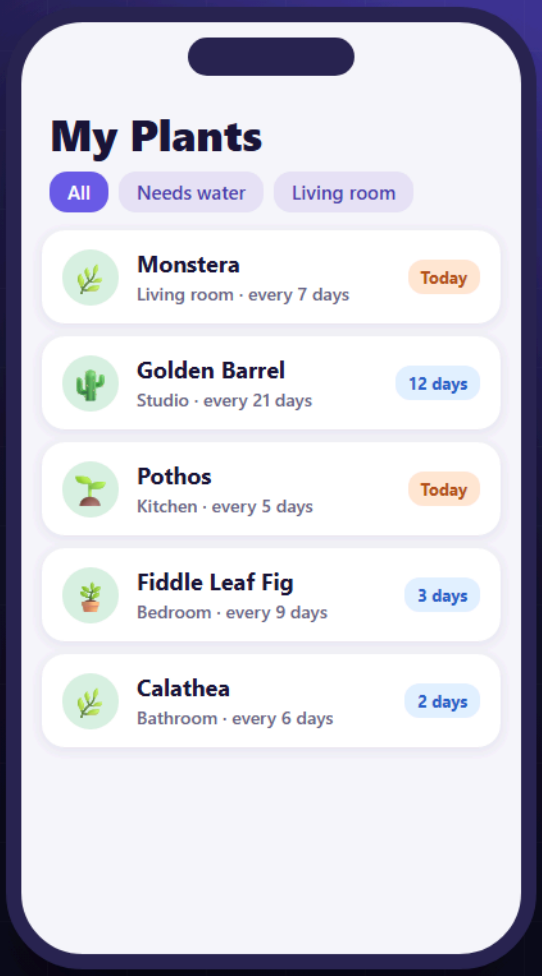
```swift
// SwiftUI
struct PlantList: View {
  @Query var plants: [Plant]

  var body: some View {
    List(plants) { plant in
      PlantRow(plant)
    }
  }
}

// async/await
try await api.species()
```



THE COMPLETE PRACTICAL GUIDE

iOS App Development with Swift and SwiftUI

From Zero to the App Store

Swift · SwiftUI · SwiftData · Concurrency · Networking · Widgets · Testing · TestFlight · App Store Release

28 CHAPTERS · 4 APPENDICES · ONE COMPLETE APP

THENEXBOOKS

iOS App Development with Swift and SwiftUI

From Zero to the App Store

Swift · SwiftUI · SwiftData · Concurrency · Widgets · Testing · Release

First Edition

THENEXBOOKS

iOS App Development with Swift and SwiftUI
From Zero to the App Store

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. Swift, Xcode, Apple's frameworks, and App Store policies change frequently; always consult Apple's current documentation and guidelines.

Apple, iPhone, iPad, iOS, iPadOS, macOS, watchOS, visionOS, Xcode, Swift, SwiftUI, App Store, and TestFlight are trademarks of Apple Inc. 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 not affiliated with or endorsed by Apple Inc.

Fernwise, the example app in this book, is fictional. Any resemblance to real apps or companies is coincidental.

Preface

There has never been a better time to start building apps for iPhone and iPad. Swift is a modern, safe, expressive language. SwiftUI lets you describe an interface in a few readable lines and see it update live. SwiftData, the Observation framework, and Swift concurrency remove much of the boilerplate and many of the bugs that used to make app development hard. And Xcode puts a simulator, a debugger, and a profiler at your fingertips.

But the modern stack is also new enough that many tutorials still teach older patterns, and it can be hard to know how the pieces fit together in a real app — one that stores data, talks to a server, sends reminders, shows a widget, works on iPad, passes App Review, and keeps working after the next iOS release.

This book is a complete, practical guide to building iOS apps the modern way. We start with the Swift language itself, build interfaces with SwiftUI, structure the app, persist data with SwiftData, use the network and platform features, test and profile, and finally release to the App Store. Every chapter builds on one realistic app, so you see how each technique fits into the whole.

Who This Book Is For

- **Developers new to Apple platforms** — web, backend, Android, Java, or C# developers who want to build native iOS apps.
- **Students and career changers** looking for a structured, complete path into iOS development.
- **iOS developers** moving from UIKit and older patterns to SwiftUI, Observation, SwiftData, and Swift concurrency.
- **Designers and product people** who want to understand how modern iOS apps are built.

You should be comfortable with basic programming concepts. No prior Swift or Apple-platform experience is required. You will need a Mac with Xcode to run the examples.

What You Will Learn

- Swift from the essentials to protocols, generics, error handling, and modern concurrency.
- SwiftUI views, layout, state and data flow, lists, navigation, forms, design, accessibility, and animation.
- App architecture with observable models and dependency injection.
- SwiftData persistence, networking with URLSession and Codable, caching, offline support, and secure storage.
- Notifications, photos, widgets, App Intents, maps, location, and background work.
- Testing with Swift Testing and XCTest, debugging and profiling with Instruments, iPad layouts, and localization.

- Privacy, security, signing, TestFlight, App Store Connect, and life after launch.

How This Book Is Organized

Part I — The Swift Language (Chapters 1–5) sets up Xcode and teaches Swift and concurrency. **Part II — Building Interfaces with SwiftUI** (Chapters 6–11) covers everything you need to build screens. **Part III — Architecture and Data** (Chapters 12–16) structures the app and stores its data. **Part IV — Platform Features** (Chapters 17–21) integrates notifications, photos, widgets, maps, and background work. **Part V — Quality** (Chapters 22–25) adds tests, performance, iPad support, localization, and privacy. **Part VI — Shipping** (Chapters 26–28) brings it all together and releases it. The appendices provide cheat sheets, a release checklist, and a glossary. Each chapter ends with a **Key Takeaways** box.

The Running Example: Fernwise

Throughout the book we build **Fernwise**, a fictional plant-care companion. You will track your plants and their watering schedules, see which plants need water today, browse a species guide from an online API, attach photos, get reminders, add a home-screen widget and a Shortcuts action, find nearby plant shops on a map, and use the app on iPhone and iPad — in several languages.

Conventions Used in This Book

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

> **NOTE**
> Background information, platform requirements, or a useful aside.

> **WARNING**
> A common mistake that causes crashes, data loss, or App Review problems.

Code appears in `monospace`, with the file path above each example. Features that require a specific minimum iOS version are noted. Apple's tools and policies change often, so always check the current documentation.

Contents

Preface

PART I — THE SWIFT LANGUAGE

1 iOS Development Today: The Platform, the Tools, and What We Will Build

2 Xcode, Simulators, and Your First App

3 Swift Essentials: Constants, Types, Optionals, Control Flow, and Functions

4 Swift in Depth: Structs, Classes, Enums, Protocols, Generics, and Errors

5 Swift Concurrency: async/await, Tasks, Actors, and Data-Race Safety

PART II — BUILDING INTERFACES WITH SWIFTUI

6 SwiftUI Fundamentals: Views, Modifiers, and Layout Stacks

7 State and Data Flow: @State, @Binding, @Observable, and the Environment

8 Lists, Grids, and Navigation

9 Forms, Input, Sheets, Alerts, and Confirmation Dialogs

10 Design and Accessibility: SF Symbols, Color, Typography, and Dynamic Type

11 Animation, Transitions, and Gestures

PART III — ARCHITECTURE AND DATA

12 App Architecture: Models, Observable View Models, and Dependency Injection

13 Persisting Data with SwiftData

14 Networking with URLSession, async/await, and Codable

15 Caching, Offline Support, and Sync

16 Settings, Files, and the Keychain

PART IV — PLATFORM FEATURES

17 Local Notifications and Reminders

18 Photos and the Camera

19 Widgets and App Intents

20 Location and Maps

21 Background Work, Scene Phases, and the App Lifecycle

PART V — QUALITY

22 Testing: Swift Testing, XCTest, and UI Tests

23 Debugging and Performance with Xcode and Instruments

24 iPad, Adaptive Layouts, and Localization

25 Security and Privacy

PART VI — SHIPPING

26 Case Study: Building Fernwise End to End

27 Releasing to the App Store: Signing, TestFlight, App Store Connect, and Review

28 After Launch: Analytics, Crash Reports, Updates, UIKit Interop, and Next Steps

APPENDICES

A Swift Cheat Sheet

B SwiftUI Reference

C App Store Release Checklist

D Glossary

PART I

The Swift Language

Set up Xcode, run your first app, and learn the Swift language — from optionals and value types to protocols, generics, and safe concurrency.

1 · iOS Development Today
2 · Xcode and Your First App
3 · Swift Essentials
4 · Swift in Depth
5 · Swift Concurrency

CHAPTER 1 · PART I — THE SWIFT LANGUAGE

iOS Development Today: The Platform, the Tools, and What We Will Build

Before we write a single line of code, we take a guided tour of the modern Apple platform: the Swift language, the SwiftUI framework, the tools you will live in every day, and Fernwise, the plant-care app we will build together from an empty project all the way to the App Store.

Why Learn iOS Development Now

Building apps for iPhone and iPad has changed a great deal over the past several years. If you looked at iOS development a long time ago, you may remember a world of Objective-C, storyboards full of connection lines, view controllers that grew to thousands of lines, and manual memory management rules that tripped up newcomers. That world still exists in older codebases, but it is no longer where new projects begin. Today a new iOS app is typically written entirely in Swift, its interface is described declaratively with SwiftUI, its asynchronous work is expressed with `async` and `await`, and its data can be stored with SwiftData using plain Swift types. The result is a platform that is far more approachable for developers coming from the web, from Android, from backend services, or from languages such as Java, C#, Kotlin, TypeScript, and Python.

This book is written for exactly those developers. We assume that you already know what a variable, a function, and a loop are, and that you have written programs in at least one language. We do not assume that you have touched Swift, Xcode, or any Apple framework before. If you are an experienced iOS developer who learned the platform with UIKit, you will also find plenty here: we spend serious time on SwiftUI's data flow, the Observation framework, structured concurrency, Swift 6 data-race safety, and SwiftData, and we point out where the modern approach differs from what you already know.

There is another reason to learn the platform now: the modern tools reward good habits. The Swift compiler catches whole categories of mistakes before your app ever runs. SwiftUI previews let you see a screen change as you type. The concurrency model makes threading bugs visible at compile time instead of in a crash report. You will spend less time fighting the framework and more time designing the experience your users actually see.

The Apple Platform Landscape

Apple ships several operating systems, and an app you build with Swift and SwiftUI can reach more than one of them. Understanding the landscape helps you make sense of the documentation, because nearly every API page lists the platforms on which that API is available.

Platform	Devices	Relevance to this book
iOS	iPhone	Our primary target throughout
iPadOS	iPad	Adaptive layouts in Chapter 24
macOS	Mac computers	Where Xcode runs; SwiftUI apps can also target it
watchOS	Apple Watch	Shares Swift and SwiftUI skills
tvOS	Apple TV	Shares Swift and SwiftUI skills
visionOS	Apple Vision Pro	Shares Swift and SwiftUI skills

iOS and iPadOS are closely related. They share almost all of their frameworks, and a single Xcode target can produce an app that runs on both iPhone and iPad. When we say "iOS" in this book, we usually mean both, and we call out iPad-specific behavior where it matters. Because SwiftUI is available across all of these platforms, the skills you learn here transfer directly: a `List`, a `NavigationStack`, or a `Toggle` works the same way conceptually on an iPhone, a Mac, or a watch, even if each platform renders it with its own look.

The layers of an iOS app

It helps to think of an iOS app as a stack of layers, each building on the one below. At the bottom sits the operating system kernel and low-level services, which you will rarely touch directly. Above that are the core frameworks, and above those is your own code.



The **Swift standard library** provides the fundamental types: integers, strings, arrays, dictionaries, optionals, and the protocols that tie them together. **Foundation** adds the everyday building blocks of real applications, such as dates, calendars, URLs, networking with `URLSession`, and encoding and decoding JSON. The **platform frameworks** expose device capabilities and system services: storing data, showing notifications, reading photos, drawing maps, and appearing on the Home Screen as a widget. Finally, **SwiftUI** is the framework we use to describe the user interface, while **UIKit** is the older, still fully supported framework that SwiftUI sometimes interoperates with.

Swift: The Language

Swift is the programming language Apple introduced in 2014 as a modern successor to Objective-C. It is open source, it runs on Apple platforms as well as Linux and Windows, and it is the language used for essentially all new Apple-platform development. If you come from another modern language, much of Swift will feel familiar, but a few ideas deserve special attention because they shape everything else.

- **Safety by default.** Variables must be initialized before use, array access is bounds-checked, and integer overflow traps instead of silently wrapping. Most importantly, the absence of a value is modeled explicitly with *optionals*, so the compiler forces you to handle "no value" cases rather than letting a null reference crash your app at runtime.
- **Value types first.** Swift encourages you to model data with `struct` and `enum`, which are copied when assigned. This makes reasoning about data much simpler, because a value you hold cannot be changed behind your back by some other part of the program.
- **Protocols and generics.** Swift favors composition through protocols over deep class hierarchies. SwiftUI itself is built on this idea: every screen element is a value that conforms to the `View` protocol.
- **Expressive syntax.** Type inference, trailing closures, and argument labels make code read almost like prose, which is why SwiftUI code can look so compact.
- **Concurrency built into the language.** `async` / `await`, tasks, and actors are language features, not library add-ons, and Swift 6 introduced a language mode with complete data-race safety checking.

Here is a small taste of Swift. Do not worry about every detail yet; Chapter 3 and Chapter 4 cover all of it. Notice how little ceremony there is: no semicolons, no explicit types where the compiler can infer them, and a clear distinction between values that can change (`var`) and values that cannot (`let`).

Playground: a first look at Swift

```
import Foundation

struct Plant {
    let name: String
    var wateringIntervalDays: Int
    var lastWatered: Date?

    var needsWaterToday: Bool {
        guard let lastWatered else { return true }
        let days = Calendar.current.dateComponents(
            [.day], from: lastWatered, to: .now
        ).day ?? 0
        return days >= wateringIntervalDays
    }
}
```

```
let monstera = Plant(name: "Monstera", wateringIntervalDays: 7, lastWatered: nil)
print("\(monstera.name) needs water: \(monstera.needsWaterToday)")
```

Even this short example shows several of Swift's personality traits. The `Plant` type is a `struct`, a value type. The property `lastWatered` has type `Date?`, which reads as "an optional date": it may hold a date, or it may hold nothing at all. The computed property `needsWaterToday` uses `guard let` to unwrap that optional safely, and the `??` operator supplies a fallback value. String interpolation with `\( )` builds the output text. Every one of these features will become second nature by the end of Part I.

SwiftUI and UIKit

Apple offers two frameworks for building user interfaces on iOS. **UIKit** has been around since the first iPhone SDK. It is an *imperative* framework: you create view objects, add them to a hierarchy, and then mutate them over time as data changes. **SwiftUI**, introduced in 2019, is a *declarative* framework: you describe what the interface should look like for a given state, and the framework figures out how to update the screen when that state changes.

The difference is easiest to see side by side. Suppose we want a label showing how many plants need water, and we want it to update whenever that number changes. In a UIKit view controller, we might write something like this:

UIKit style (for comparison only)

```
import UIKit

final class SummaryViewController: UIViewController {
    private let countLabel = UILabel()

    var thirstyCount = 0 {
        didSet { updateLabel() }   // we must remember to update the UI
    }

    override func viewDidLoad() {
        super.viewDidLoad()
        countLabel.translatesAutoresizingMaskIntoConstraints = false
        view.addSubview(countLabel)
        NSLayoutConstraint.activate([
            countLabel.centerXAnchor.constraint(equalTo: view.centerXAnchor),
            countLabel.centerYAnchor.constraint(equalTo: view.centerYAnchor)
        ])
        updateLabel()
    }

    private func updateLabel() {
        countLabel.text = "\(thirstyCount) plants need water"
    }
}
```

In SwiftUI, the same idea collapses to a description of the result. We declare that the view shows a text built from the current count, and SwiftUI takes care of updating the text when the count changes:

Fernwise/Views/ThirstySummaryView.swift

```
import SwiftUI

struct ThirstySummaryView: View {
    let thirstyCount: Int

    var body: some View {
        Text("\(thirstyCount) plants need water")
            .font(.headline)
            .foregroundStyle(thirstyCount > 0 ? .orange : .secondary)
    }
}

#Preview {
    ThirstySummaryView(thirstyCount: 3)
}
```

There is no code that "updates the label." There is no layout constraint code, because SwiftUI centers content by default and offers simple stack-based layout tools. The `body` property is a function of the data: given `thirstyCount`, it returns the interface. When the data changes, SwiftUI calls `body` again, compares the new description with the old one, and applies only the differences to the screen. This model eliminates an entire class of bugs where the interface falls out of sync with the data.

Should you still learn UIKit? For a new app, SwiftUI is the natural starting point, and it is what we use for every screen of Fernwise. UIKit remains important for maintaining existing apps and for the occasional feature where SwiftUI does not yet offer a native component. SwiftUI and UIKit interoperate well: you can host SwiftUI views inside UIKit and wrap UIKit views for use in SwiftUI. We cover that bridge in Chapter 28, once you are comfortable with SwiftUI itself.

> **NOTE**
>
> You will sometimes read that SwiftUI is "not ready" for serious apps. That opinion was more common in SwiftUI's early years. Each release has filled gaps, and large production apps are built with it today. Where a specific API requires a minimum iOS version, we say so, and we show how to check availability with `if #available`.

The Tools You Will Use

Apple-platform development is centered on a single integrated environment, Xcode, plus a handful of companion tools and services. Chapter 2 walks through Xcode in detail; here we introduce each tool so you know what it is for when it appears later.

Xcode

Xcode is Apple's integrated development environment. It includes the source editor, the Swift compiler, the interface previews for SwiftUI, a visual debugger, the Simulator, the asset catalog editor, the test runner, and the tools to sign and upload your app. Xcode runs only on macOS, which means you need a Mac to build iOS apps. It is free and is distributed through the Mac App Store and the Apple Developer website. Any reasonably recent Mac that can run a current version of macOS will work for the projects in this book, although more memory and a faster processor make builds and the Simulator noticeably snappier.

The Simulator

The Simulator lets you run your app on virtual iPhones and iPads directly on your Mac. You can choose different device sizes, switch between light and dark appearance, change the text size, rotate the device, and simulate locations. The Simulator is where you will run Fernwise most of the time. It is fast to launch and convenient, but it is not a perfect substitute for a physical device: performance characteristics differ, some hardware such as the camera is not available, and certain system behaviors only appear on real hardware.

SwiftUI previews

Previews render a SwiftUI view live inside Xcode's canvas, next to your code. Change a font, add a padding modifier, or swap sample data, and the preview updates within moments. The `#Preview` block you saw in `ThirstySummaryView` declares such a preview. Previews are one of the biggest productivity gains of SwiftUI, and we use them constantly.

Instruments

Instruments is a profiling application bundled with Xcode. It measures CPU usage, memory allocations, leaks, animation hitches, network activity, and much more. You will meet it properly in Chapter 23, when we track down a deliberately slow list in Fernwise.

Swift Package Manager

Swift Package Manager is the official tool for sharing and consuming Swift code. Xcode integrates it directly: you add a package by URL, choose a version rule, and Xcode downloads and builds it with your app. We keep Fernwise free of third-party dependencies so that every line of code in the book is something you can read and understand, but you will learn how packages work so you can use them in your own projects.

App Store Connect and TestFlight

App Store Connect is Apple's web portal for managing apps you distribute: creating the app record, writing the description, uploading screenshots, setting pricing and availability, and submitting builds for review. TestFlight, which is part of the same system, lets you distribute beta builds to testers before release. Chapter 27 walks through both in detail.

The Apple Developer Program

You can download Xcode, write code, and run apps in the Simulator without paying anything. With a free Apple Account you can also install your own apps on your own device for personal testing, with some limitations. To distribute apps through TestFlight or the App Store, you need a membership in the Apple Developer Program. Membership terms, pricing, and the exact list of capabilities change over time, so check the Apple Developer website for the current details at the time you enroll. You do not need to enroll to follow the first two thirds of this book.

> **TIP**
>
> Install Xcode early. It is a large download, and after it installs it may download additional components such as simulator runtimes on first launch. Starting the download now means it will be ready when you reach Chapter 2.

What You Need to Get Started

The requirements for following this book are modest:

1. **A Mac** capable of running a recent version of macOS and a current version of Xcode. Check the Xcode release notes for the exact minimum macOS version.
2. **Xcode**, installed from the Mac App Store or the Apple Developer website.
3. **An Apple Account** (formerly called an Apple ID) to sign in to Xcode and, later, to run apps on a device.
4. **Optionally, an iPhone or iPad** for testing on real hardware. Several later features, such as photos from the camera, notifications on the lock screen, and widgets on the Home Screen, are most satisfying to see on a real device, but the Simulator covers almost everything.
5. **Patience with the first build.** The first time you build a project, Xcode compiles and indexes a lot of code. Subsequent builds are much faster.

You do not need to know Objective-C, and you do not need prior experience with any Apple framework. Everything we use is introduced at the point where we first need it.

Modern Swift Concurrency in One Paragraph

Mobile apps are inherently concurrent. While the user scrolls, the app may be downloading data, decoding images, and writing to a database. Historically, iOS developers handled this with callbacks and dispatch queues, which worked but made code hard to follow and easy to get wrong. Modern Swift lets you write asynchronous code that reads like ordinary sequential code. A function marked `async` can suspend while it waits, and the caller uses `await` to mark the point where suspension may happen. Actors protect mutable state from simultaneous access, and the `@MainActor` attribute ensures that code touching the user interface runs on the main thread. Chapter 5 is devoted to this model, and we use it throughout the networking and persistence chapters.

Preview of Chapter 14: fetching species from the Fernwise API

```
import Foundation

struct Species: Codable, Identifiable {
    let id: String
    let commonName: String
    let preferredLight: String
}

func fetchSpecies() async throws -> [Species] {
    let url = URL(string: "https://api.fernwise.example/v1/species")!
    let (data, _) = try await URLSession.shared.data(from: url)
    return try JSONDecoder().decode([Species].self, from: data)
}
```

Three keywords do the heavy lifting: `async` says the function can suspend, `throws` says it can fail, and `try await` at the call site acknowledges both. The force unwrap on `URL(string:)` is acceptable here only because the string is a constant we control and know to be valid; in Chapter 14 we build URLs more carefully. The domain `api.fernwise.example` is fictional, as the `.example` suffix suggests, and in later chapters we show how to provide sample data so the app works without a real server.

Meet Fernwise

Learning a platform from isolated snippets is like learning a language from a phrase book: you can order coffee, but you cannot hold a conversation. To tie everything together, this book builds one realistic app across its chapters. That app is **Fernwise**, a plant-care companion that helps people keep their houseplants alive.

Fernwise is a good teaching vehicle because its domain is simple to understand yet touches almost every major iOS capability. Plants have names, species, rooms, light requirements, and watering schedules. Users need to see which plants are thirsty today, record when they watered something, look up care advice, get reminders, snap photos, and glance at a widget on their Home Screen. Each of those needs maps naturally onto a platform feature.

The feature list

Feature	What the user sees	Chapter
Plant list	All plants with name, room, and status	8
Add and edit form	A form for name, species, room, light, interval	9
Plant detail	Photo, care info, watering history	8, 12
Needs water today	A filtered list of thirsty plants	7, 8
Species guide	Care advice from the Fernwise API	14, 15
Local persistence	Plants survive app restarts	13

Settings	Units and reminder time	16
Watering reminders	Local notifications at a chosen time	17
Plant photos	Pick or capture a photo of each plant	18
Widget and App Intent	Thirsty plants on the Home Screen	19
Nearby shops map	Plant shops on a map (sample data)	20
iPad and localization	Split view layout, translated text	24
Tests and release	Automated tests, App Store submission	22, 27

A sketch of the main screens

Because this book uses no images, we sketch screens as simple text diagrams. Here is the plant list we will reach by the end of Part II. The top bar holds the title and an add button, a segmented control switches between all plants and thirsty plants, and each row shows a plant's name, room, and a status indicator.

```
+----------------------------------+
|  Fernwise                   [+] |
|  [ All Plants | Needs Water ]    |
+----------------------------------+
|  Monstera            Living Room |
|  Water today                (!)  |
+----------------------------------+
|  Fiddle-Leaf Fig       Bedroom   |
|  In 3 days                       |
+----------------------------------+
|  Snake Plant           Office    |
|  In 10 days                      |
+----------------------------------+
|  [ Plants ]  [ Shops ]  [ Settings ]
+----------------------------------+
```

Tapping a row opens the detail screen, with the plant's photo at the top, its care requirements, a prominent "Mark as Watered" button, and a history of past waterings. The tab bar at the bottom leads to the shop map and to settings.

The core model, previewed

At the heart of Fernwise is the `Plant` model. Over the course of the book it evolves from a simple struct with hard-coded sample data, to an observable model shared between screens, to a persistent SwiftData model. Here is roughly what it will look like in Part I, written entirely with the language features covered in Chapter 3 and Chapter 4:

Fernwise/Models/Plant.swift (early version)

```
import Foundation

enum LightLevel: String, CaseIterable, Codable {
    case low, medium, bright, direct
}

struct Plant: Identifiable, Hashable, Codable {
    let id: UUID
    var name: String
    var species: String
    var room: String
    var light: LightLevel
    var wateringIntervalDays: Int
    var lastWatered: Date?

    init(
        id: UUID = UUID(),
        name: String,
        species: String,
        room: String,
        light: LightLevel = .medium,
        wateringIntervalDays: Int = 7,
        lastWatered: Date? = nil
    ) {
        self.id = id
        self.name = name
        self.species = species
        self.room = room
        self.light = light
        self.wateringIntervalDays = wateringIntervalDays
        self.lastWatered = lastWatered
    }
}

extension Plant {
    static let samples: [Plant] = [
        Plant(name: "Monstera", species: "Monstera deliciosa", room: "Living Room"),
        Plant(name: "Snake Plant", species: "Dracaena trifasciata", room: "Office",
              light: .low, wateringIntervalDays: 14)
    ]
}
```

Do not try to memorize this now. The point is to show you where we are heading: a small, readable model built from value types, with sensible defaults and sample data that we can feed to SwiftUI previews. Every keyword in it, from `enum` and `CaseIterable` to `Identifiable` and the `extension` with `static let samples`, is explained in Part I.

How the code is organized

We organize the Fernwise project into folders by responsibility. This structure is a convention, not a rule enforced by Xcode, but it keeps a growing project navigable:

```
Fernwise/
    FernwiseApp.swift        app entry point
    Models/                  Plant, WateringEvent, Species, LightLevel
    Views/                   PlantListView, PlantDetailView, PlantFormView
    ViewModels/              observable state for complex screens
    Services/                networking, notifications, persistence helpers
    Resources/               asset catalog, localized strings
FernwiseWidget/              widget extension (Chapter 19)
FernwiseTests/               unit tests (Chapter 22)
FernwiseUITests/             UI tests (Chapter 22)
```

Throughout the book, code listings carry a caption showing the file they belong to, such as `Fernwise/Views/PlantListView.swift`. When a listing is just an experiment, the caption says "Playground" or describes the purpose instead.

How This Book Is Organized

The book has six parts, and each part builds on the previous ones.

- **Part I — The Swift Language** (Chapters 1–5) teaches Swift from the ground up: constants and types, optionals, control flow, functions, structs, classes, enums, protocols, generics, closures, error handling, and concurrency.
- **Part II — Building Interfaces with SwiftUI** (Chapters 6–11) covers views, modifiers, layout, state management, lists, navigation, forms, design, accessibility, and animation. By the end of this part, Fernwise has a complete, interactive interface with in-memory data.
- **Part III — Architecture and Data** (Chapters 12–16) adds structure and persistence: observable view models, dependency injection, SwiftData, networking, caching, settings, files, and the Keychain.
- **Part IV — Platform Features** (Chapters 17–21) integrates notifications, photos, widgets, App Intents, maps, and background work.
- **Part V — Quality** (Chapters 22–25) focuses on testing, debugging, performance, iPad layouts, localization, security, and privacy.
- **Part VI — Shipping** (Chapters 26–28) assembles Fernwise end to end, walks through signing, TestFlight, and App Store review, and looks at life after launch.

If you are new to Swift, read Part I in order; everything else depends on it. If you already know Swift well, skim Chapter 3 and Chapter 4, but do read Chapter 5 on concurrency, because the modern model and its data-race safety checking affect how we write nearly every later chapter. Experienced UIKit developers can move faster through Part II, but should pay close attention to Chapter 7, where SwiftUI's data flow differs most from UIKit habits.

Conventions used in this book

- Code you type appears in monospaced listings. Inline references to types, functions, and keywords look like `NavigationStack`.
- Keyboard shortcuts use the Mac symbols' names, for example Command-R to run the app and Command-B to build it.
- Menu paths are written with arrows in prose, such as File, then New, then Project.
- Callout boxes labeled **Tip**, **Note**, and **Warning** highlight shortcuts, background information, and common pitfalls.
- Each chapter ends with a **Key Takeaways** box that summarizes the most important points for quick review.

A Word on Versions and Change

Apple updates its operating systems and Xcode every year, and Swift itself evolves continuously through an open proposal process. This pace is good news, because the platform keeps getting better, but it means that any book has to be careful with version-specific claims. Our approach is straightforward. Where a feature has a well-known minimum version, we state it: `NavigationStack` requires iOS 16, and both the `@Observable` macro and SwiftData require iOS 17. For everything else we refer to "recent versions" of iOS and Xcode and show you how to check availability yourself.

Swift gives us a direct tool for this. When an API is only available on newer systems, you can guard its use at runtime and provide a fallback for older ones:

Playground: checking API availability

```
import SwiftUI

struct PlantCountBadge: View {
    let count: Int

    var body: some View {
        if #available(iOS 17, *) {
            Text("\(count)")
                .contentTransition(.numericText(value: Double(count)))
                .font(.title2.bold())
        } else {
            Text("\(count)")
                .font(.title2.bold())
        }
    }
}
```

For Fernwise we choose a deployment target of iOS 17 so that we can use SwiftData and the Observation framework without workarounds. That choice is common for new apps, but in your own projects you should weigh the features you need against the users you want to reach. We return to deployment targets in Chapter 2, when we create the project.