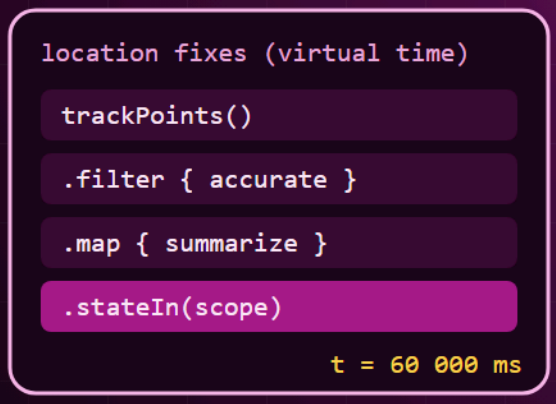







```
val uiState = query.debounce(300)
  .flatMapLatest { repo.searchTrails(it) }
  .stateIn(viewModelScope, WhileSubscribed(5_000), Loading)
// collectAsStateWithLifecycle()
```

A PRACTICAL DEEP DIVE

Kotlin Coroutines and Flow for Android

Structured Concurrency, Cancellation, StateFlow and SharedFlow, Operators, and Testing

suspend · launch · async · Dispatchers · Mutex
callbackFlow · stateIn · Compose · Room · Turbine

FROM CALLBACKS TO TESTED, LIFECYCLE-AWARE CODE

THENEXBOOKS

Kotlin Coroutines and Flow for Android

Structured Concurrency, Cancellation, StateFlow and SharedFlow, Operators, Lifecycle-Aware Collection, and Testing

From suspend functions to a tested, production-ready recording pipeline — built step by step in one real app

First Edition

THENEXBOOKS

Kotlin Coroutines and Flow for Android
Structured Concurrency, Cancellation, StateFlow and SharedFlow, Operators, Lifecycle-Aware Collection, and Testing

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. Kotlin, kotlinx.coroutines, the Android SDK, Jetpack libraries, and their tools change from release to release; APIs marked experimental or preview, default values, menu labels, and messages can differ in the version you use. Timings shown in this book come from simulated latencies measured on virtual time, not from benchmarks. Always consult the official documentation for the versions you target.

Android, Android Studio, Jetpack Compose, Google Play, and related names are trademarks of Google LLC. Kotlin is a trademark of the Kotlin Foundation. JetBrains and IntelliJ IDEA are trademarks of JetBrains s.r.o. Other product names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement. This book is independent and is not affiliated with, sponsored by, or endorsed by Google, JetBrains, or the Kotlin Foundation. All illustrations and interface sketches in this book are original.

Waypoint, the hiking app used as the running example in this book, is fictional, as are Northbeam Labs, its team members, its trails, regions, users, and all sample data and coordinates. The backend address `api.waypoint.example` uses the reserved `.example` domain and does not exist; the book's code runs against an in-memory fake and local test servers.

Preface

Almost every Android app written in Kotlin today uses coroutines somewhere. A ViewModel calls `viewModelScope.launch`, a repository exposes a `Flow`, a screen collects a `StateFlow`. The code often works on the first try — and then, months later, a screen keeps updating after it is gone, a search box fires a request for every keystroke, a background upload silently swallows its cancellation, a location listener is never unregistered, or a test passes on one machine and times out on another.

These are not exotic bugs. They come from using coroutines and Flow without a clear mental model of what is happening underneath: which coroutine runs this code, on which dispatcher, who its parent is, what happens to it when the user leaves the screen, and whether the stream it collects is cold or hot. Once that model is in place, coroutines become one of the most pleasant tools in the Android toolbox — and the bugs above become easy to spot in code review.

This book builds that model from the ground up and applies it to a realistic app. We start with suspend functions and the rules of structured concurrency, move through cancellation, exceptions, supervision, and shared state, then cover Flow in depth — operators, context, backpressure, StateFlow, SharedFlow, `stateIn`, `shareIn`, and `callbackFlow`. Finally we put it all to work across the Android stack and learn to test and debug it with confidence.

Who This Book Is For

- **Android developers who use coroutines by habit** and want to understand why their code works — and why it sometimes does not.
- **Developers moving from callbacks, RxJava, or LiveData** who need a clear path to suspend functions, Flow, and StateFlow.
- **Team members responsible for app quality** who want reliable, fast tests for asynchronous code and a checklist of common mistakes.
- **Anyone building location tracking, sync, search, or other long-running work** that must survive configuration changes and clean up after itself.

You should be comfortable with Kotlin syntax, lambdas, and data classes, and you should have built a simple Android app with a ViewModel and either Jetpack Compose or Views. No prior knowledge of coroutine internals or Flow operators is required.

What You Will Learn

- What a suspend function really is, how suspension differs from blocking, and how to turn one-shot callbacks into suspend functions safely.
- How `launch`, `async`, `coroutineScope`, and `withContext` work, how dispatchers map work to threads, and how to keep code main-safe.

- The rules of structured concurrency: jobs, parents and children, `viewModelScope`, `lifecycleScope`, and application-wide scopes.
- How cancellation, timeouts, exceptions, `CoroutineExceptionHandler`, and supervision really behave — and how to avoid swallowing cancellation.
- How to protect shared state with `Mutex`, `Semaphore`, atomics, and confinement, and how to build pipelines with channels.
- Flow from the basics to advanced operators: transforming, combining, `flatMapLatest`, `flowOn`, `catch`, `retry`, `buffer`, `conflate`, `debounce`, and more.
- Hot streams: `SharedFlow` for events, `StateFlow` for UI state, and `stateIn`/`shareIn` with `SharingStarted.WhileSubscribed`.
- How to bridge callback APIs such as location updates with `callbackFlow`, and how to integrate coroutines with ViewModels, Compose, Room, DataStore, Retrofit, WorkManager, and foreground services.
- How to test coroutines and flows with `kotlinx-coroutines-test`, virtual time, and Turbine, and how to debug them.

How This Book Is Organized

Part I — Coroutine Foundations (Chapters 1–5) covers why coroutines exist, suspend functions, coroutine builders, dispatchers, and structured concurrency. **Part II — Cancellation, Errors, and Coordination** (Chapters 6–10) covers cancellation and timeouts, exceptions, supervision and partial failure, shared mutable state, and channels. **Part III — Flow Fundamentals** (Chapters 11–15) covers cold streams, transforming and combining flows, flow context and error handling, and backpressure. **Part IV — Hot Flows and State** (Chapters 16–19) covers SharedFlow, StateFlow, `stateIn` and `shareIn`, and bridging callback APIs. **Part V — Coroutines in the Android Stack** (Chapters 20–24) applies everything to ViewModels, lifecycle-aware collection in Compose and Views, Room and DataStore, Retrofit, and background work. **Part VI — Testing, Debugging, and Production** (Chapters 25–28) covers testing coroutines and flows, debugging and common mistakes, and an end-to-end case study. The appendices provide a cheat sheet, templates and reference tables, a production checklist, and a glossary.

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

The Running Example: Waypoint

Throughout the book we build the coroutine core of **Waypoint**, a fictional hiking trip planner and tracker made by a small product studio called **Northbeam Labs**. Waypoint lets hikers browse and search trails such as Cedar Loop, Falcon Crest Trail, and Granite Steps, works offline from a local database, records a hike from a stream of location fixes, keeps running totals of distance, elevation gain, and time, reminds the hiker after three hours on the move, and uploads recorded points in

batches when the network allows. Those features exercise exactly the problems coroutines and Flow are built to solve: concurrent requests, cancellation when the user leaves, timeouts and retries, shared state, streams of values over time, callback-based platform APIs, and work that must outlive a screen.

The book comes with a starter project — a small multi-module Android app plus a plain-Kotlin playground module — so you can run every example. Because Waypoint's backend does not exist, the app talks to an in-memory fake backend with simulated network latency, and tests use virtual time. That means the timings you see in the book, such as "the two requests finish at 800 ms", are exact and repeatable on your machine.

Versions and Conventions

The code was written and tested with **kotlinx.coroutines 1.11.0**, **Kotlin 2.4.20**, and current stable Jetpack libraries at the time of writing, including Compose, Lifecycle, Room, DataStore, and WorkManager, with Android Gradle Plugin 9.4.1, a compile SDK of 37, a target SDK of 36, and a minimum SDK of 26. Some coroutine and Flow APIs are still marked experimental or preview; we point them out where we use them, and their status may have changed by the time you read this. When the exact release that introduced a feature matters and is uncertain, we say "recent versions".

Output produced on virtual time is shown exactly. Output that depends on real time or on which thread runs a coroutine, such as thread names, is shown as "something like" what you will see, because it varies from run to run.

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

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

> **WARNING**
> A common mistake or something that can cause leaks, crashes, lost updates, or flaky tests.

Code appears in `monospace` with a caption naming its module and file, such as `core/common/.../TripRecorder.kt`. Long lines are wrapped to fit the page. Class and function names are consistent throughout the book and summarized in Appendix B.

Contents

Preface

PART I — COROUTINE FOUNDATIONS

1 Why Coroutines: From Callbacks and Threads to Suspend Functions

2 Suspend Functions in Depth: Suspension Points, Continuations, and Bridging Callbacks

3 Coroutine Builders: launch, async, runBlocking, and coroutineScope

4 Dispatchers and Threads: Main, IO, Default, and withContext

5 Structured Concurrency: Jobs, Scopes, viewModelScope, and lifecycleScope

PART II — CANCELLATION, ERRORS, AND COORDINATION

6 Cancellation: Cooperative Checks, Timeouts, and Cleanup

7 Exceptions: Propagation, CoroutineExceptionHandler, and Error Strategies

8 Supervision and Partial Failure: SupervisorJob and supervisorScope

9 Shared Mutable State: Mutex, Semaphore, Atomics, and Confinement

10 Channels: Producers, Consumers, Fan-Out, and select

PART III — FLOW FUNDAMENTALS

11 Flow Basics: Cold Streams, Builders, and Terminal Operators

12 Transforming Flows: map, filter, transform, and Friends

13 Combining Flows: zip, combine, merge, and the flatMap Family

14 Flow Context and Errors: flowOn, catch, retry, and onCompletion

15 Backpressure: buffer, conflate, collectLatest, debounce, and sample

PART IV — HOT FLOWS AND STATE

16 Cold vs Hot: SharedFlow for Events and Broadcasts

17 StateFlow: Observable State for the UI Layer

18 stateIn and shareIn: Sharing Upstream Work with SharingStarted

19 callbackFlow and channelFlow: Bridging Location Updates and Other Callback APIs

PART V — COROUTINES IN THE ANDROID STACK

20 ViewModels and UI State: Loading, Errors, and One-Off Events

21 Collecting Safely: collectAsStateWithLifecycle, repeatOnLifecycle, and Compose Effects

22 Room and DataStore with Coroutines and Flow

23 Retrofit and Offline-First Repositories

24 Background Work: CoroutineWorker, Foreground Services, and Long-Running Recording

PART VI — TESTING, DEBUGGING, AND PRODUCTION

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

26 Testing Flows with Turbine

27 Debugging Coroutines and the Most Common Mistakes

28 Case Study: The Waypoint Recording Pipeline End to End

APPENDICES

A Coroutines and Flow Cheat Sheet

B Templates and Reference

C Production Checklist

D Glossary

PART I

Coroutine Foundations

Why callbacks and threads stopped scaling, what a suspend function really is, how to bridge one-shot callbacks, how launch, async, and coroutineScope build a tree of work, how dispatchers map that work to threads, and how structured concurrency ties every coroutine to a scope such as viewModelScope or lifecycleScope.

1 · Why Coroutines

2 · Suspend Functions in Depth

3 · Coroutine Builders

4 · Dispatchers and Threads

5 · Structured Concurrency

CHAPTER 1 · PART I — COROUTINE FOUNDATIONS

Why Coroutines: From Callbacks and Threads to Suspend Functions

Every Android app has to wait: for the network, for the disk, for the next location fix. This chapter shows why the two classic ways of waiting, callbacks and threads, become painful as an app grows, what a coroutine actually is, and how a few lines of suspending code load Waypoint's trail list and user profile in 800 ms instead of 1300 ms. We also meet the team, the app, and the starter project that every later chapter builds on.

The One Thread That Must Never Wait

An Android app starts with a single special thread: the **main thread**, also called the UI thread. It runs your Activity and Fragment callbacks, it runs Compose recomposition and layout, it delivers touch events, and it draws frames. All of that work shares one queue. While the main thread is busy with one task, nothing else on that queue can run: no drawing, no reaction to a tap, no animation step.

The trouble starts when a task has to *wait*. Reading a database file, decoding a large image, or asking a server for the list of trails can take anywhere from a few milliseconds to many seconds, and you cannot predict which. If the main thread performs such a call directly, it sits idle until the answer arrives, and the screen freezes. Users notice dropped frames long before anything crashes. If the main thread stays blocked long enough, the system shows the "Application Not Responding" dialog (an **ANR**) and offers to close the app. Android even refuses some kinds of waiting outright: performing a network request on the main thread throws an exception, and the StrictMode developer tool can flag disk access on the main thread during development.

So the rule every Android developer learns early is: **never block the main thread**. The interesting question is what to do instead. The work still has to happen, its result still has to reach the UI, and the UI is only allowed to be touched from the main thread. For most of Android's history there have been two answers: callbacks and background threads. Both work. Both also grow uncomfortable as soon as an app does more than one asynchronous thing at a time, and Waypoint does a lot of asynchronous things at once.

Meet Waypoint and Northbeam Labs

Throughout this book we work on **Waypoint**, a hiking trip planner and tracker built by a small, fictional product studio called **Northbeam Labs**. Lena Ortiz leads Android development, Sam Okafor owns the backend, Priya Natarajan handles product and design, and Theo Brandt is the QA

engineer who keeps finding the bugs we are about to learn to avoid. The app's package is `com.northbeam.waypoint`.

Waypoint lets hikers browse and search trails, plan a trip, and record a hike with location fixes that turn into running totals of distance, elevation gain, and duration. Recorded points are uploaded in batches when the network allows, and the app reminds the hiker after three hours on the move. The backend lives at `https://api.waypoint.example/v1/`, an address that uses the reserved `.example` domain and never answers. Instead, the book's code talks to `FakeWaypointBackend`, an in-memory fake with simulated network latency, so every example runs on your machine without a server.

The worked example that follows us through the book is Alex Rivera's planned hike: trip `trip-001` on **Cedar Loop** (`tr-001`), a 4.2 km easy loop in the Pine Ridge region, planned for 16 May 2026 with a party of two. Here are the eight sample trails the fake backend serves, which you will see again and again:

id	name	region	km / gain m	difficulty
tr-001	Cedar Loop	Pine Ridge	4.2 / 120	EASY
tr-002	Falcon Crest Trail	Pine Ridge	9.8 / 640	HARD
tr-003	Lakeshore Path	Mirror Lake	6.5 / 80	EASY
tr-004	Granite Steps	Mirror Lake	7.1 / 455	MODERATE
tr-005	Old Mill Ridge	Fernhollow	11.3 / 720	HARD
tr-006	Meadow Run	Fernhollow	3.4 / 45	EASY
tr-007	Summit Spur	Pine Ridge	5.6 / 510	MODERATE
tr-008	Brookside Way	Mirror Lake	8.0 / 210	MODERATE

The first feature we look at is tiny. When Alex opens the app, the home screen shows a greeting with the number of available trails: "8 trails for Alex Rivera". To build that text, the app needs two answers from the backend: the trail list (`GET trails`) and the signed-in user (`GET me`). In the fake backend those calls take a simulated **800 ms** and **500 ms**. Two network calls and one string: a perfect small problem to compare the three styles of asynchronous code.

Attempt One: Callbacks

The oldest answer to "do not wait on the main thread" is the callback. Instead of returning a result, a function accepts a function to call later, when the result is ready. Many Android and third-party APIs are still shaped this way, from location clients to older networking libraries. Here is a callback-style version of the two backend calls and the home-screen logic, written as code you can compile in the playground module:

Playground (Ch01Test.kt)

```
interface CallbackBackend {
    fun fetchTrails(onResult: (Result<List<Trail>>) -> Unit)
    fun fetchMe(onResult: (Result<User>) -> Unit)
}

fun loadHomeWithCallbacks(
    backend: CallbackBackend,
    onDone: (String) -> Unit,
    onError: (Throwable) -> Unit,
) {
    backend.fetchTrails { trailsResult ->
        trailsResult.onSuccess { trails ->
            backend.fetchMe { meResult ->
                meResult.onSuccess { me ->
                    onDone("${trails.size} trails for ${me.displayName}")
                }.onFailure(onError)
            }
        }.onFailure(onError)
    }
}
```

With a fake implementation that answers immediately, the test prints what you would expect:

Gradle log (STANDARD_OUT)

```
callbacks: 8 trails for Alex Rivera
```

It works, and for two calls it is still readable. Now look at what the shape of the code tells us. The logic "get the trails, then get the user, then build a string" is a straight sequence, but the code is a staircase: each step lives inside the previous step's lambda. Every additional step adds another level of indentation. Developers call this **callback hell**, and indentation is the least of its problems:

- **Error handling is repeated.** Each level must remember to forward failures. Forget one `onFailure`, and an error silently disappears; the screen spins forever.
- **There is no natural way to cancel.** If Alex leaves the screen while `fetchTrails` is in flight, the callback still fires later and tries to update a screen that no longer exists. To prevent that, every API needs its own cancel handle, and every caller must store and use it.
- **Threading is invisible.** On which thread does `onResult` run? The signature does not say. Some libraries call back on the main thread, others on a background thread. Touching the UI from the wrong one crashes the app.
- **Concurrency is hard to express.** The two calls are independent, so they could run at the same time. With callbacks, that means two counters or flags and a check in each callback to see whether the other one has finished. Possible, but easy to get wrong.
- **Exceptions do not flow.** A `try/catch` around `loadHomeWithCallbacks` catches nothing that happens inside the callbacks, because they run later, from a different stack.

Callbacks that deliver a stream of values add one more obligation. The starter project contains a callback-style location API shaped like the platform's location clients:

core/common/.../LocationSource.kt

```
fun interface LocationFixListener {
    fun onFix(fix: LocationFix)
}

interface LocationUpdatesClient {
    fun requestUpdates(intervalMs: Long, listener: LocationFixListener)
    fun removeUpdates(listener: LocationFixListener)
}
```

Whoever calls `requestUpdates` must remember to call `removeUpdates` with the same listener, on every exit path: when the hike ends, when the screen closes, when an error happens. Forget it once and the GPS keeps running, draining the battery and leaking whatever the listener captured. The compiler cannot help, because nothing ties the registration to the code that should end it. We will return to this exact API in Chapter 2, where we turn a one-shot use of it into a suspend function, and in Chapter 19, where we turn the repeating version into a `Flow` that unregisters itself.

Attempt Two: Threads and Executors

The other classic answer is to move the waiting to another thread. A background thread can call a blocking API, wait as long as it likes, and then post the result back to the main thread. Java's executors make this tidy: you submit tasks to a pool of threads, and the pool runs them. For a single request this is fine, and many apps did exactly this for years.

The cost appears when you look at what a thread *is*. A thread is an operating-system resource. It has its own stack of memory, the scheduler has to switch between threads, and a thread that is blocked waiting for a network answer is still occupying all of that while doing nothing useful. Thread pools exist precisely because threads are too expensive to create freely, but a pool has a fixed number of threads, so blocked tasks queue up behind each other.

We can see that queueing in a small real-time experiment in the playground. Eight tasks each "wait" for one second by calling `Thread.sleep`, on a pool of four threads. Then fifty thousand coroutines each wait for one second with `delay`, all started from a single thread:

playground/.../Ch01.kt

```
fun main() {
    // Threads: one blocked thread per waiting task.
    val pool = Executors.newFixedThreadPool(4)
    val threadTime = measureTime {
        val futures = (1..8).map { pool.submit { Thread.sleep(1_000) } }
        futures.forEach { it.get() }
    }
    pool.shutdown()
    println("8 sleeping tasks on 4 threads: ${threadTime.inWholeMilliseconds} ms")
```

```
    // Coroutines: suspended coroutines do not hold a thread.
    val done = AtomicInteger()
    val coroutineTime = measureTime {
        runBlocking {
            repeat(50_000) {
                launch {
                    delay(1_000)
                    done.incrementAndGet()
                }
            }
        }
    }
    println("${done.get()} coroutines on one thread: " +
        "${coroutineTime.inWholeMilliseconds} ms")
}
```

Running it with `:playground:run`, you should see something like this (the milliseconds depend on your machine and vary from run to run):

Console output (real time; varies)

```
8 sleeping tasks on 4 threads: 2053 ms
50000 coroutines on one thread: 1587 ms
```

Do not read this as a benchmark; the exact numbers are meaningless outside this machine and this run. Read it as a demonstration of a shape. The eight sleeping tasks took about two seconds because only four threads existed and each sleeping task held one of them for the whole second: the second batch of four had to wait for the first. The fifty thousand coroutines all waited *at the same time* on a single thread and finished a little after one second. A coroutine that is waiting in `delay` does not occupy a thread at all.

Threads also share the structural problems of callbacks: results must be posted back to the main thread by hand, cancelling means interrupting a thread and hoping the code checks the flag, errors hide inside a `Future` nobody inspects, and shared data needs locks.

> **NOTE**
>
> Older Android code may also use `AsyncTask`, which was deprecated years ago, or reactive libraries such as RxJava. Reactive streams solve many of the problems above with operators and disposables, and Flow, which we meet in Part III, covers similar ground. This book compares them only qualitatively; if your project still uses RxJava, the concepts in Parts III and IV will map onto what you know.

What a Coroutine Actually Is

A **coroutine** is a computation that can **suspend**: pause in the middle, give its thread back, and later **resume** where it left off, possibly on a different thread. That is the whole idea. A coroutine is not a thread. It is closer to a task: a piece of work with its own state, which a thread runs for a while, puts aside when the work has to wait, and picks up again when the wait is over.

You can picture a coroutine as a bookmark in a recipe. A cook (the thread) reads the recipe until it says "let the dough rest for an hour". A cook who blocks would stand next to the bowl for an hour. A cook who suspends puts a bookmark in the recipe, starts another recipe, and comes back when the timer rings. The bookmark remembers which step comes next and what is already in the bowl. In Kotlin terms, the bookmark is called a **continuation**, and the places where a recipe may say "wait" are **suspension points**. Chapter 2 opens the hood on both.

The language makes suspension visible with one keyword. A function marked `suspend` is allowed to suspend; it can only be called from a coroutine or from another suspend function. Everything else is ordinary Kotlin: `if`, `for`, `try/catch`, local variables, return values. The library **kotlinx.coroutines** (version 1.11.0 at the time of writing) supplies the rest: builders that start coroutines, dispatchers that decide which threads run them, `delay`, timeouts, channels, Flow, and test utilities.

Of all these, **structured concurrency** matters most: scopes and jobs organize coroutines into a tree, so a parent waits for its children and cancelling a parent cancels its children. It is the feature that fixes the "forgotten callback" problem at the root, and Chapter 5 is devoted to it.

Attempt Three: Suspend Functions

Here is the same home-screen logic written with suspend functions. `WaypointBackend`, the interface that both the fake and the real network backend implement, declares its calls as `suspend`:

core/common/.../WaypointBackend.kt

```
interface WaypointBackend {
    suspend fun fetchTrails(query: String = "", region: String? = null): List<Trail>
    suspend fun fetchTrail(id: String): Trail
    suspend fun fetchMe(): User
    suspend fun uploadPoints(tripId: String, points: List<TrackPoint>): Int
}
```

And the logic becomes a plain sequence:

playground/.../Ch01.kt

```
suspend fun loadHome(backend: WaypointBackend): String {
    val trails = backend.fetchTrails()
    val me = backend.fetchMe()
    return "${trails.size} trails for ${me.displayName}"
}
```

Read it top to bottom: get the trails, get the user, return a string. There is no nesting, no callback parameter, and no thread management. Each call to the backend is a suspension point. While the trail request is in flight, the coroutine that runs `loadHome` is suspended and its thread is free to do other work, such as drawing frames. When the answer arrives, the coroutine resumes on the next line with `trails` holding a real list. Errors work the way you already know: if `fetchTrails` throws, the exception propagates out of `loadHome`, and a `try/catch` around the call catches it.

How long does it take? To answer precisely, we run it in a test with `runTest` from **kotlinx-coroutines-test**:

Playground (Ch01Test.kt)

```
@Test
fun sequential() = runTest {
    val text = loadHome(FakeWaypointBackend())
    println("sequential: $text at $currentTime ms")
    assertEquals(1_300, currentTime)
}
```

Gradle log (STANDARD_OUT)

```
sequential: 8 trails for Alex Rivera at 1300 ms
```

The result is exactly **1300 ms**: 800 ms for the trails, then 500 ms for the user. And yet the test itself finished almost instantly. That is **virtual time**. `FakeWaypointBackend` simulates latency with `delay`, a suspending function, and under `runTest` a `delay` does not wait for the wall clock. The test scheduler simply advances a virtual clock to the moment the delay ends and resumes the coroutine. `currentTime` reads that virtual clock in milliseconds. Here is how the fake simulates the two calls:

core/common/.../WaypointBackend.kt

```
override suspend fun fetchTrails(query: String, region: String?): List<Trail> {
    trailsCalls++
    delay(latency.trailsMs)
    return trails.filter { t ->
        (query.isBlank() || t.name.contains(query.trim(), ignoreCase = true)) &&
            (region == null || t.region == region)
    }
}

override suspend fun fetchMe(): User {
    delay(latency.meMs)
    return me
}
```

Virtual time is why the timings in this book are exact and repeatable on your machine. Every number like "800 ms" in a test output is a simulated latency measured on the virtual clock, not a benchmark of real hardware. Chapter 25 explains how the test scheduler works in detail; for now, treat `runTest` as "run this coroutine, skip the waiting, and tell me how long it would have taken".

Doing Two Things at Once

The trails and the user do not depend on each other, so waiting 1300 ms is wasteful. With callbacks, running them concurrently meant flags and counters. With coroutines it means starting two child coroutines with `async` and waiting for both results:

playground/.../Ch01.kt

```
suspend fun loadHomeConcurrently(backend: WaypointBackend): String = coroutineScope {
    val trails = async { backend.fetchTrails() }
    val me = async { backend.fetchMe() }
    "${trails.await().size} trails for ${me.await().displayName}"
}
```

Gradle log (STANDARD_OUT)

```
concurrent: 8 trails for Alex Rivera at 800 ms
```

Now the screen is ready at **800 ms**, the duration of the slower call, because the 500 ms user request runs during the 800 ms trail request. `coroutineScope` creates a scope whose two children are the `async` blocks; it does not return until both are done, and if one of them fails, it cancels the other and rethrows the error. You get concurrency, error propagation, and cleanup from one block. Chapter 3 explains every piece of this snippet, including why the order of the two `await()` calls does not matter.

> **TIP**
> When you read coroutine code, ask the mental-model questions this book keeps repeating: *Which coroutine runs this line? On which dispatcher? Who is its parent? Where can it suspend? Is this stream cold or hot?* In `loadHomeConcurrently`, the answers are: two child coroutines run the requests, they inherit the caller's dispatcher, their parent is the `coroutineScope` block, they suspend inside `delay` (or a real network call), and there is no stream yet.

The Same Code in Real Time

Virtual time is great for tests, but it is fair to ask what happens on a real clock. The playground's `Hello.kt` runs the concurrent version inside `runBlocking`, which starts a coroutine and blocks the current thread until it finishes; that is acceptable in a command-line `main()` and never acceptable on the Android main thread. The playground's `run` task enables the coroutine **debug mode**, which appends the coroutine's identity to thread names. You should see something like:

Console output (real time; varies)

```
[main @coroutine#1] 8 trails for Alex Rivera in 833 ms
```

The elapsed time is a little above 800 ms, and it changes from run to run because the real clock includes scheduling and JVM warm-up. The thread name shows that coroutine number 1 ran on the `main` thread of the JVM process (not the Android main thread; this is a plain JVM program). We

return to debug mode and thread names in Chapter 4 and Chapter 27.

> **WARNING**
>
> `runBlocking` appears in this chapter only because `main()` in a JVM program needs a way to wait for a coroutine. In an Android app, calling it on the main thread blocks the UI exactly like a synchronous network call, and calling it inside a coroutine can deadlock. App code starts coroutines with scopes such as `viewModelScope`; tests use `runTest`.

The Cost of a Coroutine

Earlier we launched fifty thousand coroutines in real time. The same experiment under `runTest` shows the logical result without real-time noise:

Playground (Ch01Test.kt)

```
@Test
fun manyCoroutines() = runTest {
    var finished = 0
    coroutineScope {
        repeat(50_000) {
            launch {
                delay(1_000)
                finished++
            }
        }
    }
    println("$finished coroutines finished at $currentTime ms")
}
```

Gradle log (STANDARD_OUT)

```
50000 coroutines finished at 1000 ms
```

All fifty thousand waits overlapped, so the virtual clock moved exactly one second. The point of this experiment is not that coroutines are free; each one is an object with some state, and launching huge numbers of them still costs memory and scheduling work. The point is that a *waiting* coroutine does not hold a thread, so you can afford one coroutine per logical task instead of rationing a small pool of threads. Notice also that `finished++` on a plain `var` was safe here only because every coroutine ran on the test's single thread. Chapter 9 shows what happens to the same counter on a multi-threaded dispatcher.

Coroutines in an Android App

In a test, `runTest` starts the coroutine. In an app, the coroutine usually starts in a **scope** that is tied to something with a lifetime. The most common one is `viewModelScope`, provided by the Lifecycle libraries for every `ViewModel`. Here is the real `refresh()` function from Waypoint's `TrailsViewModel`:

feature/trails/.../TrailsViewModel.kt

```
fun refresh() {
    viewModelScope.launch {
        message.value = try {
            "Synced ${repository.refresh()} trails"
        } catch (e: CancellationException) {
            throw e
        } catch (e: Exception) {
            "Sync failed: ${e.message}"
        }
    }
}
```

A button click calls `refresh()`, which is an ordinary, non-suspending function. Inside, `launch` starts a coroutine in `viewModelScope`. That coroutine calls the suspending `repository.refresh()`, which downloads the trails and writes them into the local database. When it returns, the coroutine updates a message on the main thread. If the user leaves the screen for good and the `ViewModel` is cleared, `viewModelScope` is cancelled, and so is this coroutine, along with the network call it is waiting on. No callback fires on a dead screen, and nobody has to store a cancel handle.

You may wonder about the strange-looking `catch (e: CancellationException) { throw e }`. It exists because cancellation in coroutines is delivered as an exception, and the generic `catch (e: Exception)` below it would otherwise swallow it. That is one of the most common coroutine bugs, and Chapters 6 and 7 explain it carefully. For now, just notice that the code is honest about it.

The same building blocks scale up to the hard parts of Waypoint: a search that waits for a 300 ms pause in typing, a recording fed by a location fix every 10 seconds, running totals in a `StateFlow`, uploads in batches of 50 with retries, and a reminder after three hours on the move.

The Starter Project

The book ships with a starter project: a complete, multi-module Waypoint app whose Gradle root project is called `waypoint-kcf`. It builds, runs, and passes its tests as it is; the chapters read its code, extend it, and run experiments next to it. The modules are deliberately small, so each one is easy to read in one sitting:

Module	What it contains
`app`	Application class, `MainActivity`, navigation between the trails and recording screens
`core:model`	Plain Kotlin models, `SampleData`, `TrackMath`
`core:common`	Dispatchers, backend fake, location sources, `TripRecorder`, retry, uploader
`core:network`	Retrofit API and the network backend
`core:database`	Room database and DAOs

`core:data`	Repositories, DataStore settings, WorkManager sync, platform location
`feature:trails`	`TrailsViewModel` and the Compose trails screen
`feature:recording`	`RecordingViewModel` and the recording screen
`playground`	Plain JVM module for chapter experiments

The `playground` module deserves a special mention. It is a plain Kotlin/JVM module with no Android at all, which makes it the fastest place to try coroutine code. It depends on `core:common`, so `SampleData`, `FakeWaypointBackend`, `FakeLocationSource`, `TripRecorder`, and friends are all available. It contains a `Hello.kt` file with a `main()` function, a `HelloTest.kt` template for virtual-time tests, and a `CanonicalNumbersTest.kt` that reproduces many of the timings quoted in this book. To follow along with a chapter, create a file such as `Ch01.kt` or `Ch01Test.kt` next to them.

Running It

Open the starter's `waypoint` folder in a recent version of Android Studio and let Gradle sync. The project uses JDK 21, Kotlin 2.4.20, Android Gradle Plugin 9.4.1, and Gradle 9.8.0 at the time of writing, with a compile SDK of 37, a target SDK of 36, and a minimum SDK of 26. Android Studio can run any test from the green arrow in the editor gutter next to the class or function, and the run window shows its output.

From a terminal in the project folder, the Gradle wrapper does the same. On Windows, call it as `.\gradlew.bat`; on macOS and Linux, `./gradlew`:

Terminal (project folder)

```
.\gradlew.bat :playground:test --tests "*Ch01*"
.\gradlew.bat :playground:run -PmainClass=com.northbeam.waypoint.playground.Ch01Kt
.\gradlew.bat :core:common:test
.\gradlew.bat test assembleDebug
```

The first command runs only your chapter's tests. The second runs a `main()` function from a playground file; Kotlin compiles a top-level `main` in `Ch01.kt` into a class named `Ch01Kt`. The third runs the core tests, and the last runs every test and builds the debug app. The project's build prints standard output from tests, so every `println` inside a test shows up in the Gradle output under a heading like `Ch01Test > sequential STANDARD_OUT`. That is where every "Gradle log" block in this book comes from.

The starter's own suite has 40 tests in 16 classes. The Android tests run on the JVM with Robolectric, so no emulator is needed; when a chapter changes an app module, run that module's unit tests, for example `:feature:trails:testDebugUnitTest`. A few things, such as a real GPS fix or a foreground-service notification, need a device; for those we describe what you should see. Everything else, including every timing, comes from real runs.