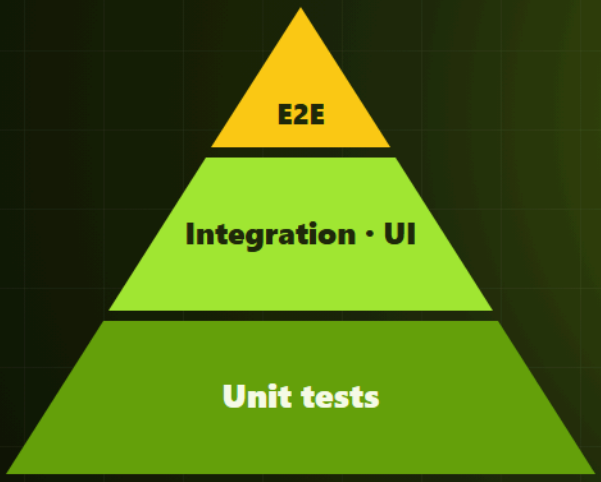
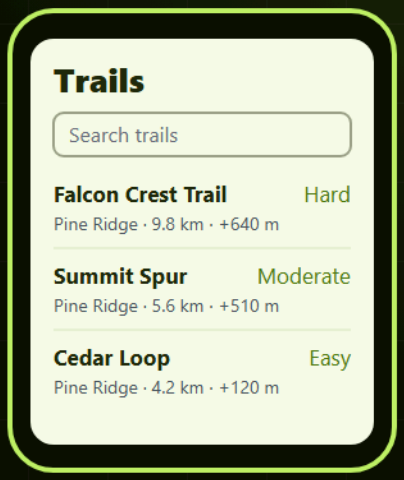
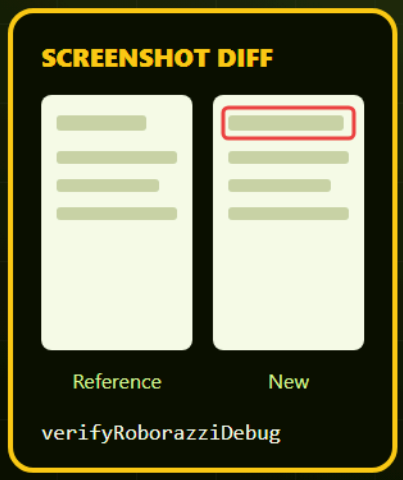



```
@Test
fun offline_showsMessage() =
  runTest {
    repo.refreshError =
      IOException("offline")
    vm.uiState.test {
      assertEquals(OFFLINE, ...)
    }
  }
```

UI Automator · Espresso · managed devices
Robolectric · Compose · Roborazzi · Room
JUnit · Turbine · MockK · fakes






```
✓ TripPlannerTest
✓ WaypointApiTest
✓ TrailDaoTest
✓ TripSyncWorkerTest
✓ TrailListViewModelTest
✓ PlanTripScreenTest
✓ NavigationTest

BUILD SUCCESSFUL
```

A COMPLETE, PRACTICAL GUIDE

Android Testing

Unit, integration, Compose UI, screenshot, and end-to-end tests that keep a real app shippable

JUnit 4 & 6 · MockK · Coroutines · Turbine · Room · MockWebServer
Robolectric · Compose · Roborazzi · Hilt · Espresso · Kover · CI

UNIT · INTEGRATION · UI · SCREENSHOT · E2E · CI

THENEXBOOKS

Android Testing

Unit, Integration, Compose UI, Screenshot, and End-to-End Tests That Keep a Real App Shippable
From your first JUnit test to a fast, trustworthy suite and a CI pipeline your team relies on

First Edition

THENEXBOOKS

Android Testing
Unit, Integration, Compose UI, Screenshot, and End-to-End Tests That Keep a Real App Shippable

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. Android, its tools, and the libraries discussed in this book change frequently; APIs, Gradle tasks, plugin options, and default behaviors can differ between versions. Always consult the official documentation for the versions you use.

Android, Android Studio, Jetpack Compose, Google Play, and Firebase are trademarks of Google LLC. Kotlin is a trademark of the Kotlin Foundation. Java is a registered trademark of Oracle and/or its affiliates. JUnit, Gradle, GitHub, and GitHub Actions are trademarks of their respective owners. All other product names, libraries, and services mentioned (including Robolectric, Roborazzi, Paparazzi, MockK, Turbine, Kover, JaCoCo, Retrofit, and OkHttp) are trademarks or names 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 Google LLC or any other company mentioned.

Northbeam Labs and Waypoint, the company and hiking app used as the running example in this book, are fictional, as are their team members, users, trails, trips, and all sample data. The backend address api.waypoint.example uses a reserved example domain, and email addresses use example.com.

Preface

Every Android team eventually reaches the same uncomfortable moment. The app works, users depend on it, and the code has grown large enough that nobody is quite sure what a change will break. Releases slow down because each one needs a long round of manual checking. Bugs that were fixed come back. A refactoring that everybody agrees is necessary never happens, because nobody dares to touch the code that "still works".

A good test suite is how teams get out of that situation. Not a suite that exists to satisfy a coverage number, and not a suite so slow and flaky that everyone learns to ignore it, but one that runs in minutes, fails only when something is really wrong, and tells you exactly what broke. Building that kind of suite for Android is a skill of its own: the platform mixes plain Kotlin logic with lifecycles, coroutines, databases, network calls, background work, and user interfaces that run on thousands of different devices.

This book teaches that skill step by step. We start with plain JUnit tests and work outward: test doubles, coroutines and Flow, ViewModels, Room, DataStore, networking, WorkManager, Robolectric, Compose UI tests, screenshot tests, Hilt, instrumented and end-to-end tests, Gradle Managed Devices, flaky tests, coverage, continuous integration, and legacy code. Every technique is applied to one realistic app, and every local test in the book was run for real on the toolchain described below.

Who This Book Is For

- **Android developers** who build apps with Kotlin and Jetpack Compose and have written few tests, or none, and want to learn how to test every layer properly.
- **Teams with a slow or flaky suite** who want faster feedback, fewer false alarms, and a clear strategy for what to test where.
- **Developers who inherited untested code** and need safe ways to add tests before they change it.
- **Tech leads and QA engineers** who are designing a test strategy and a CI pipeline for an Android app.

We assume you know Kotlin, Jetpack Compose, ViewModel, coroutines and Flow basics, and have seen Room, Retrofit, and Hilt. We do not re-teach them; we show how to test code that uses them.

What You Will Learn

- How the test pyramid applies to Android, and the difference between local tests and instrumented tests.
- How to write clear, fast unit tests with JUnit 4 and JUnit 6 Jupiter, and what each one supports on Android today.

- When to use fakes, stubs, spies, and mocks, and how to use MockK without over-mocking.
- How to test coroutines with virtual time, Flow and StateFlow with Turbine, and ViewModels with SavedStateHandle.
- How to test Room databases and migrations, DataStore, files, networking with MockWebServer, offline-first repositories, and WorkManager workers.
- How to run Android framework code on the JVM with Robolectric, and how to test Compose UIs through semantics.
- How to catch visual regressions with Roborazzi screenshot tests, and how Paparazzi compares.
- How to test accessibility, dark theme, locales, font scale, and screen sizes without an explosion of tests.
- How to replace dependencies with Hilt testing, and how to write instrumented, Espresso, and UI Automator tests that run on devices and Gradle Managed Devices.
- How to find and cure flaky tests, measure coverage with Kover and JaCoCo, design a CI test strategy with GitHub Actions, and bring legacy code under test.

How This Book Is Organized

Part I — Foundations (Chapters 1–5) explains why and what we test, sets up JUnit, explains the state of JUnit 4 and JUnit 6 on Android, and teaches the habits that make tests readable, using Waypoint's pure Kotlin business rules. **Part II — Test Doubles and Asynchronous Code** (Chapters 6–10) covers fakes and mocks, MockK, coroutines, Flow with Turbine, and ViewModels. **Part III — Data and Integration** (Chapters 11–15) tests Room, DataStore and files, networking with MockWebServer, offline-first repositories, and WorkManager. **Part IV — UI and Screenshot Testing** (Chapters 16–20) covers Robolectric, Compose UI testing, Roborazzi screenshot tests, and accessibility and configuration testing. **Part V — Hilt, Instrumented, and End-to-End Tests** (Chapters 21–24) covers Hilt testing, instrumented tests with Espresso and Compose, UI Automator, Gradle Managed Devices, and Test Orchestrator. **Part VI — Test Suite Quality and CI** (Chapters 25–28) covers flaky tests, coverage, continuous integration, and legacy code, and ends with Waypoint's complete test strategy. The appendices provide a cheat sheet, templates and reference tables, a production checklist for your test suite, and a glossary.

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

The Running Example: Waypoint

Throughout the book we test **Waypoint**, a hiking trip planner built by **Northbeam Labs**, a small fictional product studio. Waypoint lets hikers browse and search trails such as Cedar Loop and Falcon Crest Trail, plan trips with a date and a party size, record hikes, and sync everything when they are back online. We start from a modular starter project — domain rules, a Retrofit API, a Room cache, offline-first repositories, a WorkManager sync worker, and two Compose features — that already has a small test suite. Chapter by chapter we grow that suite into a complete testing strategy: hundreds of

lines of fast local tests, screenshot tests that catch visual regressions, Hilt-powered app tests, device tests for the paths that matter most, and a CI pipeline that runs them all at the right time. The sample data is small and fixed, so every expected value in a test can be checked by hand.

Versions and Conventions

The examples use the newest stable releases available at the time of writing: Android Gradle Plugin 9.4, Gradle 9.8, Kotlin 2.4, JDK 21, compileSdk 37 with targetSdk 36 and minSdk 26, the Compose BOM 2026.09.00, Hilt 2.60, Room 2.8, WorkManager 2.12, kotlinx-coroutines 1.11, Retrofit 3 with OkHttp 5, JUnit 4.13.2 and JUnit 6, Turbine, MockK, Robolectric 4.17, Roborazzi, and Kover. Libraries move quickly; where a version matters we say so, and where behavior may change we add a Note. Local tests (plain JVM, Robolectric, Compose under Robolectric, and Roborazzi) were run on this toolchain. Tests that need a device or emulator are shown with output "similar to" what you will see, because exact wording and timings vary.

> **TIP**
> A recommended practice or a technique that makes your tests faster, clearer, or more reliable.

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

> **WARNING**
> A common mistake, a source of flaky tests, or something that can hide real bugs.

Commands are shown for macOS and Linux as `./gradlew`; on Windows use `gradlew.bat` or `.\gradlew`. Kotlin code, Gradle files, and test output appear in `monospace`; long lines are wrapped to fit the page. File captions show paths relative to the project root, such as `feature/trails/src/test/.../TrailListViewModelTest.kt`.

Contents

Preface

PART I — FOUNDATIONS

1 Why Test Android Apps: The Testing Pyramid, Test Types, and Waypoint

2 Your First Local Tests: JUnit 4, Assertions, and Running Tests with Gradle

3 JUnit on Android Today: JUnit 4, JUnit 6 Jupiter, the Vintage Engine, and What Runs Where

4 Writing Good Tests: Naming, Arrange-Act-Assert, Test Data Builders, and Clear Failures

5 Testing Pure Kotlin: Domain Rules, Edge Cases, Time, and Parameterized Tests

PART II — TEST DOUBLES AND ASYNCHRONOUS CODE

6 Test Doubles: Fakes, Stubs, Spies, and Mocks — and When to Use Each

7 Mocking with MockK: Stubbing, Verification, Slots, and Pitfalls

8 Testing Coroutines: runTest, Test Dispatchers, and Virtual Time

9 Testing Flow and StateFlow with Turbine

10 Testing ViewModels: UI State, SavedStateHandle, Errors, and Events

PART III — DATA AND INTEGRATION

11 Testing Room: In-Memory Databases, DAOs, Converters, and Migrations

12 Testing DataStore, Files, and Other Local Storage

13 Networking Tests with MockWebServer: Retrofit, OkHttp, JSON Fixtures, and Failures

14 Integration Tests for Repositories and Offline-First Sync

15 Testing WorkManager: Workers, Factories, Retries, and Constraints

PART IV — UI AND SCREENSHOT TESTING

16 Robolectric: Android Framework Tests on the JVM

17 Compose UI Testing Fundamentals: Semantics, Finders, Assertions, and Actions

18 Compose UI Testing in Depth: Synchronization, Idling, Lists, and Navigation

19 Screenshot Testing with Roborazzi (and Where Paparazzi Fits)

20 Accessibility, Themes, Locales, Font Scale, and Screen Sizes

PART V — HILT, INSTRUMENTED, AND END-TO-END TESTS

21 Dependency Injection in Tests: Hilt Testing and Replacing Modules

22 Instrumented Tests: AndroidJUnitRunner, Espresso, and Compose on a Device

23 End-to-End Tests with UI Automator

24 Gradle Managed Devices, Test Orchestrator, and Sharding

PART VI — TEST SUITE QUALITY AND CI

25 Flaky Tests: Causes, Diagnosis, Quarantine, and Cures

26 Code Coverage with Kover and JaCoCo: Measuring What Matters

27 A CI Test Strategy: Gradle Tasks, GitHub Actions, Caching, and Reports

28 Testing Legacy Code and Case Study: Waypoint's Complete Test Strategy

APPENDICES

A Android Testing Cheat Sheet

B Templates and Reference

C Test Suite Production Checklist

D Glossary

PART I

Foundations

Why we test Android apps and where each kind of test runs, writing and running your first JUnit tests, the real state of JUnit 4 and JUnit 6 on Android, the habits that make tests readable and trustworthy, and thorough tests for Waypoint's pure Kotlin business rules.

1 · Why Test Android Apps

2 · Your First Local Tests

3 · JUnit on Android Today

4 · Writing Good Tests

5 · Testing Pure Kotlin

CHAPTER 1 · PART I — FOUNDATIONS

Why Test Android Apps: The Testing Pyramid, Test Types, and Waypoint

Tests are how a team keeps shipping without fear. In this chapter we look at what tests buy you, how Android tests are classified, where each kind runs, and we meet Waypoint, the hiking app whose test suite we build in the rest of the book.

What a Test Suite Is Really For

Ask ten Android developers why they write tests and you will hear "to catch bugs" from most of them. That answer is true but incomplete. A test suite catches some bugs, but its most valuable effect is less visible: it lets you **change code with confidence**. Every app that survives its first release spends the rest of its life being changed. New features arrive, libraries are upgraded, a screen is redesigned, the minimum SDK moves, a new team member rewrites a repository in a style they prefer. Each of those changes can break behavior that used to work, and nobody on the team remembers every rule the app is supposed to follow.

A good test suite is the team's memory of those rules, written in a form the computer can check in seconds. When a test says "a free user with three planned trips cannot plan a fourth one", that rule is no longer folklore that lives in the product manager's head. It is a check that runs on every pull request, long after the person who wrote the rule has moved on to something else.

We can split the value of tests into four practical benefits:

- **Confidence to ship.** A green build is evidence, not proof, that the app still does what it did yesterday. The more meaningful the tests, the stronger that evidence.
- **Confidence to refactor.** Without tests, cleaning up a messy class is a gamble. With tests that describe the class's behavior, you can restructure its internals and know within a minute whether you changed anything a user would notice.
- **Protection against regressions.** A bug that was fixed once should never come back. A test written for that bug guarantees it, because the fix and its proof travel together.
- **Design feedback.** Code that is hard to test is usually hard to use. If a class needs a real network, a real clock, and a running Activity just to check one calculation, the test is telling you something about its design.

Notice what is missing from that list: "to reach a coverage number" and "because the process says so". Tests that exist only to satisfy a metric tend to assert little and break often. We will come back to coverage in Chapter 26; for now, keep in mind that every test should earn its place by protecting behavior someone cares about.

The cost of finding a bug late

A bug costs more the later it is found. That claim is often decorated with precise multipliers, but we do not need invented numbers to see why it holds. Consider the same mistake caught at four different moments:

1. **While you type**, a unit test you just wrote fails. You read the message, fix the line, and move on. The bug cost you a minute and nobody else ever knew about it.
2. **In code review**, a teammate spots it. Now two people spend time on it, and the fix needs another round of review.
3. **In QA**, Theo, Waypoint's QA engineer, files a ticket. Someone has to reproduce it, find the cause in code they wrote last week, fix it, and ship a new build to QA.
4. **In production**, a hiker cannot save a trip on the morning of their hike. They leave a one-star review. The team writes a hotfix under pressure, waits for store review, and hopes users update.

Each step adds people, context switching, and delay. Mobile apps make the last step particularly painful, because you cannot redeploy a server and fix everyone at once: old versions stay installed on phones for months. That is why we push as much checking as possible toward the first step, where it is cheapest.

Two Ways to Classify Android Tests

Android tests are usually described along two separate axes. Mixing them up is the source of many confused discussions, so let us separate them clearly.

Axis one: where the test runs

The first axis is purely technical: on which machine does the test code execute?

- **Local tests** live in a module's `src/test` directory and run on your development machine's JVM (or on a CI server's JVM). They are compiled and run by Gradle tasks such as `test` or `testDebugUnitTest`. A local test can exercise pure Kotlin code directly, or it can exercise Android code through **Robolectric**, a library that simulates the Android framework on the JVM.
- **Instrumented tests** live in `src/androidTest`. Gradle packages them into a separate test APK, installs it next to your app on a device or emulator, and runs them there through `AndroidJUnitRunner`. They see the real Android framework, a real display, real system services, and the device's real SQLite version.

The trade-off is the obvious one. Local tests are fast, cheap to run, and need no hardware, but some of what they see is simulated. Instrumented tests are faithful, but they need a device or emulator, they are slower to start, and they are harder to keep stable.

Axis two: how much the test covers

The second axis describes the **scope** of a test: how much of the system it exercises at once. Android's own testing guidance uses the words small, medium, and big; most teams say unit, integration, and end-to-end. The exact boundaries are fuzzy, and that is fine.

Scope	Also called	Waypoint example
Small	Unit test	`TripPlanner` rejects a solo hiker on a HARD trail
Medium	Integration test	A repository writes to a real in-memory Room database and reads from a fake server
Big	End-to-end test	Launch the app on a device, tap a trail, plan a trip, see it saved

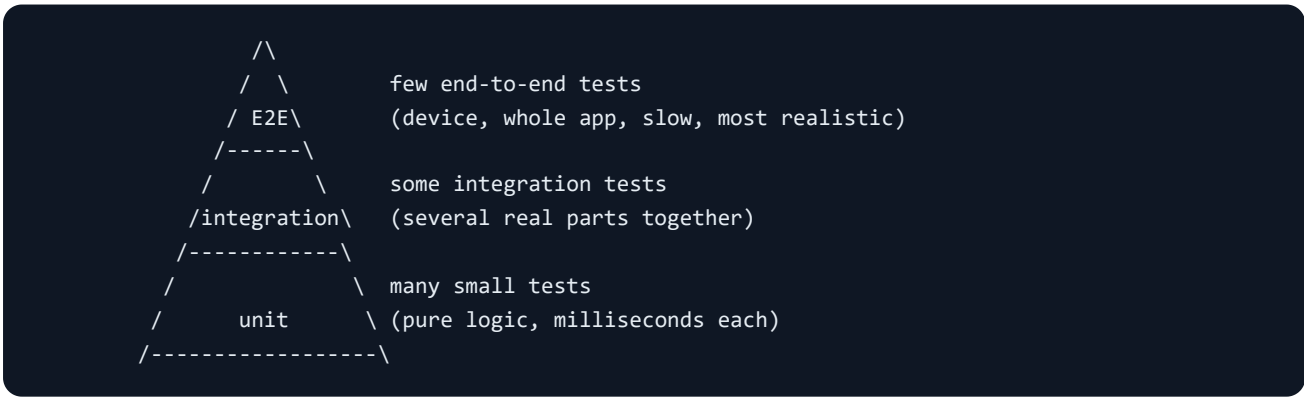
The two axes are independent. A small test usually runs locally, but a Compose screen test under Robolectric is a local test with a fairly wide scope. An instrumented test is often big, but a tiny instrumented test that checks one database query on a real device is still small in scope. When someone says "unit test", ask which axis they mean before you argue about it.

> **NOTE**
> Gradle and Android Studio still call local tests "unit tests" in several task and folder names (for example `testDebugUnitTest`). Those names describe *where* the tests run, not their scope. A test in `src/test` that spins up Room, a fake server, and a repository is an integration test that happens to run through a "unit test" task.

The Testing Pyramid and Its Critics

The most famous picture in software testing is the **test pyramid**: a wide base of many small, fast tests, a narrower middle of integration tests, and a small tip of end-to-end tests.

The test pyramid (a heuristic, not a law)



The reasoning behind the shape is about cost and feedback. Small tests are cheap to write, run in milliseconds, and point precisely at the line that broke. End-to-end tests are expensive to write, slow to run, more prone to random failures, and when they fail they often say little more than "something

in the trip flow is wrong". So the pyramid suggests: check most rules with small tests, check that the parts fit together with fewer integration tests, and keep a handful of end-to-end tests for the journeys that must never break.

The pyramid is a heuristic, and it has thoughtful critics. Some teams draw a **testing trophy**, with a fat middle of integration tests, because they found that many unit tests checked implementation details and broke on every refactor while proving little about real behavior. Others draw a **honeycomb** for service-heavy systems, again emphasizing tests that exercise real collaborations. These shapes are reactions to real problems: a suite of hundreds of tiny tests that mock everything can be both fast and nearly useless.

Our position in this book is pragmatic. The shapes disagree mostly about the middle layer, and the disagreement shrinks once you notice that on Android the middle layer has become cheap. Robolectric, in-memory Room databases, MockWebServer, and Compose tests on the JVM let you write integration-style tests that still run locally in seconds. So we aim for this:

- Plenty of small tests for **rules**: validation, calculations, formatting, search, state reduction.
- A solid layer of local integration tests for **wiring**: repositories with real Room, network code with a fake server, ViewModels with fakes, screens rendered under Robolectric.
- A thin layer of device tests for things that only a device can prove: the real app launching, system dialogs, the real framework on a real API level.

We never claim a "correct" ratio between the layers. The right mix depends on where your app's risk lives, and we will make that decision layer by layer for Waypoint.

Meet Northbeam Labs and Waypoint

Throughout the book we work on **Waypoint**, a hiking trip planner built by a small product studio called **Northbeam Labs**. The team is four people:

- **Lena** leads Android development and owns the architecture.
- **Sam** builds the backend (which, for us, is fictional: every server response in the book comes from a local fake).
- **Priya** owns product and design. Her rules, such as "free users can plan three trips", are exactly the kind of behavior tests must protect.
- **Theo** is the QA engineer. He keeps a list of flaky tests and has strong opinions about them.

Waypoint lets hikers browse trails, see an estimated hiking time for each one, and plan a trip on a chosen date with a party size. Trips are stored offline first and synced to the server in the background. The app already works; what it lacks is a test suite the team trusts. Lena's goal for the quarter is simple to state: "Every pull request tells us within minutes whether Waypoint still works."

The screens we will test

The starter app has two screens. The trail list shows every cached trail with a search field and a refresh button. Each row shows the trail's name, its region, distance, and elevation gain, an estimated hiking time, the difficulty, and the rating. Tapping a row opens the plan form for that trail.

Waypoint screens (ASCII sketch)

```
+---------------------------------+     +---------------------------------+
| Trails                  [Refresh]|     | Plan a trip                     |
| [ Search trails               ]  |     | Falcon Crest Trail              |
|---------------------------------|     | 9.8 km · about 3 h 02 min       |
| Falcon Crest Trail        Hard  |     |                                 |
| Pine Ridge · 9.8 km · +640 m    |     | [ Date (YYYY-MM-DD)           ] |
| About 3 h 02 min      Rating 4.8|     | [ Party size                  ] |
|---------------------------------|     |                                 |
| Summit Spur           Moderate  |     | Hard trails need at least 2     |
| Pine Ridge · 5.6 km · +510 m    |     | hikers.                         |
| About 1 h 58 min      Rating 4.7|     |                                 |
|---------------------------------|     |            [ Save trip ]        |
| ...                             |     |                                 |
+---------------------------------+     +---------------------------------+
```

The plan form validates what the user types: the date must be `YYYY-MM-DD` and not in the past, the party size must be between 1 and 12, hard trails need at least two hikers, and free users can have at most three planned trips. Those four rules alone produce a surprising number of interesting test cases, and Chapter 5 works through them exhaustively.

A Tour of the Starter Project

The book ships with a **starter project**, a Gradle build whose root project is named `waypoint-testing-starter`, with the application id `com.northbeam.waypoint`. It is a small but realistic multi-module app: Compose UI, ViewModels, Hilt, Room, Retrofit, DataStore, and WorkManager, built with the toolchain listed in the front matter (at the time of writing: AGP 9.4.1 with built-in Kotlin, Gradle 9.8.0, Kotlin 2.4.20, JDK 21). The starter already contains a first set of tests, so you can see every kind of test working before we take each one apart.

Module	What it contains	Kind of tests
`core:model`	Plain Kotlin data classes: `Trail`, `Trip`, `User`, `TrackPoint`; shared test fixtures	(provides fixtures)
`core:domain`	Business rules: `TrailSearch`, `NaismithEstimator`, `TripPlanner`, `TripStatsCalculator`, `WaypointFormat`	Pure JVM, JUnit Platform
`core:network`	Retrofit `WaypointApi`, DTOs, `AuthInterceptor`	MockWebServer
`core:database`	Room `WaypointDatabase`, DAOs, converters	Robolectric + in-memory Room

`core:data`	Offline-first repositories, DataStore preferences, sync worker	MockK, WorkManager testing
`core:testing`	Fakes and test rules shared by other modules	(test helper library)
`feature:trails`	Trail list ViewModel and Compose screen	Turbine, Compose UI, Roborazzi
`feature:trips`	Plan-trip ViewModel and Compose screen	Turbine, Compose UI, Roborazzi
`app`	`MainActivity`, navigation, Hilt application	Hilt + Robolectric; instrumented tests

The dependencies flow in one direction: features depend on `core:data`, which depends on the network, database, and domain modules, which depend on `core:model`. Nothing depends on a feature module except `app`. That layout is not only good architecture; it is good for testing, because the most important rules live in modules with no Android dependency at all, where tests are fastest.

What each layer needs from its tests

Different layers fail in different ways, so they need different tests. Here is how we will approach each one, with the chapter that covers it:

- **Domain rules** (`core:domain`) fail through wrong logic: an off-by-one boundary, a rounding mistake, a time-zone surprise. Small, exhaustive JVM tests catch these best (Chapters 2 to 5).
- **Asynchronous code** (ViewModels, flows, coroutines) fails through timing: a state that never arrives, a job that is never cancelled. Coroutine test tools and Turbine make time controllable (Chapters 8 to 10).
- **Data code** (Room, DataStore, Retrofit, WorkManager) fails at boundaries: a wrong SQL query, a migration that loses data, a JSON field that does not parse, a retry that never stops. Local integration tests with real libraries and fake servers catch these (Chapters 11 to 15).
- **UI** fails visibly: a button that stays disabled, a list that does not scroll, a layout that breaks at large font sizes. Compose tests and screenshot tests catch these, mostly on the JVM (Chapters 16 to 20).
- **The whole app** fails in integration with the device: dependency injection wiring, navigation, system dialogs. A few Hilt, instrumented, and end-to-end tests cover that (Chapters 21 to 24).

Running the starter's tests

Make a copy of the starter, open it in Android Studio or a terminal, and run all local tests from the project root. On macOS and Linux the wrapper script is `./gradlew`; on Windows use `gradlew.bat` (or `.\gradlew` in PowerShell). We use the macOS/Linux spelling in the book.

Terminal

```
$ ./gradlew test
```

The first run downloads nothing new if your Gradle cache is warm, but it compiles every module, so it takes a minute or two. The end of the output looks like this on our machine (task lines trimmed; your times will differ):

Terminal (end of output)

```
> Task :feature:trails:testDebugUnitTest
> Task :feature:trails:test
> Task :app:testDebugUnitTest
> Task :app:test

BUILD SUCCESSFUL in 30s
240 actionable tasks: 88 executed, 152 from cache
```

That is all Gradle prints when every test passes. There is no "42 tests passed" line, which surprises many developers the first time. Gradle reports test results in files instead: an HTML report per module and an XML file per test class. On our copy, the reports add up to **42 tests in 16 classes**, all green:

Module	Test classes	Tests
`core:domain`	`TrailSearchTest`, `NaismithEstimatorTest`, `TripStatsCalculatorTest`, `TripPlannerTest`	21
`core:network`	`WaypointApiTest`	2
`core:database`	`TrailDaoTest`	2
`core:data`	`OfflineFirstTrailRepositoryTest`, `TripSyncWorkerTest`, `UserPrefsRepositoryTest`	5
`feature:trails`	`TrailListViewModelTest`, `TrailListScreenTest`, `TrailListScreenshotTest`	6
`feature:trips`	`PlanTripViewModelTest`, `PlanTripScreenTest`, `PlanTripScreenshotTest`	5
`app`	`WaypointNavigationTest`	1

Two details in that output deserve attention now, because they will matter all through the book.

First, the line `152 from cache`. The starter enables Gradle's build cache and configuration cache in `gradle.properties`. When the inputs of a test task have not changed, Gradle may reuse the stored result instead of running the tests again. That is a big speed win on CI, but it means "the build passed" does not always mean "the tests just ran". When you deliberately want to rerun tests whose inputs did not change, add `--rerun` to a specific task (for example `./gradlew :core:domain:test --rerun`).

Second, in our setup the `test` task of an Android module runs only the **debug** unit tests (`testDebugUnitTest`). For pure Kotlin modules such as `core:domain`, `test` is the only test task there is. In later chapters we name `testDebugUnitTest` explicitly for Android modules, so there is never any doubt about which variant ran.

> **TIP**
>
> Open `core/domain/build/reports/tests/test/index.html` in a browser after a run. The HTML report lists every class and test, with durations, standard output, and full stack traces for failures. For Android modules the report lives under `build/reports/tests/testDebugUnitTest/`. Chapter 2 shows how to read it.

A first look at a real test

Before we discuss tooling, look at one of the starter's tests to see the shape we are aiming for. This is a ViewModel test from `feature:trails`. Do not worry about every API yet; Chapters 8 to 10 cover them in depth.

feature/trails/src/test/.../TrailListViewModelTest.kt (excerpt)

```
@Test
fun offline_keepsCachedTrails_andShowsMessage() = runTest {
    repository.setCachedTrails(listOf(SampleTrails.meadowRun))
    repository.refreshError = IOException("offline")

    val viewModel = TrailListViewModel(repository)

    viewModel.uiState.test {
        assertEquals(
            TrailListUiState.Content(
                listOf(SampleTrails.meadowRun),
                message = TrailListMessages.OFFLINE,
            ),
            awaitItem(),
        )
        cancelAndIgnoreRemainingEvents()
    }
}
```

Read it as a story. Given a cache that holds Meadow Run and a server that is unreachable, when the trail list opens, the user sees the cached trail and the message "Offline. Showing saved trails." The test uses a **fake** repository (`FakeTrailRepository` from `core:testing`) instead of a real network, so it runs on the plain JVM, needs no emulator, and finishes in a fraction of a second. It checks behavior a user would notice, not which private method was called. That is the style we will practice throughout the book.

The tests you cannot run on your laptop alone

The `app` module also has instrumented tests in `app/src/androidTest`: a Compose test that taps a trail on a real device, and a first draft of a UI Automator smoke test that launches the app from the home screen. That draft compiles but is not yet ready for a device: it is missing a piece of Hilt setup, a gap

Chapter 23 finds and fixes. Running them needs an emulator, a connected device, or a **Gradle Managed Device**, which the starter configures as `pixel6Api34`. You can always compile them without a device:

Terminal

```
$ ./gradlew :app:assembleDebugAndroidTest
```

Compiling device tests on every pull request is a useful cheap check: it catches a broken test API long before the nightly device run. Chapters 22 to 24 cover running them.

> **WARNING**
> Do not start a testing effort by writing end-to-end tests for every screen. They look impressive in a demo, but a suite built mostly of device tests is slow, needs emulators on CI, and tends to fail randomly. Teams in that situation often stop trusting red builds, and an ignored suite protects nothing. Build the fast layers first.

Waypoint's Testing Policy

Before writing any more tests, Lena and Theo agreed on a short policy. It is *our* choice for Waypoint, not an industry standard, and we explain the reasoning behind each item as the book goes on. We list it here so you know where the book is heading:

1. **Every bug fix starts with a failing test.** Before fixing a bug, write a test that reproduces it and watch it fail. Then fix the code and watch it pass. The test proves the fix and prevents the bug from returning.
2. **Fakes for our own interfaces, MockK only at boundaries.** When a class depends on one of our interfaces, such as `TrailRepository`, tests use a hand-written fake from `core:testing`. Mocking libraries are reserved for code we do not own or for checking an interaction explicitly (Chapters 6 and 7).
3. **No `Thread.sleep` in tests.** Time is injected: a `java.time.Clock` for dates and test dispatchers for coroutines. Tests control time; they never wait for it.
4. **Local tests on every pull request; instrumented tests nightly.** JVM, Robolectric, and screenshot tests run on every pull request. Device tests run on Gradle Managed Devices every night.
5. **Flaky tests are quarantined within one day** with an issue link, and fixed or deleted within two weeks (Chapter 25).
6. **Coverage is a signal, not a target.** The build enforces a floor of 60 % line coverage on the merged Kover report, chosen as our starting point; nobody is asked to chase a higher number for its own sake (Chapter 26).

Some of these items will feel strict, especially the first. In practice it is the item that pays back fastest. A bug report is a free test case: it describes exactly the input that broke the app. Turning it into a failing test takes minutes, and from then on the bug cannot return without the build turning red.

Tools We Use, and Why

The Android testing ecosystem has many overlapping libraries. We chose a small, mainstream set and use it consistently:

Need	Tool in this book	First chapter
Test framework	JUnit 4; JUnit Jupiter (JUnit 6) in `core:domain`	2, 3
Coroutines and flows	kotlinx-coroutines-test, Turbine	8, 9
Test doubles	Hand-written fakes, MockK	6, 7
Android on the JVM	Robolectric	11, 16
Network	OkHttp MockWebServer	13
Compose UI	Compose UI test APIs	17
Screenshots	Roborazzi	19
Dependency injection	Hilt testing	21
Device tests	AndroidJUnitRunner, Espresso, UI Automator	22, 23
Coverage	Kover, JaCoCo	26

Every version number in this book is the one we used at the time of writing. Libraries move quickly, so check each project's release notes when you upgrade; the ideas and most APIs carry over between recent versions.

There are good tools we mention but do not use as the main path: Paparazzi for screenshots, Kotest or Truth for assertions, Mockito for mocks. They are fine choices. We picked one tool per need so that every listing in the book fits together, and so that the starter builds with a small, predictable set of dependencies.

What This Book Does Not Cover

To keep the book focused, we leave a few topics out or only mention them:

- **Teaching Kotlin, Compose, Room, or Retrofit.** We assume you already build apps with them. When a test needs background, we give one short refresher and move on.
- **Performance benchmarking.** Macrobenchmark and Baseline Profiles measure speed rather than correctness. We mention them where they meet testing, and leave the details to dedicated material.

- **Manual QA processes and test management tools.** Theo's exploratory testing matters, but this book is about automated tests that live in the repository.
- **Security testing and penetration testing.** Those are specialist disciplines with their own tooling.
- **Every possible library.** We use one well-supported tool per job and explain the alternatives briefly.

Every chapter, on the other hand, is hands-on. You will see the production code under test, the test code, the command that runs it, and what you see when it passes or fails. Where a test can only run on a device, we say so clearly and describe the output you should expect.

How to Use This Book

The chapters build on each other, but each one starts from the same starter project and says which files it adds. You can read Part I straight through, then jump to the part that matches your current problem: Part II if your ViewModels are untested, Part III if your data layer is the risk, Part IV if UI regressions keep slipping through, Part V for device and end-to-end tests, Part VI if your suite exists but is slow or flaky.

We recommend a fresh copy of the starter for each chapter. Run the tests before you change anything, so that a failure you see later is always caused by your change. When a chapter quotes a test count, it counts a fresh starter plus that chapter's own additions (and any earlier test file the chapter says it extends), so the numbers in different chapters are snapshots, not a running total. When a chapter adds a test, type it in rather than pasting it; writing tests is a skill of the hands as much as of the head, and noticing what you have to import tells you a lot about how the test tools are organized.

Finally, break things on purpose. When you add a test, change the production code so that the test should fail, run it, and read the failure. A test you have never seen fail has not yet proven that it can detect anything.

Try It Yourself

1. Copy the starter, run `./gradlew test`, and open the HTML report of `core:domain`. Find the slowest test class. Then open the report for `feature:trails` and compare: why are its tests slower?
2. Run `./gradlew test` a second time without changing anything. Compare the summary line with the first run and explain what "from cache" and "up-to-date" mean for your tests.
3. Run `./gradlew :app:assembleDebugAndroidTest`. Locate the test APK it produced under `app/build/outputs/apk/androidTest/debug/`.
4. Pick one rule from the plan form (for example "hard trails need at least 2 hikers") and write down, in plain English, three test cases you would want for it. Keep the list; you will implement them in Chapter 5.
5. Classify each starter test class on both axes: where it runs (JVM, Robolectric, device) and its scope (small, medium, big).