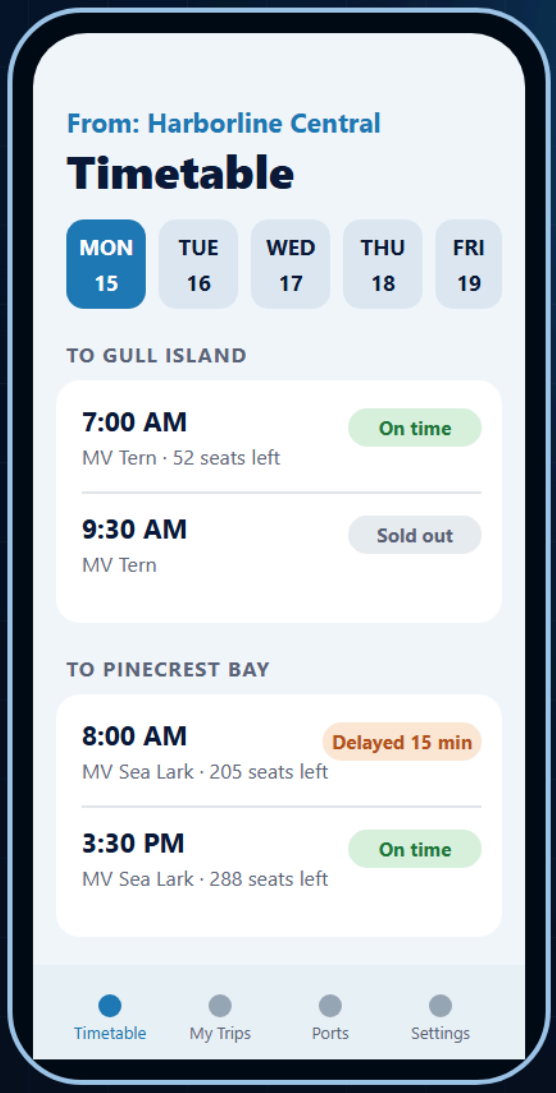
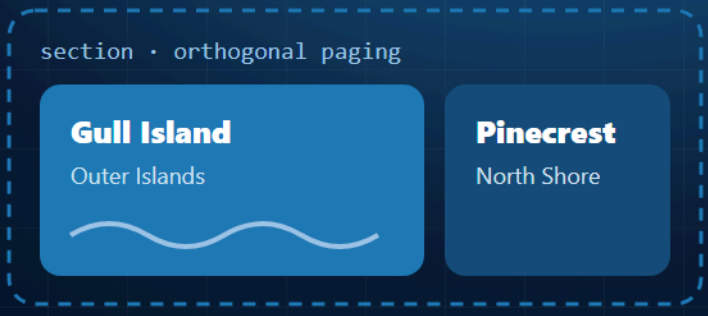
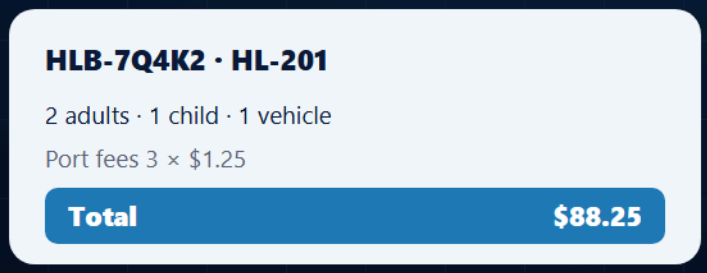



```
var snapshot = Snapshot()
snapshot.appendSections(
  [.route("HLC-PNC")])
snapshot.reconfigureItems(
  ["HL-201"])
dataSource.apply(snapshot,
  animatingDifferences: true)
```





A PROFESSIONAL, HANDS-ON GUIDE

UIKit in Practice

Building Professional iOS Apps with UIKit, Auto Layout, and Modern Collection Views

View Controllers · Auto Layout · Diffable Data Sources
Compositional Layout · Coordinators · Swift 6 · Testing

LAYOUT · LISTS · NAVIGATION · ACCESSIBILITY · INTEROP

THENEXBOOKS

UIKit in Practice

Building Professional iOS Apps with UIKit, Auto Layout, and Modern Collection Views
From view controller lifecycle to diffable data sources, coordinators, accessibility, SwiftUI interop, and Swift 6 — built step by step

First Edition

THENEXBOOKS

UIKit in Practice
Building Professional iOS Apps with UIKit, Auto Layout, and Modern Collection Views

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. iOS, Xcode, Swift, and UIKit change with every release; APIs, build settings, and default behaviors can differ between SDK versions. Always consult Apple's current documentation for the versions you use.

Apple, iPhone, iPad, iOS, iPadOS, Xcode, Swift, SwiftUI, UIKit, Instruments, VoiceOver, SF Symbols, and App Store are trademarks of Apple Inc., registered in the U.S. and other countries and regions. Other product names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement. This book is independent and is not affiliated with, sponsored by, or endorsed by Apple Inc. The screen sketches in this book are original illustrations of the fictional sample app.

Harborline Ferries, the ferry operator used as the running example in this book, is fictional, as are its ports, routes, vessels, timetables, fares, passengers, bookings, and web addresses. Example domains such as `api.harborline.example` and `example.com` are reserved and do not resolve.

Preface

SwiftUI gets the headlines, but a very large share of the iOS code that teams maintain, extend, and ship every week is written in UIKit. Banking apps, airline apps, retail apps, internal enterprise tools — many of them began years before SwiftUI existed, and they keep growing. When you join one of those teams, you are expected to open a view controller with eight hundred lines, understand why a table view crashes on a batch update, fix a constraint conflict that only appears on an iPad in landscape, and add a new screen that looks and behaves like the rest of the app.

UIKit is also still the best tool for some jobs in brand-new apps: complex, high-performance lists; precise layout and text handling; custom controls; and fine-grained control over transitions and gestures. The frameworks are not rivals. Modern iOS apps mix them, and professional iOS developers move between them with confidence.

This book teaches UIKit the way it is used in real, professional codebases today. We cover the legacy patterns you will read in existing code — storyboards, classic table view data sources, trait-change callbacks — and then the modern APIs you should write in new code: diffable data sources, compositional layout, cell registrations and content configurations, sheet detents, button configurations, and Swift 6 concurrency. Throughout, we explain *why* UIKit works the way it does, so you can reason about problems instead of copying fixes.

Who This Book Is For

- **Swift developers starting a UIKit job** who learned with SwiftUI and now face a large existing UIKit codebase.
- **iOS developers who learned UIKit years ago** and want to catch up on modern APIs: diffable data sources, compositional layout, content configurations, sheets, and Swift 6.
- **Developers maintaining production apps** who need solid habits for layout debugging, architecture, accessibility, testing, and incremental SwiftUI adoption.
- **Developers moving from Android, web, or cross-platform frameworks** who already know a programming language and basic Swift and want to build native UIKit apps properly.

You should know basic Swift — optionals, structs and classes, closures, protocols, and `async` / `await` at a beginner level — and have opened Xcode before. We do not re-teach the Swift language or SwiftUI, except where SwiftUI interoperability is the topic.

What You Will Learn

- How UIKit works underneath: the app and scene life cycle, windows, the view controller life cycle, the view hierarchy, the layout pass, hit testing, and the responder chain.
- How to build any layout with Auto Layout, stack views, scroll views, safe areas, size classes, traits, and Dynamic Type — and how to debug layouts when they break.

- How to build lists and grids with table views and collection views, diffable data sources, compositional layout, cell registrations, content configurations, prefetching, swipe actions, context menus, and drag and drop.
- How to structure navigation with navigation controllers, tab bars, split views, coordinators, dependency injection, and MVVM.
- How to present sheets, popovers, alerts, and menus, build forms with solid keyboard handling, and create custom controls and gestures.
- How to polish an app with text, images, SF Symbols, animations, custom transitions, Core Animation, dark mode, and thorough accessibility support.
- How to run UIKit in a modern production app: SwiftUI interop in both directions, Swift 6 concurrency and the main actor, testing at every level, performance, state restoration, and maintaining a large codebase.

How This Book Is Organized

Part I — UIKit Foundations (Chapters 1–5) explains how UIKit works: the life cycle, view controllers, views and rendering, and storyboards versus code. **Part II — Layout** (Chapters 6–10) covers Auto Layout, stack views, adaptive layout, scroll views, safe areas, self-sizing, Dynamic Type, and layout debugging. **Part III — Lists and Collection Views** (Chapters 11–15) moves from classic table views to diffable data sources, compositional layout, modern cells, and advanced collection features. **Part IV — Navigation and Interaction** (Chapters 16–20) covers navigation containers, coordinators and architecture, modal presentation, forms and the keyboard, and controls and gestures. **Part V — Polish: Visuals, Motion, and Accessibility** (Chapters 21–24) covers text and images, animations and transitions, Core Animation, dark mode, and accessibility. **Part VI — Modern UIKit in Production** (Chapters 25–28) covers SwiftUI interop, Swift 6 concurrency in UIKit, testing, and a closing case study on performance, restoration, and maintaining large codebases. The appendices provide a cheat sheet, templates and reference tables, a production checklist, and a glossary.

Each chapter includes hands-on code, exercises in a **Try It Yourself** section, and a **Key Takeaways** box for quick review.

The Running Example: Harborline Ferries

Throughout the book we build the iOS app for **Harborline Ferries**, a fictional ferry operator that runs passenger and vehicle ferries between five ports: Harborline Central, Gull Island, Pinecrest Bay, Saltmarsh Point, and North Cove. Travelers use the app to check today's timetable, see which sailings are delayed, cancelled, or sold out, book seats and vehicle spaces with a clear fare breakdown, keep their trips and boarding passes, and browse the ports. The app has four tabs — Timetable, My Trips, Ports, and Settings — and grows chapter by chapter from an empty window into a well-architected,

accessible, tested app. The sample data is small and fixed, so every fare and total in the book can be checked by hand; our main worked example, booking `HLB-7Q4K2` for two adults, one child, and a car on the 8:00 AM sailing to Pinecrest Bay, totals exactly $88.25.

Versions and Conventions

The code uses the **Swift 6 language mode** with complete concurrency checking, Xcode 26-era tooling, and a **deployment target of iOS 18**, so every iOS 18 API is used freely. Where a feature requires **iOS 26**, the code wraps it in an availability check such as `if #available(iOS 26, *)` and the text says so. Apple's frameworks evolve every year; if an API or setting is not exactly where the book says, check Apple's current documentation for your SDK. Screens are described in words and with original ASCII sketches; displayed values use US English formatting such as `7:00 AM` and `$21.00`.

> **TIP**
>
> A recommended practice or a technique that makes your work easier.

> **NOTE**
>
> Background information, version differences, or a useful aside.

> **WARNING**
>
> A common mistake or something that can cause crashes, data loss, or a poor user experience.

Code appears in `monospace`, with the file path shown above each listing, such as `Harborline/Features/Timetable/TimetableViewController.swift`. Long lines are wrapped to fit the page. When a listing extends code from an earlier chapter, `// ...` marks the parts that are unchanged. Output that depends on your device or simulator is introduced with "you should see something like".

Contents

Preface

PART I — UIKIT FOUNDATIONS

1 UIKit Today: Why It Still Matters, How It Works, and What We Will Build

2 App and Scene Lifecycle: AppDelegate, SceneDelegate, and Windows Without Storyboards

3 View Controllers in Depth: Lifecycle, Responsibilities, and Containment

4 Views and Rendering: Frames, Bounds, Hit Testing, and Custom Drawing

5 Storyboards, XIBs, and Programmatic UI: Reading Old Code and Writing New Code

PART II — LAYOUT

6 Auto Layout Fundamentals: Anchors, Constraints, and Priorities

7 Stack Views and Adaptive Layout: UIStackView, Size Classes, and Traits

8 Scroll Views, Safe Areas, and Layout Margins

9 Self-Sizing Views and Dynamic Type

10 Debugging and Tuning Layout: Conflicts, Ambiguity, and the View Debugger

PART III — LISTS AND COLLECTION VIEWS

11 UITableView Essentials: Data Sources, Delegates, and Cell Reuse

12 Diffable Data Sources: Snapshots, Identifiers, and Animated Updates

13 Compositional Layout: Items, Groups, Sections, and Orthogonal Scrolling

14 Modern Cells: List Configuration, Content Configurations, and Registrations

15 Advanced Collections: Images, Prefetching, Swipe Actions, Menus, and Drag and Drop

PART IV — NAVIGATION AND INTERACTION

16 Navigation Controllers, Tab Bars, and Split Views

17 Coordinators and App Architecture: Routing, Dependency Injection, and MVVM

18 Modal Presentation: Sheets, Detents, Popovers, Alerts, and Menus

19 Forms, Text Input, and Keyboard Handling

20 Controls and Gestures: Buttons, Custom UIControls, and Gesture Recognizers

PART V — POLISH: VISUALS, MOTION, AND ACCESSIBILITY

21 Text, Fonts, Images, and SF Symbols

22 UIKit Animations and Custom View Controller Transitions

23 Core Animation Essentials: Layers, Shadows, Gradients, and Shape Animations

24 Dark Mode, Accessibility, and Inclusive Design

PART VI — MODERN UIKIT IN PRODUCTION

25 SwiftUI and UIKit Together: Hosting Controllers and Representables

26 Swift 6 Concurrency in UIKit: MainActor, Tasks, Loading State, and Observation

27 Testing UIKit Apps: Swift Testing, View Controller Tests, and UI Tests

28 Case Study: Shipping and Maintaining Harborline — Performance, Restoration, and Large Codebases

APPENDICES

A UIKit Cheat Sheet

B Templates and Reference

C Production Checklist

D Glossary

PART I

UIKit Foundations

How UIKit really works: the app and scene life cycle, windows without storyboards, the view controller life cycle and containment, the view hierarchy, rendering, hit testing, and the responder chain — and how to read storyboard-based code while writing new screens in Swift.

1 · UIKit Today

2 · App and Scene Lifecycle

3 · View Controllers in Depth

4 · Views and Rendering

5 · Storyboards, XIBs, and Programmatic UI

CHAPTER 1 · PART I — UIKIT FOUNDATIONS

UIKit Today: Why It Still Matters, How It Works, and What We Will Build

UIKit is the framework underneath a very large number of production iOS apps, and it is still the most precise tool Apple gives you for lists, layout, text, and custom interaction. In this chapter we look at why UIKit matters in a SwiftUI era, how an event travels from a finger on the glass to your code, which objects make up a running app, and what the Harborline Ferries app we build in this book looks like. We finish by creating the Xcode project and running a first screen.

Why UIKit Still Matters

UIKit shipped with the first iPhone SDK and has grown with every release since. SwiftUI arrived much later, and it is an excellent framework for many screens. So why would a professional developer invest in UIKit today? The honest answer is that most iOS jobs require it, and many hard problems are still easiest to solve with it.

Think about the apps you might join as a new team member. A banking app, an airline app, a hospital scheduling tool, or a ride-sharing client was very often started before SwiftUI existed, and it has years of features, tests, and bug fixes built on view controllers, table views, and Auto Layout. Nobody rewrites a working app of that size in one go; SwiftUI is introduced screen by screen where it pays off. To work in such a codebase you have to read UIKit fluently, and you have to write new UIKit code that fits in.

There are also technical reasons. UIKit offers mature, highly tuned APIs for long and complex lists, with fine control over cell reuse, prefetching, and animated updates. Its text system gives you deep access to text layout, selection, and input. Auto Layout lets you specify exactly how a screen behaves when space runs out. Custom controls, cooperating gesture recognizers, and interactive transitions get direct, imperative control over every step.

Finally, SwiftUI on iOS does not replace UIKit underneath; the two frameworks live side by side in the same app process. Navigation bars, tab bars, sheets, text input, and many other pieces of a SwiftUI app interoperate with UIKit infrastructure, and Apple provides first-class bridges in both directions: `UIHostingController` shows SwiftUI inside UIKit, and `UIViewRepresentable` shows UIKit inside SwiftUI. Understanding UIKit makes you better at SwiftUI too, because you understand the platform that both are built for.

UIKit and SwiftUI as complements

This book does not argue that one framework is better. Teams use both, and the useful question is which tool fits a given screen. The table below summarizes how we think about the choice. It is a guide, not a rule; Chapter 25 shows the interop techniques that let you mix the two freely.

Situation	Usually a good fit	Why
Existing UIKit app, new screen in an existing flow	UIKit (or SwiftUI hosted)	Matches surrounding code, coordinators, and tests
Very long or complex lists with custom updates	UIKit collection views	Diffable data sources, precise reuse and prefetching control
Simple settings or form-like screen	SwiftUI	Less code, live previews, easy state binding
Custom gestures, interactive transitions	UIKit	Imperative control over every frame of the interaction
Rich text editing	UIKit text views	Direct access to the text system and input handling

In Harborline, every screen is UIKit except Settings, which is a small SwiftUI form hosted inside a `UIHostingController`. That mirrors what many real teams do: UIKit as the backbone, SwiftUI where it is clearly simpler.

How UIKit Works

UIKit is an **imperative**, **object-oriented**, **event-driven** framework that runs on the **main thread**. Each of those words matters when you debug real code, so let us unpack them.

- **Imperative.** You tell UIKit what to do, step by step: create a label, set its text, add it to a view, activate constraints. When data changes, you update the views yourself (or ask a data source to compute the changes). SwiftUI is declarative; UIKit is not, and a large part of good UIKit code is keeping views in sync with state in a disciplined way.
- **Object-oriented.** Screens and components are classes: `UIViewController`, `UIView`, `UIControl`, and their many subclasses. You customize behavior by subclassing and overriding methods, by setting delegates, and by composing objects. Objects have identity and a lifetime, which is why memory management and retain cycles come up again and again.
- **Event-driven.** Your code mostly runs in response to something: the app launched, a scene connected, a view appeared, the user tapped a button, a network response arrived. UIKit calls you; you rarely call UIKit in a loop.
- **Main thread.** UIKit objects must be used from the main thread. In Swift 6 this is expressed in the type system: `UIView`, `UIViewController`, `UIApplication`, and most other UIKit types are isolated to the `@MainActor`, so the compiler checks that you only touch them from main-actor code.

The main run loop in one picture

Every iOS app has a main thread that runs a **run loop**: a loop that waits for something to happen, handles it, and goes back to waiting. Touches, timers, network callbacks scheduled on the main queue, and system notifications all arrive through it. The diagram below follows one tap from the glass to your code and then to the screen.

One turn of the main run loop

```
finger touches the screen
        |
        v
+------------------+   the system packages the touch as a UIEvent
|  main run loop   |   and hands it to the app's main thread
+------------------+
        |
        v
UIApplication.sendEvent(_:)
        |
        v
UIWindow.sendEvent(_:) ----> hit test: which view contains the point?
        |                    window.hitTest(point, with: event)
        |                          |
        |                          v
        |                    deepest visible, interactive view
        v
gesture recognizers, UIControl target-action, touchesBegan(_:with:)
        |
        | not handled? the event moves up the responder chain:
        | view -> superview -> view controller -> ... -> window -> application
        v
YOUR CODE RUNS (update state, set label text, change constraints)
        |
        v
end of this turn of the run loop:
  update constraints -> layout (layoutSubviews) -> display (draw)
  -> changes committed to the render server, which draws the next frame
```

Three ideas from this picture will follow us through the whole book.

First, **hit testing** decides which view receives a touch. The window asks its subviews, from front to back, whether the point is inside them, and recursion finds the deepest view that is visible and accepts interaction. Chapter 4 shows the algorithm and how to bend it, for example to enlarge a small button's tap area.

Second, the **responder chain** is the path an unhandled event or action takes upward: from a view to its superview, to the view controller that owns it, through container controllers, to the window, the scene, and finally the application and its delegate. Menus, keyboard shortcuts, and actions sent to a nil target all rely on it.

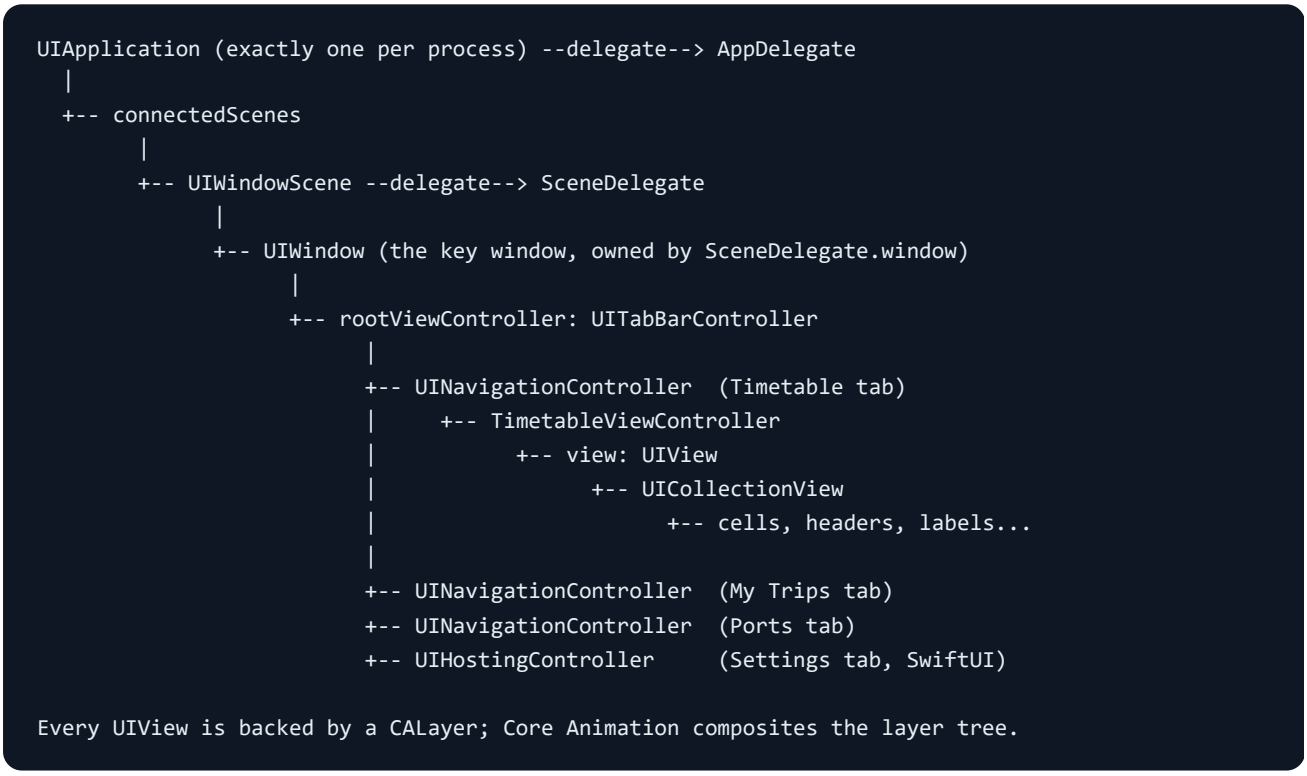
Third, **UIKit batches visual work**. When you set a label's text or change a constraint, nothing is drawn immediately. UIKit marks the affected views as needing layout or display, and at the end of the current run-loop turn it performs one layout pass and one display pass for everything that changed. This is why you can change ten properties in a row without ten redraws, and why code that blocks the main thread for a long time freezes the entire interface: the run loop cannot reach the end of its turn, so nothing new reaches the screen and no new touches are processed.

> **WARNING**
>
> Never perform slow work, such as parsing a large file, waiting on the network, or decoding big images, synchronously on the main thread. The user sees a frozen app, and the system may terminate apps that stay unresponsive for too long. Chapter 26 shows how Swift concurrency moves that work off the main actor and brings results back safely.

The core object graph

When Harborline is running, a small number of long-lived objects form the skeleton that everything else hangs from. Knowing this graph by heart makes crash logs, view debugger output, and memory graphs much easier to read.

Objects in a running Harborline app



- `UIApplication` represents the running process. There is exactly one, created for you at launch. It receives events from the system and forwards them, and it talks to your **app delegate** about process-level events such as launch and push-notification registration.

- `UIWindowScene` represents one instance of your app's user interface. On iPhone there is usually one scene; on iPad the user can open several windows of the same app side by side, and each one is a separate scene with its own **scene delegate**. Chapter 2 covers the scene life cycle in depth.
- `UIWindow` is the root of a scene's view hierarchy. It is itself a view, it delivers events to its subviews, and it holds the **root view controller**.
- **View controllers** manage a screen or part of a screen. Container view controllers such as `UITabBarController` and `UINavigationController` manage other view controllers. Chapter 3 is all about view controllers.
- **Views** draw content and receive touches. They form a tree: every view has at most one superview and any number of subviews. Chapter 4 explores frames, bounds, drawing, and hit testing.
- **Layers** (`CALayer`) hold the actual visual content that the system composites. Each view has one, and you can add more. Chapter 23 covers Core Animation.

> **NOTE**
>
> In Swift 6 language mode the compiler enforces that UIKit objects are used from the main actor. A `Task { }` started inside a view controller method inherits main-actor isolation, so it can update views directly. Values that cross from a background actor back to the main actor must be `Sendable`; Harborline's models are all `Sendable` structs for exactly this reason.

Meet Harborline Ferries

Throughout the book we build one realistic app instead of many disconnected snippets. **Harborline Ferries** is a fictional ferry operator that runs passenger and vehicle ferries between five ports: Harborline Central on the mainland, Gull Island and Saltmarsh Point in the Outer Islands, and Pinecrest Bay and North Cove on the North Shore. Its fleet has four vessels, from the small MV Puffin to the large MV Cormorant, which carries up to 60 vehicles.

The app is called simply "Harborline". It shows a live timetable, lets travelers open a sailing's details, book seats and a vehicle space with a clear fare breakdown, keep their trips and boarding passes in one place, browse ports, and adjust a few settings. Everything is built with UIKit, with SwiftUI used only where interop is the lesson.

Harborline talks to a fictional API at `https://api.harborline.example/v1/`. The `.example` domain is reserved and never resolves, so the app runs on carefully prepared mock data by default. The demo clock is fixed at **Monday, June 15, 2026, 6:30 AM** in the port time zone (Pacific time). Fixing "now" makes every screenshot in this book reproducible and every test deterministic: at that moment the 6:45 AM sailing to North Cove has just closed for booking, the 8:00 AM sailing to Pinecrest Bay is running fifteen minutes late, and the 5:15 PM sailing has been cancelled because of high winds.

A tour of the four tabs

The app has four tabs, from left to right: **Timetable**, **My Trips**, **Ports**, and **Settings**. Here is what each looks like when finished. The sketches are simplified, but every label and value is what the real screens show.

Timetable tab (Chapters 12–14)

```
+-----------------------------------------+
| Timetable                 From: Harborl.. v|
| [Mon 15] [Tue 16] [Wed 17] [Thu 18] [Fri ..|
|                                         |
| Harborline Central to Gull Island       |
| +-------------------------------------+ |
| | 7:00 AM - 7:45 AM          On time  | |
| | MV Tern - 52 seats left           > | |
| | 9:30 AM - 10:15 AM         Sold out | |
| | MV Tern - Sold out                > | |
| | 1:00 PM - 1:45 PM          On time  | |
| | MV Tern - 117 seats left          > | |
| +-------------------------------------+ |
| Harborline Central to Pinecrest Bay     |
| +-------------------------------------+ |
| | 8:00 AM - 9:10 AM    Delayed 15 min | |
| | MV Sea Lark - 205 seats left      > | |
| | ...                                 | |
+-----------------------------------------+
| Timetable | My Trips |  Ports  | Settings  |
+-----------------------------------------+
```

The Timetable lists the day's sailings grouped by route. A strip of seven day chips, Monday June 15 through Sunday June 21, sits at the top, and a menu in the navigation bar picks the departure port. By default it shows sailings from Harborline Central: seven sailings in three route sections. Choosing "All ports" adds the Gull Island routes for a total of nine sailings in five sections. Each row shows a colored status badge with a text label, because color alone must never carry meaning.

Sailing details, pushed from the timetable (Chapters 3–9, 20)

```
+-----------------------------------------+
| < Timetable          HL-101             |
|                                         |
|  7:00 AM                       7:45 AM  |
|  Harborline   o- - - - - - - - -o  Gull  |
|  Central                         Island |
|                                         |
|  MV Tern                     [On time]  |
|  45 min crossing           52 seats left|
|                                         |
|  Fares                                  |
|  Adult                          $14.50  |
|  Child                           $7.25  |
|  Senior                         $10.15  |
|                                         |
```

```
|  +-------------------------------------+  |
|  |         Book This Sailing           |  |
|  |            From $14.50              |  |
|  +-------------------------------------+  |
+-------------------------------------------+
```

Tapping a sailing pushes its detail screen. It shows departure and arrival times, a small drawing of the route with a dot for each port, the vessel and its status, and a fare summary. The fare summary is a child view controller, which is the containment example in Chapter 3. The "Book This Sailing" button starts the booking flow in a sheet.

Booking sheet, final summary (Chapters 18–20)

```
+-------------------------------------------+
|                  -----                    |
| Cancel        Review Booking              |
|                                           |
| HL-201  Harborline Central to Pinecrest Bay|
| Mon, Jun 15 - 8:00 AM - MV Sea Lark       |
|                                           |
| Adult      2 x $21.00             $42.00  |
| Child      1 x $10.50             $10.50  |
| Vehicle    1 x $32.00             $32.00  |
| Port fees  3 x $1.25               $3.75  |
| ----------------------------------------- |
| Total                             $88.25  |
|                                           |
| Lead passenger: Mara Lindqvist            |
| mara@example.com                          |
|                                           |
|  [           Confirm Booking           ] |
+-------------------------------------------+
```

The booking flow is the book's main worked example. Mara Lindqvist books two adults, one child, and one car on sailing HL-201 to Pinecrest Bay. The sheet walks through passenger counts, passenger details, a summary, and a confirmation with the booking reference **HLB-7Q4K2**. The total is $88.25, and every number is computed with integer cents, never floating point.

My Trips tab (Chapters 11–12, 21)

```
+-------------------------------------------+
| My Trips                                  |
|                                           |
| UPCOMING                                  |
| +---------------------------------------+ |
| | Mon, Jun 15 - 8:00 AM                 | |
| | Harborline Central to Pinecrest Bay   | |
| | HLB-7Q4K2 - 3 passengers, 1 vehicle  > | |
| +---------------------------------------+ |
| | Mon, Jun 15 - 1:00 PM                 | |
| | Harborline Central to Gull Island     | |
| | HLB-3D8PX - 2 passengers            > | |
| +---------------------------------------+ |
```

```
|                                          |
| PAST                                     |
|   No past trips                          |
+------------------------------------------+
| Timetable | My Trips |  Ports  | Settings  |
+------------------------------------------+
```

My Trips lists bookings, sorted by departure. The mock store comes with one pre-seeded trip for Tomas Reyes, reference HLB-3D8PX on the 1:00 PM sailing to Gull Island, so the screen is never empty during development. Mara's booking appears after you complete the booking flow. Selecting a trip shows a boarding pass with a QR code. We build this screen first with a classic table view data source in Chapter 11, then migrate it to a diffable data source in Chapter 12, which is exactly the journey many real codebases are on.

Ports tab (Chapters 13, 15, 16)

```
+------------------------------------------+
| Ports                       [ Search   ] |
|                                          |
|  Featured                                |
|   +--------------------------------+  +--|
|   |      [ photo of Gull Island ]  |  |  |
|   |  Gull Island                   |  |  |
|   |  Outer Islands                 |  |  |
|   +--------------------------------+  +--|
|                                          |
|  All Ports                               |
|  +-------------------+ +-----------------+ |
|  | [photo]           | | [photo]         | |
|  | Harborline Central| | Gull Island     | |
|  | Mainland          | | Outer Islands   | |
|  +-------------------+ +-----------------+ |
|  | Pinecrest Bay     | | Saltmarsh Point | |
|  | ...               | | ...             | |
+------------------------------------------+
```

The Ports tab is a compositional layout showcase: a paging carousel of featured ports on top and a grid below that shows two columns on iPhone and more on wide iPad windows. Photos load asynchronously through an actor-based image loader, which is where we tackle the classic cell-reuse race in Chapter 15.

Settings tab, SwiftUI (Chapter 25)

```
+------------------------------------------+
| Settings                                 |
|                                          |
| TRAVEL                                   |
| Show vehicle fares                [ ON ] |
| Default departure port                   |
|                       Harborline Central > |
|                                          |
| REMINDERS                                |
```

```
| Departure reminders                [OFF ] |
|                                           |
| ABOUT                                     |
| About Harborline                        > |
+-------------------------------------------+
```

Settings is a SwiftUI form backed by an observable model that UIKit screens can read too. The "About Harborline" row opens the one screen in the app built with a storyboard, which we keep deliberately as a reading exercise for legacy code in Chapter 5. The reminder switch is interface only; Harborline does not schedule notifications.

The data behind the screens

Because the demo data is fixed, you can check every screen against known values. The table below lists the five routes. Child fares are 50 percent and senior fares 70 percent of the adult fare, rounded half-up to the cent; a vehicle costs $32.00 on vessels with a car deck, and every passenger pays a $1.25 port fee.

Route	Duration	Adult	Child	Senior
HLC-GUL	45 min	$14.50	$7.25	$10.15
GUL-HLC	45 min	$14.50	$7.25	$10.15
HLC-PNC	1 hr 10 min	$21.00	$10.50	$14.70
GUL-SLT	30 min	$9.75	$4.88	$6.83
HLC-NCV	1 hr 35 min	$27.25	$13.63	$19.08

Nine sailings run on the demo Monday, from HL-401 at 6:45 AM to HL-403 at 5:15 PM. Appendix B lists them all with seats and vehicle spaces. You do not need to memorize any of this; we introduce each value when a chapter uses it, and the tests in Chapter 27 pin them down.

The Project Layout

A consistent folder structure helps both humans and tools. Harborline groups code by layer at the top and by feature inside `Features`. Every file caption in this book uses these paths, so you always know where a listing belongs.

Harborline project structure

```
Harborline/
  App/          AppDelegate.swift  SceneDelegate.swift  AppCoordinator.swift
                AppEnvironment.swift  Coordinator.swift
  Models/       Port.swift  Route.swift  Vessel.swift  Sailing.swift  Money.swift
                Fare.swift  Booking.swift  SampleData.swift
  Services/     TimetableService.swift  MockTimetableService.swift
                LiveTimetableService.swift  BookingStore.swift  ImageLoader.swift
                BookingReference.swift
```

```
  Features/
    Timetable/  TimetableCoordinator.swift  TimetableViewController.swift
                TimetableViewModel.swift  SailingCell.swift
                SailingContentConfiguration.swift  DayStripCell.swift
    SailingDetail/  SailingDetailViewController.swift  VoyageProgressView.swift
    Booking/    BookingCoordinator.swift  PassengerCountViewController.swift
                PassengerDetailsViewController.swift  BookingSummaryViewController.swift
                BookingConfirmationViewController.swift
    Trips/      TripsCoordinator.swift  MyTripsViewController.swift
                TripTableViewCell.swift  BoardingPassViewController.swift
    Ports/      PortsViewController.swift  PortCell.swift  PortDetailViewController.swift
    Settings/   SettingsView.swift (SwiftUI)  SettingsModel.swift
    About/      About.storyboard  AboutViewController.swift  FareRulesView.xib
                FareRulesView.swift
  Shared/       Theme.swift  HarborFormat.swift  UIView+Layout.swift
                StatusBadgeView.swift  FareStepperControl.swift  RouteLineView.swift
                AccessibilityID.swift
  Resources/    Assets.xcassets  LaunchScreen.storyboard  Localizable.xcstrings
                Info.plist
HarborlineTests/   FareCalculatorTests.swift  BookingValidatorTests.swift
                   TimetableViewModelTests.swift  TimetableViewControllerTests.swift
HarborlineUITests/ BookingFlowUITests.swift
```

Models are plain Swift value types with no UIKit import, so they are easy to test and safe to pass between actors. **Services** hide where data comes from behind protocols, so the app can run on mock data and tests can inject anything they need. **Features** keep each screen's view controllers, cells, and coordinators together, which keeps merge conflicts local when a team grows. **Shared** holds the small reusable pieces: colors, formatting, layout helpers, and custom views and controls used by more than one feature.

The app is wired together by **coordinators**, objects that own navigation for a flow, and by an **environment** object that carries dependencies such as the timetable service, the booking store, the image loader, and the clock. View controllers receive what they need through their initializers and report user intent through closures such as `onSelectSailing`. We stub the first coordinator in Chapter 2 and explain the pattern fully in Chapter 17.

Versions and Conventions

Every listing in this book follows the same conventions. Knowing them up front makes the code easier to read and explains choices that might otherwise look arbitrary.

Topic	Convention in this book
Language	Swift 6 language mode with complete data-race checking
Tooling	Xcode 26-era tooling; menu names can differ slightly between versions
Deployment target	iOS 18.0, so every iOS 18 API is used without availability checks
Newer APIs	iOS 26 features only inside `if #available(iOS 26, *)` with a fallback

Screens	Built in code; `final class` view controllers with injected dependencies
Money	`Money(cents: Int)`, never `Double`
UIKit constants	Computed `static var` for colors and fonts, never `static let`
Logging	`Logger` from the `os` module instead of `print`

Two of these deserve a word of explanation now.

Why iOS 18? Choosing a recent minimum lets us use modern APIs such as the `UITab` tab bar API, cell registrations, content configurations, and the trait-change registration API without wrapping each call in availability checks. Real apps often support older versions, and where an API is newer than iOS 18 we say so and show the availability check. When you build with the iOS 26 SDK, standard bars, sheets, and controls adopt the new system design that Apple calls Liquid Glass automatically; we point out where that affects your code, and you should check the current documentation for details.

Why computed colors? In Swift 6 mode, a `static let` stored property must hold a `Sendable` value, because any thread could read it. Several UIKit types are not marked `Sendable` in every SDK, so declaring `static let seaBlue = UIColor(...)` can produce an error. A computed `static var` creates the value on demand and sidesteps the problem at negligible cost. You will see this pattern in `Theme.swift` in Chapter 2.

> **TIP**
>
> New app projects in recent Xcode versions may enable a **default actor isolation** build setting that treats your code as `@MainActor` unless you say otherwise, together with other "approachable concurrency" options. The code in this book is written to compile either way, but the error messages you see can differ. Check your target's build settings early so that you know which mode you are in.

Creating the Xcode Project

Time to start building. We create the project once here and grow it chapter by chapter. Menu labels and template options change a little between Xcode releases, so treat the steps below as a guide and look for the closest match in your version.

1. In Xcode choose **File > New > Project**, select the **iOS** platform, and pick the **App** template.
2. Enter **Harborline** as the product name and **com.example** as the organization identifier. Xcode combines them into a bundle identifier. Harborline uses `com.example.harborline`; if Xcode shows a different capitalization, edit it in the target's Signing settings after creating the project.
3. For **Interface**, choose **Storyboard**. This is the only UIKit option the template offers; the other choice creates a SwiftUI app. We will delete the storyboard in Chapter 2 and build the window in code, but starting from this template gives us the app delegate, scene delegate, and scene manifest we need.