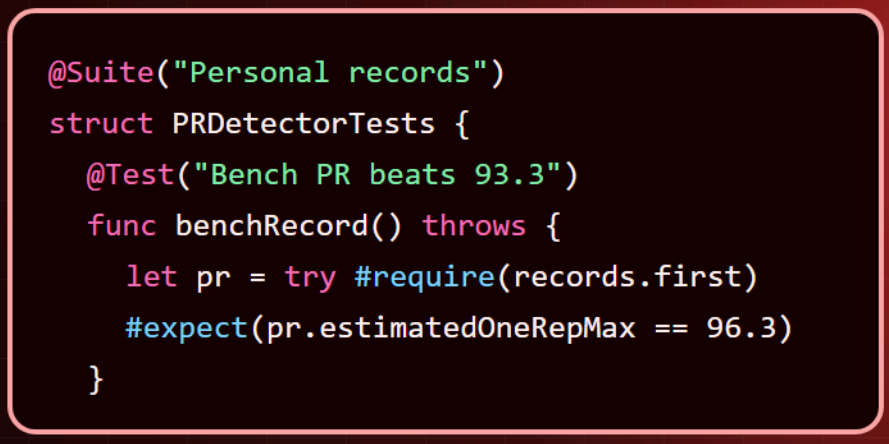
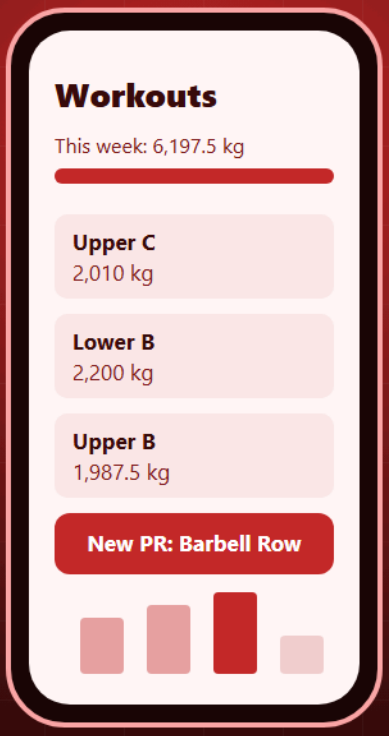
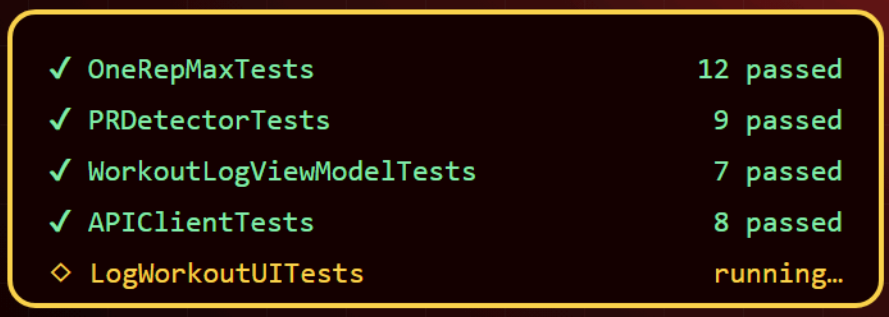








A COMPLETE, PRACTICAL GUIDE

Testing iOS Apps

Swift Testing, XCTest, UI Tests, Snapshot Tests, and Continuous Integration

@Test · #expect · Traits · Test Doubles · Async Tests
XCUITest · Test Plans · Coverage · Flaky Tests · CI

UNIT · INTEGRATION · UI · SNAPSHOT · CI

THENEXBOOKS

Testing iOS Apps

Swift Testing, XCTest, UI Tests, Snapshot Tests, and Continuous Integration

From your first #expect to a trusted test suite running on every pull request

First Edition

THENEXBOOKS

Testing iOS Apps
Swift Testing, XCTest, UI Tests, Snapshot Tests, and Continuous Integration

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, test code, scripts, and CI configuration 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. Xcode, Swift, the iOS SDK, testing frameworks, CI services, and third-party tools change frequently; menu labels, command-line options, runner images, and pricing can differ between versions and over time. Always consult the current documentation for the versions you use. Nothing in the running example is medical, fitness, or training advice; the one-rep-max formula is a common estimate used only as test material.

Apple, iPhone, iPad, iOS, iPadOS, macOS, Mac, Xcode, Xcode Cloud, Swift, SwiftUI, SwiftData, App Store, TestFlight, and Instruments are trademarks of Apple Inc., registered in the U.S. and other countries and regions. GitHub and GitHub Actions are trademarks of GitHub, Inc. Other product and company names, including those of open-source projects such as fastlane and swift-snapshot-testing, are the property 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. or GitHub, Inc. All screen mockups in this book are original illustrations.

Fitlark, the workout logger app used as the running example in this book, is fictional, as are its exercises, workouts, records, API, issue numbers, and all sample data. Domains use the reserved `example` top-level domain.

Preface

Most iOS teams do not lack tests because they do not care. They lack them because the first tests they wrote were slow, flaky, or painful to maintain, and the suite quietly stopped being trusted. A UI test fails every third run on the build server, someone adds a retry, then someone disables it. A view model test passes on one Mac and fails on another because of a time zone. Coverage reaches a number nobody believes in. The result is the worst of both worlds: the cost of a test suite without the confidence it should buy.

This book is about building the other kind of suite: one that is fast enough to run on every change, honest enough that a red result means a real problem, and structured so that adding the next test is easy. We use the modern tools Apple ships — **Swift Testing** for most tests, **XCTest** where it is still the right tool, **XCUITest** for the flows that matter — and we wire everything into **continuous integration** with `xcodebuild`, GitHub Actions, Xcode Cloud, and fastlane.

We do not stop at the API. Each technique appears at the moment it solves a real problem in a real app, and we explain *why* a test is shaped the way it is: what makes code testable, which test double to choose, how to control time, locale, and concurrency, and how to tell a flaky test from a real bug.

Who This Book Is For

- **iOS developers who build SwiftUI apps** and want to start testing properly, or who have a handful of tests and want a strategy instead of a collection.
- **Developers moving from XCTest to Swift Testing** who want to know what changes, what stays, and how to migrate gradually.
- **Teams fighting slow or flaky suites**, especially UI tests, who need concrete diagnosis techniques and policies.
- **Developers responsible for CI** who need to run iOS tests on GitHub Actions or Xcode Cloud, gate pull requests, and publish useful reports.

You should already be comfortable with Swift, SwiftUI, the Observation framework, async/await, and the basics of SwiftData. We do not teach the language or how to lay out views; we teach how to test them.

What You Will Learn

- A practical testing mindset: what to test, what not to test, the test pyramid as a heuristic, and the properties of good tests.
- Swift Testing in depth: `@Test`, `#expect` and `#require`, suites, traits, tags, parameterized tests, async tests, and confirmations.
- XCTest: assertions, lifecycle, expectations, and when to keep it — plus a gradual migration strategy.

- Designing for testability: test doubles, dependency injection, testing view models built with Observation, and test-driven development on a real feature.
- Testing async and concurrent code, networking with `URLProtocol`, persistence with in-memory SwiftData, and hidden inputs such as dates, time zones, and locales.
- UI testing with XCUITest, page objects, accessibility identifiers, launch arguments, accessibility audits, and snapshot testing.
- Test plans and configurations, code coverage that means something, performance tests with `XCTMetric`, and how to diagnose and cure flaky tests.
- Continuous integration with `xcodebuild` and result bundles, GitHub Actions macOS runners, Xcode Cloud, fastlane scan, and test reports.

How This Book Is Organized

Part I — Foundations of Testing (Chapters 1–5) builds the mindset and the tools: Swift Testing from the first test to traits, tags, and parameterized tests, and XCTest with a migration strategy. **Part II — Designing for Testability** (Chapters 6–10) tests Fitlark's domain logic, introduces test doubles and dependency injection, tests Observation-based view models, and builds a feature test-first. **Part III — Async, Network, and Data** (Chapters 11–15) covers async and concurrent code, networking with `URLProtocol`, SwiftData persistence, and dates, time zones, and locales. **Part IV — UI and Visual Testing** (Chapters 16–20) covers XCUITest, page objects, launch configuration, snapshot tests, and accessibility audits. **Part V — Test Suite Quality** (Chapters 21–24) covers test plans, code coverage, performance tests, and flaky tests. **Part VI — Continuous Integration** (Chapters 25–28) runs everything from the command line, on GitHub Actions, and on Xcode Cloud, and ends with test reports and Fitlark's complete test strategy. The appendices provide a cheat sheet, templates, a readiness checklist, and a glossary.

Each chapter ends with **Try It Yourself** exercises and a **Key Takeaways** box for quick review.

The Running Example: Fitlark

Throughout the book we test **Fitlark**, a fictional workout logger. Fitlark users log workouts made of sets — an exercise, a number of reps, and a weight — and the app shows their weekly training volume against a goal, detects new personal records using an estimated one-rep max, suggests which plates to load on a barbell, runs a rest timer between sets, and syncs workouts to a (fictional) web API. Under the hood it has everything a real app has: pure domain logic, Observation-based view models, an actor that coordinates sync, a networking client, SwiftData persistence, and a SwiftUI interface with three tabs. We start with a single test for the volume of one set and finish with hundreds of tests organized in two test plans and running in CI on every pull request. The sample data is small enough to check by hand: five workouts over two weeks, with every volume, estimate, and record worked out in advance.

Versions and Conventions

Examples use the **Swift 6 language mode** with complete data-race checking, **Xcode 26-era tooling**, and an app deployment target of **iOS 18**. Where a feature needs iOS 26 or a recent version of Swift Testing, a Note says so and the code is marked with availability checks. Tools evolve quickly: menu labels, simulator names, command-line options, and CI runner images change between releases. When something is not exactly where the book says, check the current documentation for your version. Test output, build logs, and command-line results are shown as "output similar to" what you will see; exact wording and timings differ between versions and machines.

> **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 make tests flaky, slow, or misleading.

Menu commands are written in bold with arrows, such as **Product > Test**. Code, file names, and commands appear in `monospace`; every listing is captioned with its file path, such as `FitlarkTests/Domain/OneRepMaxTests.swift`, and long lines are wrapped to fit the page. The project layout and naming conventions are introduced in Chapter 1 and summarized in Appendix B.

Contents

Preface

PART I — FOUNDATIONS OF TESTING

1 Why We Test: The Testing Mindset, the Test Pyramid, and Fitlark

2 Your First Swift Tests: Test Targets, @Test, #expect, and Running Tests in Xcode

3 Swift Testing in Depth: #require, Errors, Suites, Traits, and Tags

4 Parameterized Tests and Test Organization

5 XCTest: Assertions, Lifecycle, Migration, and When to Keep It

PART II — DESIGNING FOR TESTABILITY

6 Testing Pure Domain Logic: Volume, One-Rep Max, and Personal Records

7 Test Doubles: Dummies, Stubs, Fakes, Spies, and Mocks

8 Dependency Injection: Protocols, Closures, Clocks, and the Environment

9 Testing View Models Built with Observation

10 Test-Driven Development on a Feature: The Plate Calculator

PART III — ASYNC, NETWORK, AND DATA

11 Testing Async Code: async Tests, Confirmations, and AsyncStream

12 Testing Concurrent Code: Actors, MainActor, Cancellation, and Time

13 Networking Tests with URLProtocol

14 Persistence Tests with In-Memory SwiftData

15 Dates, Time Zones, Locales, and Other Hidden Inputs

PART IV — UI AND VISUAL TESTING

16 UI Testing with XCUITest: Launching, Querying, and Interacting

17 Page Objects and Accessibility Identifiers

18 Launch Arguments, Environment, and Test Data for UI Tests

19 Snapshot Testing SwiftUI Views

20 Accessibility Audits and Testing Across Languages and Appearances

PART V — TEST SUITE QUALITY

21 Test Plans, Configurations, and Schemes

22 Code Coverage: Measuring What Matters

23 Performance Tests with XCTMetric

24 Flaky Tests: Diagnosis, Quarantine, and Cures

PART VI — CONTINUOUS INTEGRATION

25 Command-Line Testing with xcodebuild and Result Bundles

26 GitHub Actions on macOS Runners

27 Xcode Cloud and fastlane scan

28 Test Reports and Case Study: Fitlark's Complete Test Strategy

APPENDICES

A Testing Cheat Sheet

B Templates and Reference

C Test Suite and CI Readiness Checklist

D Glossary

PART I

Foundations of Testing

Why we test and what good tests look like, then the tools: Swift Testing from the first test to #require, suites, traits, tags, and parameterized tests, and XCTest with a clear view of when to keep it and how to migrate.

1 · Why We Test
2 · Your First Swift Tests
3 · Swift Testing in Depth
4 · Parameterized Tests and Organization
5 · XCTest and Migration

CHAPTER 1 · PART I — FOUNDATIONS OF TESTING

Why We Test: The Testing Mindset, the Test Pyramid, and Fitlark

Before we write a single `@Test`, we need a clear picture of what automated tests buy us, what they cost, and which kinds of tests belong where. This chapter builds that picture and introduces Fitlark, the workout logger whose test suite and CI pipeline we will build across the rest of the book.

A Small Change on a Friday Afternoon

Imagine you maintain a workout logging app. A user writes in: the weekly volume on the Progress screen looks wrong for people who train late on Sunday nights. You find the culprit quickly. The code that decides which week a workout belongs to uses the device calendar, and in some regions the device calendar starts the week on Sunday rather than Monday. The fix is a one-line change: build the calendar from the ISO 8601 identifier instead. You make the change, build, tap around in the simulator, and the Progress screen looks right.

Now the uncomfortable questions start. Did that one line change anything else? The personal record detector also compares dates. The weekly goal progress bar reads the same weekly volume. The list of workouts is sorted by date. The sync service uploads workouts oldest first. You could open each screen and check by eye, in English and in German, in two time zones, with and without sample data. Or you could press **Cmd+U**, wait a few seconds, and see a green check next to every test that encodes what those parts of the app are supposed to do.

That difference is the whole argument for automated testing. A good suite turns "I hope nothing broke" into "the behaviors we care about still hold." In this book we build such a suite for a realistic SwiftUI app and then make a CI pipeline run it on every pull request.

What Tests Buy Us

"Finding bugs" is only part of the story. The real value comes from several benefits that compound over the life of a codebase.

Confidence to change code

Code that nobody dares to touch rots. When a module has no tests, every refactor is a gamble, so developers add special cases around it instead of improving it. A suite that pins down the important behaviors lets you rename, restructure, and replace implementations while the tests tell you, within seconds, whether the observable behavior survived. This is the benefit you feel most on an app that lives for years: the third rewrite of the sync code is far less frightening than the first.

Executable documentation

A well-named test is a precise statement about the system. "A warm-up set contributes zero volume" or "an exercise with no earlier history produces no personal record, because the first session is a baseline" are rules that are easy to forget and hard to discover by reading the implementation. Prose documentation drifts out of date silently; a test that drifts out of date fails loudly. When a new teammate wants to know how the personal record rules work, the test file is often the best place to start reading.

Design feedback

Tests are the first client of your code. If a view model is painful to test because it creates its own database and calls `Date()` deep inside a method, the pain is information: the type has hidden dependencies. Passing them in makes the code easier to test, reuse, and reason about, which is exactly how Fitlark's view models are built (Part II).

Earlier, cheaper bug discovery

A bug caught by a unit test while you are still editing the file costs a minute; the same bug reported by a user costs a bad review and an expedited release. We will not attach made-up multipliers to those stages, but the direction is obvious: the closer a check runs to the moment the bug was written, the cheaper the fix.

What Tests Cost

Being honest about cost is part of the testing mindset. Teams that oversell testing end up with large, slow, brittle suites that developers learn to ignore, which is worse than a small suite everyone trusts.

- **Writing time.** A test takes time to write, especially the first tests for a component, when you also have to build test data and test doubles.
- **Maintenance.** Tests are code. They need to be read, updated when requirements change, and refactored when they grow repetitive.
- **Run time.** Every second a suite takes is multiplied by every developer and every CI run. A unit suite that takes several minutes stops being run locally.
- **Brittleness.** A test that checks *how* code does something instead of *what* it achieves breaks on every harmless refactor. Brittle tests train people to "fix the test" without thinking.
- **False confidence.** A green suite only says the checked behaviors hold. If the important behaviors are not checked, green means very little. Code coverage numbers can make this worse, as we will see in Chapter 22.
- **Flakiness.** A test that sometimes fails without any code change costs more than a missing test, because it erodes trust in all the others. Chapter 24 is devoted to this problem.

The goal is not "as many tests as possible" but the smallest suite that gives high confidence, runs fast enough to run constantly, and fails only when something real is wrong.

> **WARNING**
>
> A slow or flaky suite does not just waste time; it changes behavior. Once developers expect red builds that "go away on retry," they stop reading failures, and a genuine regression can be merged with a shrug. Treat suite speed and reliability as features with the same priority as the tests themselves.

Kinds of Tests

Test vocabulary varies between teams, and there is no official taxonomy. The following definitions are the ones we use in this book. What matters more than the labels is understanding what each kind of test exercises, how fast it runs, and what it can and cannot tell you.

Kind	What it exercises	Typical speed	Fitlark example
Unit	One type or function, in memory, with its dependencies replaced or absent	Milliseconds	`OneRepMax.estimate`, `PRDetector`, view models with a fake store
Integration	Several real pieces together, such as a store and SwiftData	Milliseconds to seconds	`SwiftDataWorkoutStore` with an in-memory container
UI	The running app, driven through the accessibility layer	Seconds per test	Log a workout and see the PR banner
Snapshot	A rendered view compared with a stored reference image	Fast to moderate	`WeeklyProgressView` in light and dark mode
Performance	Time, memory, or other metrics of a piece of code	Seconds (repeated runs)	Weekly volume over thousands of sets; app launch

Unit tests are the workhorses: so fast that you can run hundreds after every change. "Unit" need not mean one class; what matters is that the test avoids slow or uncontrollable things such as the network, the file system, the real clock, and the user interface. **Integration tests** check that real components fit together; only a test against real SwiftData can prove that set order survives a save and reload. **UI tests** drive the running app through accessibility and give end-to-end confidence at a high price in speed and diagnosability. **Snapshot tests** catch visual regressions no view model assertion would notice, at the cost of rendering differences between devices (Chapter 19). **Performance tests** measure rather than assert and are noisy on shared machines (Chapter 23).

The Test Pyramid, as a Heuristic

The classic test pyramid suggests a shape for a healthy suite: many fast unit tests at the bottom, fewer integration tests in the middle, and a small number of slow end-to-end UI tests at the top.

The classic test pyramid

```
                /\
               /  \        UI tests: few, slow, end-to-end
              /----\
             /      \      Integration + snapshot tests:
            /        \     some, real components together
           /----------\
          /            \   Unit tests: many, fast, isolated
         /______________\
```

The reasoning is about feedback. Unit tests are cheap to write and run, and they point directly at the broken code. UI tests cover more ground per test but are slow, and a failure often says only that "something on the Log screen" broke. Treat the pyramid as a heuristic, not a law. The "testing trophy" idea, popular in the web world, emphasizes integration tests instead; others describe a "honeycomb" for service-oriented systems. Each picture argues about where confidence is cheapest for a particular kind of software, and you do not need to pick a camp.

For Fitlark, a pyramid-ish shape fits because most of its logic is pure computation (volumes, one-rep max estimates, personal records, plate math) or view model state, all testable in memory in milliseconds. So Fitlark ends up with a broad base of unit tests, focused integration tests for persistence and networking, a few snapshot tests for the progress chart, and a handful of UI tests for the flows that matter most.

> **NOTE**
>
> The layers are not about the testing framework. A Swift Testing `@Test` can be a unit test or an integration test depending on what it touches. What decides the layer is the scope of the code under test and the dependencies it uses, not the macro in front of the function.

Properties of Good Tests

Whatever the layer, good tests share a few properties. If you remember one list from this chapter, make it this one.

- **Fast.** A unit test should run in milliseconds. Speed is what lets you run the suite after every change instead of once a day. Anything that sleeps, waits for a real timer, or talks to a server is not fast.
- **Isolated.** Each test sets up its own state and does not depend on other tests. It passes when run alone, in any order, and in parallel with the others. Swift Testing runs tests in parallel by default, so isolation is not optional.
- **Repeatable.** A test gives the same result on every run, on every machine, in every time zone and language. That means controlling hidden inputs: the current date, the calendar, the locale, random numbers, and generated identifiers.

- **Self-checking.** A test decides on its own whether it passed. No one should need to read a log or look at a screenshot to know the outcome.
- **Timely.** Tests are written close to the code they check, ideally before or just after it. Tests written months later tend to encode whatever the code happens to do, bugs included.
- **Readable.** A failing test should explain itself. A descriptive name, a clear arrange, act, assert structure, and good failure messages save the next person a debugging session.
- **Behavior-focused.** A test checks an observable outcome, not internal steps. That keeps it stable under refactoring.

You may recognize the first five as a variation on the well-known "FIRST" mnemonic. The names are less important than the habit of asking, for every test you write: would this still pass, for the same reason, on a colleague's machine at midnight in Tokyo with the phone set to German?

What Not to Test

Knowing what to leave out is as important as knowing what to include. A test that checks nothing useful still costs time to run and maintain.

- **The frameworks themselves.** We do not test that `Array.sorted` sorts or that SwiftUI renders a `Text`. Apple tests those. We test *our* use of them, for example that our store returns workouts newest first.
- **Trivial code.** A stored property with no logic, or a computed property that only forwards to another, rarely deserves its own test. It will be exercised by tests of the behavior that uses it.
- **Layout details in views.** Asserting that a stack has three children or that a padding equals 16 points makes tests brittle and catches few real bugs. Put logic in view models and test it there; use a few snapshot tests for visual regressions.
- **Private implementation steps.** If a test needs to reach into private state to verify how something happened, it is probably testing the wrong thing. Test the outcome through the public (or `internal`) surface.
- **Real external services.** Unit tests never call the real Fitlark API or the network at all. Network behavior is tested with stubbed responses (Chapter 13).

Each of these is a judgment call. A computed property that looks trivial but encodes a business rule, like `WorkoutSet.volume` ignoring warm-up sets, deserves a test precisely because the rule is easy to break.

Meet Fitlark

Fitlark is a fictional workout logger. Users record workouts made of sets (an exercise, a number of repetitions, a weight in kilograms, and a flag marking warm-up sets). The app computes training volume, tracks a weekly volume goal, estimates one-rep maxes, and celebrates personal records. It is

small enough to understand completely and rich enough to need every kind of test this book covers: pure domain logic, view models, async code, an actor, a network client, SwiftData persistence, date and locale handling, SwiftUI views, and UI flows.

We do not build Fitlark's user interface from scratch. Its production code is given, and you will see each file when we test it. Our job is to build the **test suite, the test plans, and the CI pipeline** around it, the same situation you face when you join a team with an existing app.

The app at a glance

Fitlark has a tab bar with three tabs. The **Log** tab shows a list titled "Workouts", newest first. Each row shows the workout name, its date, and its volume, like "1,987.5 kg". A "+" button opens the **Add Workout** sheet, where you enter a name, pick an exercise, type reps and weight, mark warm-ups, add sets, and save. If the new workout beats an earlier best, a banner announces it, for example "New personal record: Barbell Row". Above the list, a label summarizes the current week, such as "This week: 6,197.5 kg of 6,000 kg", with a progress bar toward the weekly goal. The **Records** tab lists the best estimated one-rep max per exercise. The **Progress** tab shows a bar per ISO week with the current week highlighted.

Fitlark's main screen (sketch)

```
+-------------------------------------+
| Workouts                          +  |
|  This week: 6,197.5 kg of 6,000 kg   |
|  [###################################]|
|-------------------------------------|
|  Upper C        Mar 6     2,010 kg   |
|  Lower B        Mar 4     2,200 kg   |
|  Upper B        Mar 2     1,987.5 kg |
|  Lower A        Feb 26    2,100 kg   |
|  Upper A        Feb 24    2,640 kg   |
|-------------------------------------|
|   Log        Records       Progress  |
+-------------------------------------+
```

The project layout

The Xcode project is `Fitlark.xcodeproj` with a scheme named `Fitlark`, bundle identifier `com.example.fitlark`, and a deployment target of iOS 18. Production code is organized by layer. Here is the part of the tree that matters for testing:

Fitlark project layout (excerpt)

```
Fitlark/
  Model/        Exercise, WorkoutSet, Workout, PersonalRecord, WeightUnit
  Domain/       OneRepMax, VolumeCalculator, PRDetector, PlateCalculator,
                WeightParser, Calendar+Fitlark
  Persistence/  WorkoutStore, WorkoutEntity, SwiftDataWorkoutStore
  Networking/   APIClient, APIError
  Sync/         WorkoutSyncService
```

```
  Features/     Log/, AddWorkout/, Rest/, Progress/, Records/
  App/          FitlarkApp, AppLaunchConfiguration, AccessibilityID
FitlarkTests/
  Support/      SampleData, Tags, Fixture, MockURLProtocol
  Fixtures/     exercises.json, sync-receipt.json
  Doubles/      InMemoryWorkoutStore, SpyWorkoutStore,
                FailingWorkoutStore, StubUploader
  Domain/  ViewModels/  Concurrency/  Networking/  Persistence/
  Snapshots/  Legacy/  Performance/
FitlarkUITests/
  Screens/      LogScreen, AddWorkoutScreen, RecordsScreen, ProgressScreen
  Support/      XCUIApplication+Fitlark
  LogWorkoutUITests, RecordsUITests, AccessibilityAuditUITests,
  LaunchPerformanceTests
```

The production layers map neatly onto the test pyramid. **Model** and **Domain** contain plain value types and pure functions: no I/O, no global state. They are the easiest code in the world to test, and Fitlark deliberately puts as much logic there as possible. **Features** contains view models marked `@MainActor` and `@Observable`, which receive their dependencies through initializers. **Persistence**, **Networking**, and **Sync** sit at the edges, behind protocols such as `WorkoutStore` and `WorkoutUploading`, so the rest of the app can be tested without a database or a server.

Three targets

The project has three targets, and you will meet each one in turn:

- `Fitlark` is the app itself.
- `FitlarkTests` is the unit test bundle. It contains mostly Swift Testing tests plus a few XCTest classes (a legacy example in Chapter 5 and performance tests in Chapter 23). Its files import the app with `@testable import Fitlark`, which exposes `internal` symbols to the tests.
- `FitlarkUITests` is the UI test bundle, written with XCUITest. It launches the app as a separate process and never imports the app's code directly.

Later, two test plans will select which of these run and how: `FitlarkUnit.xctestplan` for the fast suite that runs on every push, and `FitlarkFull.xctestplan`, which adds UI tests and runs in both English (US) and German (Germany). Chapter 21 builds both.

A first look at the code under test

Here is the smallest interesting piece of Fitlark, the model for a single set. Notice the rule in the doc comment: warm-up sets count as zero volume, because lifters do not consider them training load.

`Fitlark/Model/WorkoutSet.swift`

```
struct WorkoutSet: Hashable, Codable, Sendable {
    var exerciseID: String
    var reps: Int
    var weight: Double          // kilograms; 0 for an unloaded bodyweight set
    var isWarmup: Bool = false
```

```
    /// Training volume of this set. Warm-up sets count as 0.
    var volume: Double { isWarmup ? 0 : Double(reps) * weight }
}
```

And here, as a teaser, is a complete Swift Testing test for it. We will set up the test target and explain every part in Chapter 2, but it is worth seeing how little ceremony is involved:

A first test (preview)

```
@Test func workingSetVolume() {
    #expect(WorkoutSet(exerciseID: "bench-press", reps: 5, weight: 80).volume == 400)
}
```

Two lines of body: build a set of five reps at 80 kg, and expect its volume to be 400. If someone later "simplifies" `volume` and breaks the multiplication, this test fails and tells you exactly which values were involved. If someone removes the warm-up rule, a second test, which you will write in the next chapter, catches it.

> **TIP**
> When you join a project with no tests, do not start with the hardest code. Start where Fitlark starts: pure functions and value types in the domain layer. You get quick wins, you build the test target and sample data you will need later, and you learn the business rules by writing them down as tests.

Swift Testing and XCTest at a Glance

Apple platforms have two testing frameworks today. **XCTest** has been part of Xcode for many years. Test classes inherit from `XCTestCase`, test methods start with `test`, and assertions are functions such as `XCTAssertEqual`. **Swift Testing** ships with Xcode 16 and later and is open source. It uses macros (`@Test`, `@Suite`, `#expect`, `#require`), works naturally with Swift concurrency, and supports parameterized tests and traits. Both frameworks can live side by side in the same test target.

Topic	Swift Testing	XCTest
Declaring a test	`@Test` on any function	Method named `test...` in an `XCTestCase` subclass
Grouping	`@Suite`, usually a `struct`	The `XCTestCase` subclass
Assertions	`#expect`, `#require`	`XCTAssertEqual`, `XCTUnwrap`, and others
Setup	The suite's `init`	`setUp` / `tearDown`
Parallelism	In-process, parallel by default	Parallel across processes when enabled
Parameterized tests	Built in: `@Test(arguments:)`	Loops or separate methods

UI and performance tests	Not supported at the time of writing	Supported

In this book we write new unit and integration tests with Swift Testing, because its failure messages are better, its parameterized tests remove a lot of repetition, and its traits make it easy to tag, disable, and time-limit tests. We use XCTest where it is still required: UI tests with XCUITest and performance tests with `measure(metrics:)`. As of Xcode 26, Swift Testing cannot run either of those. Chapter 5 covers XCTest properly and gives you a migration strategy for older suites.

One rule applies from day one: never mix the frameworks inside a single test. An `XCTAssertEqual` inside a `@Test`, or a `#expect` inside an `XCTestCase` method, does not report to the framework running the test. Recent versions may flag this, but do not rely on it.

Versions Used in This Book

Testing tools have changed quickly in recent years, so it helps to be explicit about what we assume.

- **Swift 6 language mode** with a Swift 6.2-era toolchain. Strict concurrency checking is on, which means data races are compile-time errors. You will see `Sendable`, actors, and `@MainActor` in test code as well as production code.
- **Xcode 26-era tooling.** Screenshots are described in words rather than shown, and we note when a menu label or option might differ between Xcode versions.
- **Deployment target iOS 18.** Anything that requires iOS 26 is marked with `@available(iOS 26, *)` and called out.

Where a feature arrived in a specific recent release, such as Swift Testing attachments, we say so and suggest you check the current documentation. Where we are not certain which release introduced something, we say "recent versions."

New projects in Xcode 26 can enable a default actor isolation setting (part of "approachable concurrency") that makes code `@MainActor` unless stated otherwise. Fitlark does not rely on it: view models and the rest timer are annotated explicitly, and domain types are plain `Sendable` values, so the code compiles the same way with the setting on or off.

> **NOTE**
>
> Simulator names change with every Xcode release. Our command-line examples use `platform=iOS Simulator,name=iPhone 17`. If that device does not exist on your machine, run `xcrun simctl list devices available` and substitute any iPhone simulator from the list.

The Testing Mindset

Tools and techniques fill most of this book, but a few attitudes run underneath all of them. They are worth stating plainly at the start.

Tests are specifications first

When you write a test, you are writing down what the code should do, in a form a computer can check. That means the first question is not "how do I call this function?" but "what is the rule?" For Fitlark's personal records, the rule is subtle: a record must *strictly* beat the best estimate from *earlier* workouts, the first session for an exercise is a baseline and never a record, warm-up and bodyweight sets do not count, and at most one record is reported per exercise. Each clause becomes a test. If you cannot state the rule, you are not ready to test it, and you may have found a requirements gap.

Make the test fail first

A test that has never failed has not proven anything. Before trusting a new test, break the code on purpose (flip a comparison, remove the warm-up check) and watch the test turn red. It takes a few seconds and catches the surprisingly common case of a test that cannot fail, for example because it asserts on the wrong variable. Chapter 10 makes this the core of a workflow, test-driven development, but the habit is useful everywhere.

Control every input

Most "works on my machine" test failures come from inputs the test did not choose: the current date, the device time zone, the region's decimal separator, the order in which a dictionary iterates, or a random identifier. Fitlark's tests use a fixed "now" of `2026-03-06T20:00:00Z` (a Friday evening), a UTC calendar whose weeks start on Monday, explicit locales, and fixed UUIDs. When we show production code that reads the current date, it receives it through an injected closure, so tests can supply a constant.

Prefer real objects, then fakes, then mocks

Use the real thing when it is fast and deterministic; otherwise a **fake**, such as an in-memory store; and a **mock** that verifies exact calls only when the interaction itself is the behavior. Fakes lead to outcome-based tests that survive refactoring. Chapter 7 covers the full vocabulary.

Keep the suite honest

A suite degrades if nobody owns it. Fitlark's team has explicit policies, which are *our* choices rather than industry rules: a unit suite that runs in well under a minute with no network and no sleeps, an 80% line coverage floor in CI (while remembering that coverage is not quality), quarantine plus a bug link for flaky tests, and the full UI plan on `main` and nightly. Parts V and VI implement each one.

How a Test Run Fits Together

When you press **Cmd+U**, Xcode builds the app, builds the test bundle against it, and loads unit tests into a host process, by default the app running in the simulator. Swift Testing discovers `@Test` functions and runs them concurrently; XCTest discovers `test` methods. UI tests run in a separate

runner process that launches the app and drives it through accessibility, which is why they are slower and cannot call your code. On CI, `xcodebuild` does the same work and writes an `.xcresult` bundle. Part VI builds this pipeline:

From commit to green check (overview)

```
developer pushes
      |
      v
CI runner (macOS) --> xcodebuild build-for-testing
      |
      v
  FitlarkUnit plan ---------> unit + integration tests (parallel)
      |                          |
      |                          +--> coverage report --> 80% gate
      v
  result bundle (.xcresult) --> uploaded as a build artifact
      |
      v
  status check on the pull request: passed / failed
```

The Plan of the Book

- **Part I — Foundations of Testing** (Chapters 1 to 5): test targets, Swift Testing, parameterized tests, XCTest and migration.
- **Part II — Designing for Testability** (Chapters 6 to 10): domain tests, test doubles, dependency injection, view models, and TDD.
- **Part III — Async, Network, and Data** (Chapters 11 to 15): async code, actors, `URLProtocol`, SwiftData, and hidden inputs.
- **Part IV — UI and Visual Testing** (Chapters 16 to 20): XCUITest, page objects, launch configuration, snapshots, and accessibility.
- **Part V — Test Suite Quality** (Chapters 21 to 24): test plans, coverage, performance tests, and flaky tests.
- **Part VI — Continuous Integration** (Chapters 25 to 28): `xcodebuild`, GitHub Actions, Xcode Cloud and fastlane, reports, and the final case study.

Conventions Used in This Book

Code listings carry a caption with their path, such as `FitlarkTests/Domain/OneRepMaxTests.swift`. Production code is shown as given; only Chapter 10 grows production code on screen. Output blocks captioned "Test output (similar to)" show the shape of real output; exact wording varies between versions, and any durations are placeholders. Menu paths appear in bold, such as **Product > Test**; menu labels can differ slightly between Xcode versions. When an AI tool helps draft tests, we call it "[your AI assistant]": what matters is that you review its output and see each test fail before trusting it.