) { item in
                  VStack(alignment: .leading) {
                    Text(item.title)
                    Text(item.detail)
                      .font(.caption.monospaced())
                      .foregroundStyle(.secondary)
                  }
                }
              }
            }
            // ... the dish and the error message
          }
          // ... the toolbar
        // ... the four unavailable cases
        }
      }
      .navigationTitle("PantryChef")
    }
  }
}
```

12. Use the preview canvas in live mode to try the finished app. Tap **Suggest a Dish**, and the activity section lists the call with the

shelf the model chose, then the pantry the tool returned, above the dish. Open the pantry, delete the eggs, go back, and ask again. Check that the tool output no longer includes eggs, then compare it with the suggested dish. If the activity section stays empty, the model skipped the tool call. Then replace the request in the text field with another, such as **A pasta dish** , and compare the answers. For the complete code, see the book's GitHub repository.



Figure 15.2: The pantry, and a suggestion that shows t he tool call the model made

How it works...

When we tap the button, the session sends more than our prompt. It adds each tool's name, description, and argument schema to the instructions, and the model can reply in one of two ways: with the response we asked for, or with a request to call a tool. When it asks for `checkPantry`, the framework generates the arguments under the same guided generation that fills a `DishSuggestion`, so `shelf` can only be `Fridge`, `Cupboard`, or `All`. It then runs `call(arguments:)`, adds the output to the transcript, and lets the model continue from there to write the dish. In our testing, whenever the model called `checkPantry` for the default request, **Something quick for dinner**, it selected `All`, which is the sensible choice for a request that names no shelf. The name reads as a verb and the description fits in one sentence because both are sent with every request and both steer the model's decision.

The `await` in front of `pantry.items(on:)` comes from where the tool runs. A `Tool` is `Sendable` and `call(arguments:)` is `@concurrent`, so the framework calls it off the main actor; the call hops to the main actor, reads the `@MainActor` pantry, and carries back an array of `Sendable` items. `Chef` builds the tool from its own `pantry` for every session, so a row deleted in `PantryView` is gone from the next lookup.

Every session keeps a `Transcript`, the ordered record of its instructions, prompts, tool calls, tool output, and responses. The session is observable, so the activity section updates as transcript entries arrive without extra state. Each suggestion starts a new session, so the section shows only the current request. The transcript

is also where we look when an answer seems wrong in our own apps, because it shows whether the model called a tool and what it passed.

The model decides whether to call a tool. With our instructions, it called `checkPantry` in nine of ten runs in our testing; when it skipped the call, it invented ingredients the pantry does not have, and the activity section stayed empty. Stronger wording caused another problem in the same tests: telling the model to always call the tool first led to repeated calls without an answer. Forcing the call is no better: setting the `toolCallingMode` of `GenerationOptions` to `.required` loops the same way, because the session keeps calling tools until something stops it. Apple's documentation calls for an exit condition: either a tool throws an error, surfaced to the caller as `LanguageModelSession.ToolCallError`, or a dynamic profile switches the mode back to `.allowed` after the first call.

Calling the tool does not make the answer honest either. With both shelves empty, `checkPantry` returned **Nothing on that shelf.** in every run of our testing, and the model still suggested a dish with ingredients the user does not have. A rule that must always hold, such as never suggesting a dish from an empty pantry, belongs in our own code, for example by disabling the button while both shelves are empty.

There's more...

Our tool reads app data, but a tool can run any code we write, and a few ship with the SDK. `SpotlightSearchTool`, from the CoreSpotlight overlay, lets the model search the content an app has indexed with

Spotlight. The Vision overlay provides `OCRTool` and `BarcodeReaderTool` for reading text and barcodes in images; in Xcode 27.0 they are part of the device SDK only, so code that uses them builds for a device but not for the simulator. When a session has to change while it runs, `LanguageModelSession.DynamicProfile` composes instructions, tools, and options that can switch as the work progresses; the WWDC 2026 session in the *See also* section covers it.

The model is replaceable too. `LanguageModelSession(model:tools:instructions:)` accepts any type that conforms to the `LanguageModel` protocol, so a provider package can plug in a server model while our tool, the transcript, and guided generation keep working. A model on someone else's server needs credentials, and those must never ship inside the app: fetch short-lived tokens from your own server, keep them in the Keychain, and use App Attest to show that the requests come from your app.

See also

- `Tool` documentation: https://x.swiftuicookbook.com/y6hjk
- `Transcript` documentation: https://x.swiftuicookbook.com/u6qkf
- *Expanding generation with tool calling* article: https://x.swiftuicookbook.com/jl7is
- `GenerationOptions.ToolCallingMode` documentation: https://x.swiftuicookbook.com/i0ff4
- Build agentic app experiences with the Foundation Models framework (WWDC 2026): https:

//x.swiftuicookbook.com/rx819

Multimodal Prompts

A typed list of ingredients is only as good as the person typing it. Most of us know what is in the fridge only roughly, and a photo of the open door holds more than we would bother to type. With image input, the on-device model can look at the shelves itself: we send the photo along with the question, and the answer comes back as the same structured dish as before.

In this recipe, we give our cooking assistant a second view that shows a photo of a fridge and asks the model what we can cook with the food on its shelves. We attach the photo to a prompt with `Attachment`, check the model's `LanguageModelCapabilities` for image support before offering the button, and print the token usage of each request to the console. The dish comes back as a `DishSuggestion`, so we first move its sections into a view of thei r own that both views can use.

Getting ready

This recipe continues the app from the *Letting the Model Call Your App's Code* section. We have already provided that finished app as a starter project, so download or clone the code for this chapter from GitHub as explained in the *Technical requirements* section, and open the **PantryChef** project located in the **Starter** folder of this recipe. If you completed the previous recipe, you can continue with your own project instead.

How to do it...

We add the photo and extract a shared view for the dish first, so the fridge view can show an answer as soon as its request works. Then we build the image request and the view around it and finish by checking image support and inspecting token usage. These are the steps:

1. Download the `FridgeSample.jpg` file from the `ch15/assets/` folder in the book's GitHub repository, then drag it into the asset catalog (`Assets.xcassets`) in the Xcode Project Navigator. The photo shows a fridge with eggs, milk, butter, feta, tomatoes, spinach, and peppers. Xcode names the image set after the file, which gives us a `.fridgeSample` image resource to use in code.
2. Both views will show a dish the same way, so move the dish sections out of `ContentView` into a view of their own. Create `DishSections.swift` in the `Views` folder. The view takes a `DishSuggestion` and returns two sections, so it fits inside any `Form` ; the preview wraps it in one with a sample dish:

```
import SwiftUI
struct DishSections: View {
  let dish: DishSuggestion
  var body: some View {
    Section(dish.name) {
      LabeledContent("Ready in", value: "\(dish.minutes) m
      ForEach(dish.ingredients, id: \.self) { Text($0) }
    }
    Section("Steps") {
      ForEach(dish.steps.enumerated(), id: \.element) { in
        Text("\(index + 1). \(step)")
      }
    }
  }
```

```
}
#Preview {
  Form {
    DishSections(
      dish: DishSuggestion(
        name: "Spinach and Tomato Frittata",
        minutes: 20,
        ingredients: ["4 eggs", "100 g spinach", "6 cherry
        steps: [
          "Whisk the eggs.",
          "Cook the spinach and tomatoes.",
          "Add the eggs and bake."
        ]
      )
    )
  }
}
```

3. In `ContentView.swift`, replace the two sections inside `if let dish = chef.suggestion` with the new view:

```
struct ContentView: View {
  // ... the chef, request, isEditingRequest, and model pr
  var body: some View {
    NavigationStack {
      Group {
        switch model.availability {
        case .available:
          Form {
            // ... the request section and the activity se
            if let dish = chef.suggestion {
              DishSections(dish: dish)
            }
            // ... the error message
          }
          // ... the toolbar
        // ... the four unavailable cases
        }
      }
      .navigationTitle("PantryChef")
```

```
    }
  }
}
```

4. Create `FridgeChef.swift` in the `Models` folder, and replace the template's `import Foundation` with `import UIKit` and `import FoundationModels`. UIKit provides `UIImage`, and with both frameworks imported, `Attachment` gains an initializer that takes one. `FridgeChef` follows the shape of `Chef`, without the pantry tool, and its instructions limit the dish to the food in the photo. The prompt is a closure that lists its parts in order: the question, then the photo wrapped in an `Attachment`, whose `label(_:)` gives the image a name:

```
import UIKit
import FoundationModels
@MainActor
@Observable
final class FridgeChef {
  private(set) var suggestion: DishSuggestion?
  private(set) var errorMessage: String?
  private var session = LanguageModelSession(
    instructions: FridgeChef.instructions
  )
  private static let instructions = """
    You are a home cooking assistant. Suggest one realisti
    the food you can see in the photo and common kitchen e
    markdown, bold text, headings, or lists.
    """
  var isResponding: Bool { session.isResponding }
  func suggest(from photo: UIImage) async {
    errorMessage = nil
    suggestion = nil
    session = LanguageModelSession(instructions: Self.inst
    do {
      let response = try await session.respond(
```

```
        generating: DishSuggestion.self
      ) {
        "Here is a photo of my fridge. What can I cook ton
        Attachment(photo).label("fridge")
      }
      suggestion = response.content
    } catch LanguageModelError.guardrailViolation {
      errorMessage = "That request was blocked by the safe
    } catch {
      errorMessage = error.localizedDescription
    }
  }
}
```

5. Create `FridgeView.swift` in the `Views` folder. It loads the bundled photo with `UIImage(resource:)`, shows it above a **What Can I Cook?** button, and reuses `DishSections` for the answer:

```
import SwiftUI
struct FridgeView: View {
  @State private var chef = FridgeChef()
  private let photo = UIImage(resource: .fridgeSample)
  var body: some View {
    Form {
      Section {
        Image(uiImage: photo)
          .resizable()
          .scaledToFit()
          .frame(maxHeight: 240)
          .frame(maxWidth: .infinity)
        Button("What Can I Cook?", systemImage: "sparkles"
          Task { await chef.suggest(from: photo) }
        }
        .disabled(chef.isResponding)
        if chef.isResponding {
          ProgressView("Looking at the photo...")
        }
      }
      if let dish = chef.suggestion {
```

```
          DishSections(dish: dish)
        }
        if let message = chef.errorMessage {
          Text(message)
            .foregroundStyle(.red)
        }
      }
      .navigationTitle("Fridge")
    }
  }
  #Preview {
    NavigationStack {
      FridgeView()
    }
  }
```

6. Back in `ContentView.swift`, add a second toolbar button after the pantry one. In the toolbar, its label shows only the photo symbol:

```
struct ContentView: View {
  // ... the chef, request, isEditingRequest, and model pr
  var body: some View {
    NavigationStack {
      Group {
        switch model.availability {
        case .available:
          Form {
            // ... the request and activity sections, the
          }
          .toolbar {
            // ... the pantry link
            NavigationLink {
              FridgeView()
            } label: {
              Label("Fridge Photo", systemImage: "photo")
            }
          }
        // ... the four unavailable cases
```

```
        }
      }
      .navigationTitle("PantryChef")
    }
  }
}
```

In the preview canvas, tap the photo icon, then **What Can I Cook?** . After a few seconds, a suggested dish appears below the photo. If the preview canvas cannot reach the model, run the app once in Device Hub, as explained in the *Technical requirements* section.

7. Not every model accepts images, so offer the button only when this one does. Add `import FoundationModels` to `FridgeView.swift` , keep a reference to `SystemLanguageModel.default` , and show the button only when its capabilities include `.vision` :

```
import SwiftUI
import FoundationModels
struct FridgeView: View {
  @State private var chef = FridgeChef()
  private let photo = UIImage(resource: .fridgeSample)
  private let model = SystemLanguageModel.default
  var body: some View {
    Form {
      Section {
        // ... the photo
        if model.capabilities.contains(.vision) {
          // ... the button and the progress view
        } else {
          Text("The on-device model cannot read images on
            .foregroundStyle(.secondary)
        }
      }
      // ... the dish sections and the error message
    }
```

```
      .navigationTitle("Fridge")
  }
}
```

8. A photo adds to what a request sends to the model, and all of it has to fit in the context window we measured in the first recipe. To see how much room the complete request and its answer take, print the input and output token counts from the response's `usage` in `FridgeChef.swift`, together with the size of the model's context window:

```
final class FridgeChef {
  // ... the suggestion, errorMessage, session, instructio
  func suggest(from photo: UIImage) async {
    errorMessage = nil
    suggestion = nil
    session = LanguageModelSession(instructions: Self.inst
    do {
      // ... the request with the question and the photo
      suggestion = response.content
      print("Input tokens:", response.usage.input.totalTok
      print("Output tokens:", response.usage.output.totall
      print("Context window:", SystemLanguageModel.default
    } catch LanguageModelError.guardrailViolation {
      errorMessage = "That request was blocked by the safe
    } catch {
      errorMessage = error.localizedDescription
    }
  }
}
```

With the preview canvas in live mode, ask for a dish in the fridge view, and show the console in Xcode's debug area with **Canvas** selected at its bottom. Each answer adds three lines to

the console: the input and output token counts of that request, and the number of tokens the model can hold.

9. Still in the preview canvas, tap **What Can I Cook?** a few more times, scroll down to the steps, and keep an eye on the console. The suggested dish may change between requests, but the input count stays the same, because every request starts a new session with the same instructions, question, and photo. For the complete code, see the book's GitHub repository.

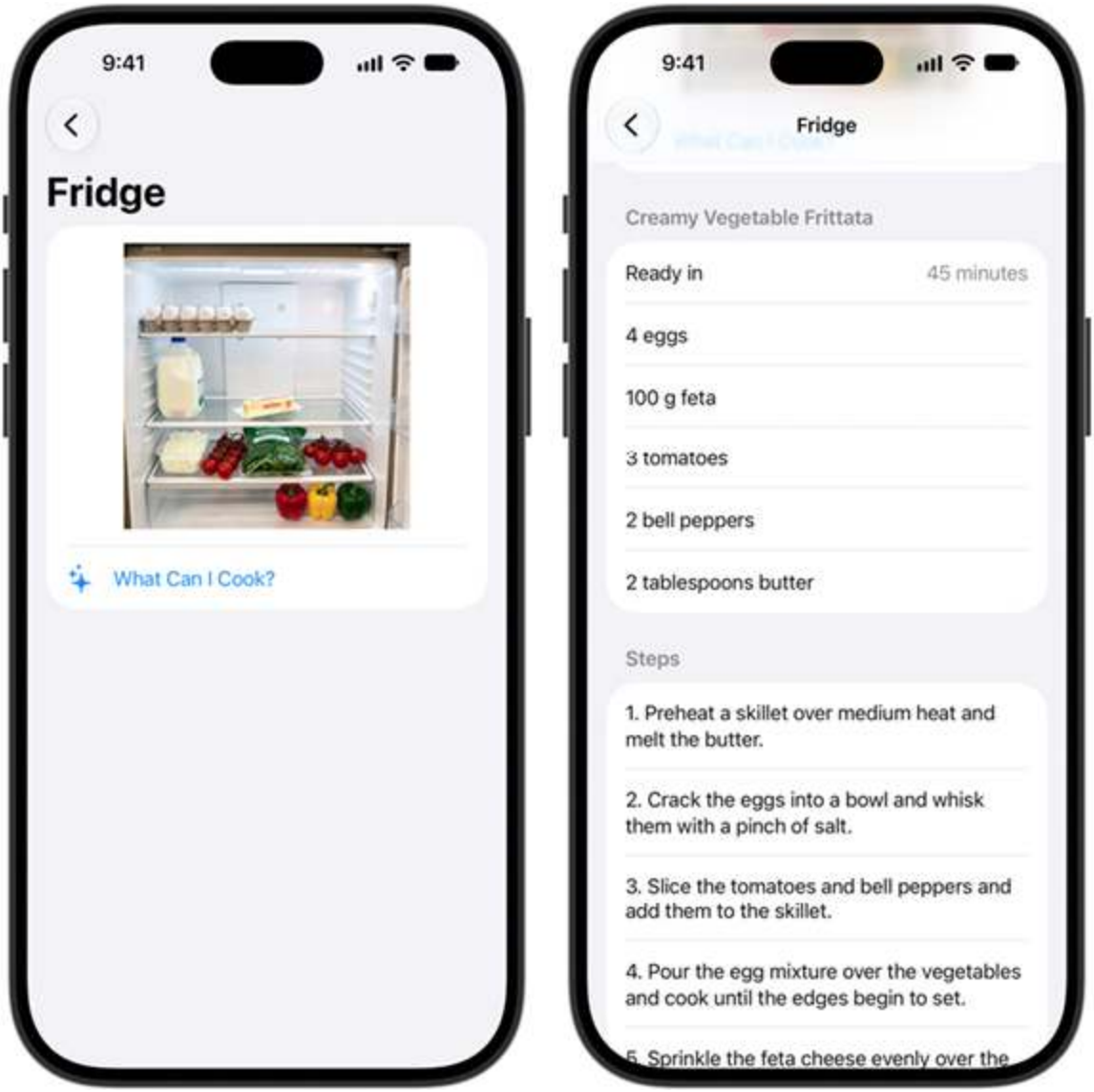


Figure 15.3: The fridge photo, and the dish the model suggests from it

How it works...

The fridge request asks for a `DishSuggestion` just as `Chef` does, but through `respond(generating:)` , whose prompt is a `@PromptBuilder` closure instead of a string, so the question and the photo travel together as one prompt. `Attachment` also accepts a `CGImage` , a `CIImage` , a pixel buffer, or a file URL, and the initializer that takes a `UIImage` is part of an overlay that the compiler loads only when a file imports both UIKit and FoundationModels. With a single photo, `label(_:)` is optional; it names the image so that the text of the prompt, or a tool, can refer to it.

The model can misidentify food in the photo. In our testing, it described food on each shelf, but in a free-text answer it took the crumbled feta for cottage cheese, and one structured answer added a clove of garlic that the fridge does not have. The instruction to use only the food in the photo guides the suggestion, but does not guarantee that every suggested ingredient is actually present. That is worth remembering before an app treats a description of a photo as fact.

The `.vision` check in step 7 reads `SystemLanguageModel.default.capabilities` , which lists what a particular model can do. When we checked, the on-device model reported `.vision` but not `.reasoning` .

The lines we print in `FridgeChef` show the token usage of the complete request and its answer. One run printed the following:

```
Input tokens: 461
Output tokens: 159
Context window: 8192
```

The input count covers the instructions, the question, the photo, and the `DishSuggestion` schema. It stayed at 461 on every run. Against the 8,192 tokens of the context window, which `SystemLanguageModel.default.contextSize` reports on iOS 27, one request with a photo uses a small share. `response.usage` counts a single request, while `session.usage` adds up every request in the session, so it measures the session's total work rather than how m uch of the context window is in use.

There's more...

This recipe uses a bundled photo, but we could let the user choose one with `PhotosPicker`. We load the selected item's image data with `loadTransferable(type:)` and convert it with `UIImage(data:)`. The resulting image can go straight to `suggest(from:)`, so `FridgeChef` needs no changes.

Larger requests can also run on Apple's servers. `LanguageModelSession(model: PrivateCloudComputeLanguageModel())` creates a session backed by the Private Cloud Compute model, which supports `.reasoning`, so `ContextOptions(reasoningLevel: .deep)` can ask it to think longer before it answers. Using that model requires a managed entitlement that Apple grants to developers who meet its eligibility requirements, which is why this recipe stays on device; the developer page in the *See also* sec tion explains how to request access.

See also

- `Attachment` documentation: https://x.swiftuicookbook.com/4m1gq
- `LanguageModelCapabilities` documentation: https://x.swiftuicookbook.com/uvrqu
- `contextSize` documentation: https://x.swiftuicookbook.com/90l12
- `PrivateCloudComputeLanguageModel` documentation: https://x.swiftuicookbook.com/yjj45
- Private Cloud Compute developer page: https://x.swiftuicookbook.com/x01f6

Generating Images with Image Playground

In a food app, pictures add up quickly: each dish needs a photograph, an illustration, or a licensed stock image. Image Playground moves that work to the system, which provides both the interface for creating an image and the model behind it. Our app hands over a few words and receives a finished picture, with no session to manage.

In this recipe, we build an app that creates artwork for a dish from its name. We present the system's image-creation sheet with the `imagePlaygroundSheet` modifier, seed it with text concepts, load the image it returns, and offer the button only where the `supportsImagePlayground` environment value says the device can create images. We then use `imagePlaygroundGenerationStyle(_:in:)` to open the

sheet in **Any Style** , which supports photographic results. With `ImagePlaygroundOptions` , we request an image size and disable personalization. Finally, we pass the artwork back as the source image so the user ca n refine it instead of starting over.

Getting ready

Create a new SwiftUI iOS app named **DishArt** . Unlike the language model in the previous recipes, Image Playground is not available on simulated devices, so the steps that create images need an iPh one that supports Apple Intelligence.

How to do it...

The whole app lives in `ContentView.swift` . We get a created image on screen as early as possible, then add the feedback, the availability check, and the settings that shape the result, trying each change on the iPhone. These are the steps:

1. Open `ContentView.swift` and replace the `ContentView` struct with a form that holds the dish name and a place for the artwork. The artwork is a `UIImage` , because we will load it from the file that Image Playground returns, and until it exists a `ContentUnavailableView` fills its section:

```
struct ContentView: View {
  @State private var dish = "Spinach and feta omelette"
  @State private var artwork: UIImage?
  var body: some View {
    Form {
      Section("Dish") {
        TextField("Dish name", text: $dish)
```

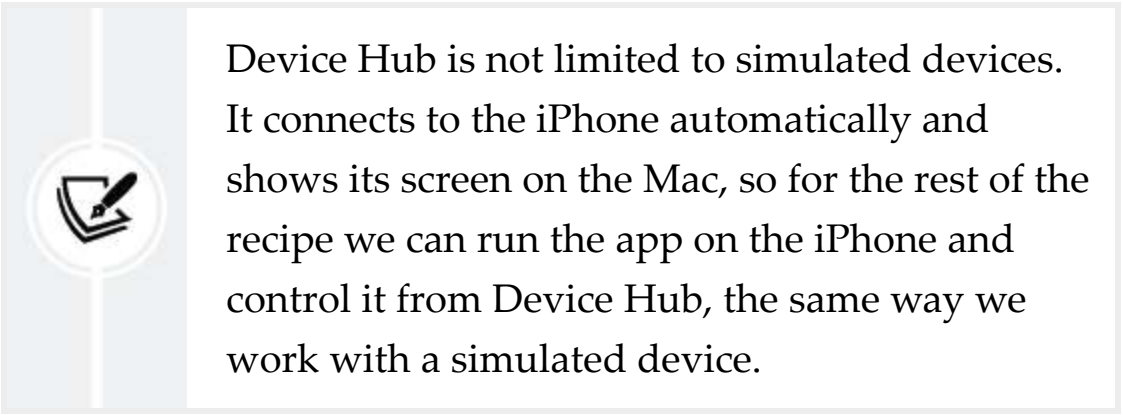
```
        }
        Section("Artwork") {
          if let artwork {
            Image(uiImage: artwork)
              .resizable()
              .scaledToFit()
              .clipShape(.rect(cornerRadius: 12))
          } else {
            ContentUnavailableView(
              "No Artwork Yet",
              systemImage: "photo",
              description: Text("Name a dish and create its
            )
          }
        }
      }
    }
  }
```

2. Add `import ImagePlayground` below `import SwiftUI`, then present the sheet. The `imagePlaygroundSheet` modifier works like the `.sheet(isPresented:)` modifier from the *Modal Sheets and Full-Screen Covers* section of *Chapter 11*: setting a Boolean to `true` presents it, and the sheet sets it back to `false` when it closes. We seed the sheet with the dish name as a `.text` concept. When the user accepts an image, `onCompletion` receives the URL of a temporary file, so we load the image right away:

```
import SwiftUI
import ImagePlayground
struct ContentView: View {
  @State private var dish = "Spinach and feta omelette"
  @State private var artwork: UIImage?
  @State private var isPlaygroundPresented = false
  var body: some View {
    Form {
      // ... the Dish and Artwork sections
```

```
        Section {
          Button("Create Artwork", systemImage: "apple.image
            isPlaygroundPresented = true
          }
          .disabled(dish.isEmpty)
        }
      }
      .imagePlaygroundSheet(
        isPresented: $isPlaygroundPresented,
        concepts: [.text(dish)],
        onCompletion: { url in
          artwork = UIImage(contentsOfFile: url.path())
        }
      )
    }
  }
```

3. Select an iPhone that supports Apple Intelligence as the run destination and run the app. Tap **Create Artwork** . The sheet opens with the dish name as its prompt and creates an image. Tap the checkmark button to accept it, and the artwork appears in the form.



Device Hub is not limited to simulated devices. It connects to the iPhone automatically and shows its screen on the Mac, so for the rest of the recipe we can run the app on the iPhone and control it from Device Hub, the same way we work with a simulated device.

4. The sheet takes a moment to appear, and until it does, nothing in the form acknowledges the tap. Wrap the button in an `HStack` with a `ProgressView` that shows while `isPlaygroundPresented` is `true`

, and disable the button for the same period. Track the text field's focus with `@FocusState` as well, and clear it in the button's action, so the keyboard is out of the way before the sheet opens:

```
struct ContentView: View {
  @State private var dish = "Spinach and feta omelette"
  @State private var artwork: UIImage?
  @State private var isPlaygroundPresented = false
  @FocusState private var isEditingDish
  var body: some View {
    Form {
      Section("Dish") {
        TextField("Dish name", text: $dish)
          .focused($isEditingDish)
      }
      // ... the Artwork section
      Section {
        HStack {
          Button("Create Artwork", systemImage: "apple.ima
            isEditingDish = false
            isPlaygroundPresented = true
          }
          .disabled(dish.isEmpty || isPlaygroundPresented)
          if isPlaygroundPresented {
            Spacer()
            ProgressView()
          }
        }
      }
    }
    // ... the imagePlaygroundSheet modifier
  }
}
```

5. A device that cannot create images should not offer a button that cannot work. Read the `supportsImagePlayground` environment

value and show the button only when it is `true`, with a short message otherwise:

```
struct ContentView: View {
  @Environment(\.supportsImagePlayground) private var supp
  // ... the dish, artwork, isPlaygroundPresented, and isE
  var body: some View {
    Form {
      // ... the Dish and Artwork sections
      Section {
        if supportsImagePlayground {
          HStack {
            // ... the button and the progress view
          }
        } else {
          Text("Image Playground is not available on this
            .foregroundStyle(.secondary)
        }
      }
    }
    // ... the imagePlaygroundSheet modifier
  }
}
```

The preview canvas now shows the message, because simulated devices cannot create images, while the iPhone still shows the button.

6. The dish name alone leaves the scene to the model. Add a second concept that places the dish on a rustic wooden table, and choose how the sheet styles the image. `.any` opens it in **Any Style**, which lets the model choose the look, and the list limits its style picker to that style, **Illustration**, and **Sketch**:

```
struct ContentView: View {
  // ... the five properties
  var body: some View {
    Form {
      // ... the three sections
    }
    .imagePlaygroundSheet(
      isPresented: $isPlaygroundPresented,
      concepts: [.text(dish), .text("plated on a rustic wo
      onCompletion: { url in
        artwork = UIImage(contentsOfFile: url.path())
      }
    )
    .imagePlaygroundGenerationStyle(.any, in: [.any, .illu
  }
}
```

Run the app on the iPhone and create the artwork again. The sheet joins the two concepts into one prompt, **Spinach and feta omelette plated on a rustic wooden table** . In our tests, this prompt produced an omelette that looked like a photograph.

7. Some settings are not part of the prompt at all. Collect them in an `options` property: `sizeSpecification` asks for a square image, at the supported size closest to 1024 by 1024, and turning `personalization` off removes the options for adding people to the image, which a food app does not need. Apply them with `imagePlaygroundOptions(_:)` :

```
struct ContentView: View {
  // ... the five properties
  private var options: ImagePlaygroundOptions {
    var options = ImagePlaygroundOptions()
    options.sizeSpecification = .closest(to: CGSize(width:
    options.personalization = .disabled

    
```

```
    return options
  }
  var body: some View {
    Form {
      // ... the three sections
    }
    // ... the imagePlaygroundSheet and generation style m
    .imagePlaygroundOptions(options)
  }
}
```

8. At the moment, every tap on the button starts over. Pass the current artwork to the sheet as its `sourceImage`, and set the `creationStrategy` to `.editExisting` whenever there is artwork, so the sheet changes the image we already have instead of creating a new one:

```
struct ContentView: View {
  // ... the five properties
  private var options: ImagePlaygroundOptions {
    var options = ImagePlaygroundOptions()
    options.creationStrategy = artwork == nil ? .generateN
    options.sizeSpecification = .closest(to: CGSize(width:
    options.personalization = .disabled
    return options
  }
  var body: some View {
    Form {
      // ... the three sections
    }
    .imagePlaygroundSheet(
      isPresented: $isPlaygroundPresented,
      concepts: [.text(dish), .text("plated on a rustic wo
      sourceImage: artwork.map { Image(uiImage: $0) },
      onCompletion: { url in
        artwork = UIImage(contentsOfFile: url.path())
      }
    )
```

```
    .imagePlaygroundGenerationStyle(.any, in: [.any, .illu
    .imagePlaygroundOptions(options)
  }
}
```

9. Rename the button to match, so it says what the next tap will do:

```
struct ContentView: View {
  // ... the five properties and the options property
  var body: some View {
    Form {
      // ... the Dish and Artwork sections
      Section {
        if supportsImagePlayground {
          HStack {
            Button(
              artwork == nil ? "Create Artwork" : "Edit Ar
              systemImage: "apple.image.playground"
            ) {
              isEditingDish = false
              isPlaygroundPresented = true
            }
            // ... the disabled modifier and the progress
          }
        } else {
          Text("Image Playground is not available on this
            .foregroundStyle(.secondary)
        }
      }
    }
    // ... the imagePlaygroundSheet modifier and the two c
  }
}
```

10. Run the finished app on the iPhone. Create the artwork and accept it, then tap **Edit Artwork** . The sheet opens with our

image as its source; describe a change, such as a different plate, in the **Describe a change** field. The sheet applies it and tries to keep the rest of the image. Try a few other dishes as well. To explore the **Illustration** and **Sketch** styles, open the sheet's style picker using the pencil next to the prompt. For the complete code, see the book's GitHub repository.

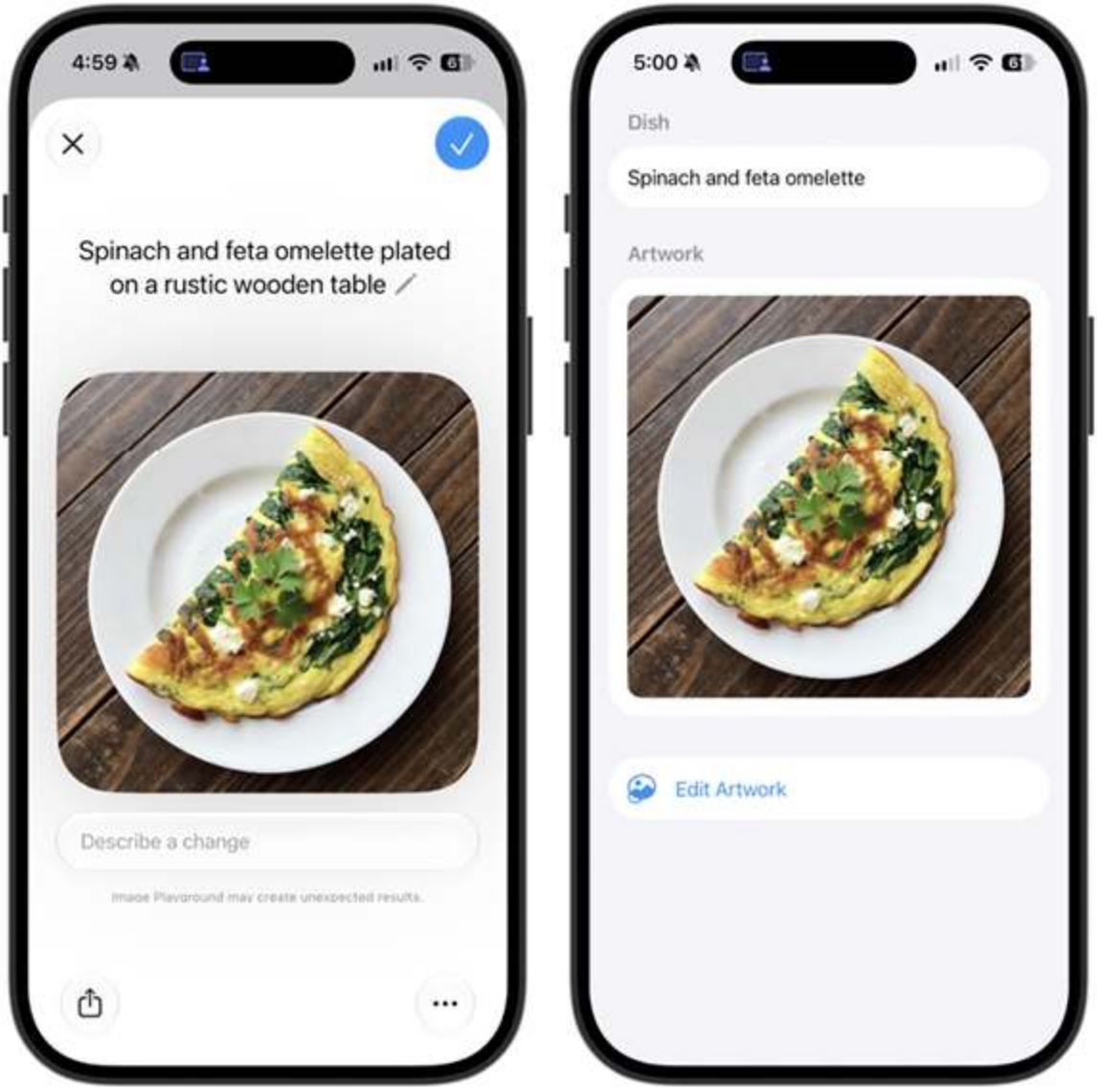


Figure 15.4: The Image Playground sh eet in Any Style, and the finished artwork

How it works…

After we tap the button, the system handles both the editing interface and image generation; our side is a few concepts going in and a file URL coming out. The concepts are only a starting point, because the user can rewrite the prompt or replace the source image before accepting. The file sits at a temporary location in the app's container, which is why we load it inside `onCompletion`; an app that keeps its artwork should move it somewhere permanent. The binding behaves like any sheet binding, so `isPlaygroundPresented` also drives the progress view and the disabled button, and we leave out the optional `onCancellation`, because a dismissed sheet should leave the current artwork alone.

The photograph comes from a new iOS 27 image model that can create images in almost any style. It runs on Private Cloud Compute, Apple's server infrastructure for Apple Intelligence, which also hosts the larger language model mentioned at the end of the *Multimodal Prompts* section. There is no photorealistic style value; `.any` lets the model take the look from the prompt. `supportsImagePlayground` is `false` wherever image creation is unavailable, including every simulated device, because Image Playground cannot borrow the Mac's model the way the language model of the previous recipes does.

The style list from step 6 governs only the picker. `imagePlaygroundGenerationStyle(_:in:)` sets the style the picker opens on and the styles it lists, and a single style locks it, but none of that limits the prompt: in our testing, with **Any Style** left out, describing a watercolor in the prompt still produced one. This modifier and `imagePlaygroundOptions(_:)` write to the environment, so they belong

on the view that presents the sheet or one of its ancestors; the *Reading and Defining Environment Values* section of *Chapter 6* shows how environment values flow down the view hierarchy.

When we tapped **Edit Artwork** in step 10, the sheet applied our change and tried to keep the rest of the image. That is the `.editExisting` strategy: it preserves as much of the source image as possible and falls back to generating a new image when necessary, while `.generateNew` treats the source only as inspiration. Of the other two options, `sizeSpecification` selects the supported resolution and aspect ratio closest to our request. Apple does not list the supported sizes, and the closest one can be larger than the request: in our testing, asking for 1024 by 1024 returned a square image of 2048 by 2048 pixels, so the request settles the aspect ratio more reliably than the exact size. Disabling `personalization` removes name detection in the prompt along with the people picker.

There's more...

Short text is not the only seed. The `ImagePlaygroundConcept.extracted(from:title:)` method pulls the key ideas out of a longer passage, such as a **PantryChef** suggestion, and `.drawing(_:)` adds a PencilKit drawing as a visual suggestion. Adding `.externalProvider` to the style list offers the third-party provider the user has set up in **Settings** , such as ChatGPT.

See also

- `imagePlaygroundSheet(isPresented:concepts:sourceImage:onCompletion:onCancellation:)` documentation:

https://x.swiftuicookbook.com/i4ts6

- `supportsImagePlayground` documentation: https://x.swiftuicookbook.com/l6qhk
- `imagePlaygroundGenerationStyle(_:in:)` documentation: https://x.swiftuicookbook.com/er0ca
- `ImagePlaygroundOptions` documentation: https://x.swiftuicookbook.com/htxlb
- Create high-quality images using Image Playground (WWDC 2026): https://x.swiftuicookbook.com/yaawr



packtpub.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

A t www.packtpub.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:



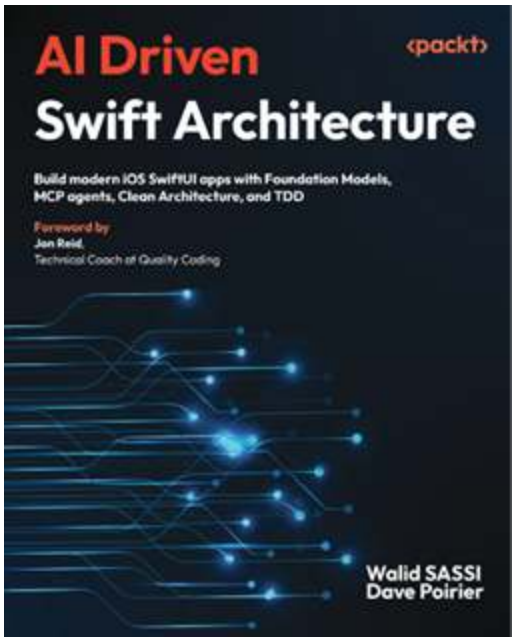
SwiftUI for Beginners

Ahmad Sahar

ISBN: 978-1-80778-557-4

- Build a complete iOS app on iPad using Swift Playground and SwiftUI
- Design adaptive, multi-screen interfaces using SwiftUI components

- Create navigation flows and manage data across multiple screens
- Store persistent app data with SwiftData
- Integrate Core Location and MapKit for location-aware features
- Access the Photo Library for rich media experiences
- Add search functionality and build reusable custom SwiftUI views
- Test and publish your finished app to the App Store via TestFlight



AI Driven Swift Architecture

Walid SASSI, Dave Poirier

ISBN: 978-1-83588-654-0

- Configure Xcode workflows for AI assisted Swift workflows
- Apply Swift concurrency patterns with AI guided reasoning
- Design SwiftUI features using Clean Architecture boundaries

- Learn to use Claude Code to drive TDD, refactors, and code reviews
- Modernize UIKit codebases with safe, incremental migrations
- Integrate Apple Foundation Models for private on device AI
- Build MCP servers in Swift to power feature focused agents
- Write RFCs and ADRs to govern AI adoption in iOS teams

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packt.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* , 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.



https://packt.link/r/1806663775

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

Index

Symbols

3D charts 532 - 540

@AppStorage 228 - 236

@Binding

basics 146 - 157

@SceneStorage 228 - 236

@State

basics 146 - 157

debugging 308 - 320

A

accessibility APIs 211 - 226

ActivityTrends app

creating 512

alerts and dialogs 111

AlertShowcase app, creating 112 - 119

animations 461 - 472

explicit form 469, 471

implicit form 471

AppPreferences app 229

ArtWall app

animations 461 - 472

matched geometry 461 - 472

transitions 461 - 472

async/await

with remote APIs 281 - 289

AsyncImage 296 - 306

AttributedString

rich text 556 - 566

automated UI tests 335 - 345

B

BasicViews app 4

BrewTimer app

keyframe and phase animations 473 - 482

ButtonShowcase app 12 - 20

C

caching 296 - 306

Canvas 452 - 461

CityClimate app

building 494

chart 495 - 503

Codable 268 - 280

Combine

with local data 170 - 178

with remote data 289 - 295

ControlsShowcase app 21

custom drawing 440 - 451

custom layouts 378

SkillCloud app, creating 379 - 388

custom rendering 452 - 461

D

data

sharing with environment, using Observation 182 - 194

deep linking 432

ShopLinks app, creating 432 - 438

drag and drop 51 - 57

device simulations 78 - 90

dismissal and presentation customization 425

LocationExplorer app, creating 425 - 431

E

editable cells 359 - 378

environment

Observation, used for sharing data with 182 - 194

values, reading and defining 199 - 210

essentials, of SwiftUI

basic views and layout 4 - 12

buttons and navigation 12 - 20
pickers 21 - 29
SF Symbols 21 - 29
toggles 21 - 29
UIKit interoperability 29 - 34
ExpenseLedger app 320

F

filtering 46
forms 102
RegistrationForm app, creating 102 - 110
Foundation Models
tool calling 593 - 601
full-screen covers 416 - 425
FunctionLab app
creating 522
function plot 522 - 531

G

glass effects 482 - 490
GlassGallery app
glass and material effects 482 - 490
Gradebook app 360
Grid 350 - 359
GridRow 350 - 359

MedalStandings app, creating 350 - 359

I

Image Playground

images, generating 609 - 617

Instruments 320

performance 321 - 335

K

keyframe animation 473 - 482

L

Liquid Glass 125 - 133 , 405- 415 , 482 - 490

List

using 36 - 45

local data

Combine, using with 170 - 178

local files 255 - 266

M

MarkdownPress app 567 - 578

matched geometry 461 - 472

material effects 482 - 490

menus and disclosure groups 120

MenuExplorer app, creating 121 - 125

MiniBrowser app 542

mock data

previews with 91 - 97

MockDataPreviews app 91

modal sheets 416

ContactManager app, creating 416 - 425

modifiers 11

multimodal prompts 601 - 609

MusicPlayer app

creating 183 - 194

rebuilding, with ObservableObject 195 - 199

MusicSearch app

async/await, with remote APIs 281 - 289

Combine, with remote data 289 - 295

N

NavigationPath 392

ParkGuide app, creating 393 - 405

NavigationStack 392 -405

O

ObservableObject

used, for creating MusicPlayer app 195 - 199

used, for sharing data with environment 195 - 199

observation framework 158 - 169

used, for sharing data with environment 182 - 194

P

PantryChef app 583 - 593 , 596 - 599 , 602 - 60 8

paths 440 - 451

phase animation 473 - 482

PhotoBrowser app

AsyncImage and caching 296 - 306

pickers 21 - 29

PreviewExplorer app 72

previews

configuring, with #Preview 72 - 78

with mock data 91 - 97

ProductCatalog app

creating 268

R

RecipeBrowser app

accessibility APIs 211 - 226

remote APIs

async/await with 281 - 289

remote data

Combine, using with 289 - 295

reusable views and modifiers 133

ViewComposer app, creating 134 - 140

RichNotes app 557

rich text

RichNotes app, creating 557 - 566

with AttributedString 556 - 566

S

ScoreTracker app 237

scroll views

using 57 - 63

search capabilities

adding 46 - 50

Section

using 36 - 45

semantic colors 12

SettingsDashboard app

environment values, reading and defining 199 - 210

SF Symbols 21 - 29

shapes 440 - 451

SimpleJournal app

creating 256

local files 255 - 266

single-token filtering 51

SkyChart app

Canvas and custom rendering 452 - 461

structured data

generating 583 - 592

Swift Charts 494 - 503

customizing 503 - 511

interactivity 511 - 521

SwiftData 246 - 255

SwiftUI

WebView 542 - 556

T

TabView 405 - 415

MediaLibrary app, creating 406 - 415

TaskManager app 247

TerrainExplorer app

3D charts 532 - 540

creating 532

text

generating 583 - 592

TextPreviewer app 30

toggles 21 - 29

toolbars and Liquid Glass design 125

ToolbarShowcase app, creating 126 - 133

TrailStats app

creating 441

custom drawing 440 - 451

custom shapes 440 - 451

paths 440 - 451

TraitExplorer 78

trait variations 78 - 90

transitions 461 - 472

trip-planning board

building 64 - 70

U

UIKit app

components 29 - 34

previews, adding 97 - 100

URL handling 432 - 438

UserDefaults 237 - 246

V

vectorized plot 522 - 531

views

debugging 308 - 320

W

WebView

in SwiftUI 542 - 556

MiniBrowser app, creating 542 - 556