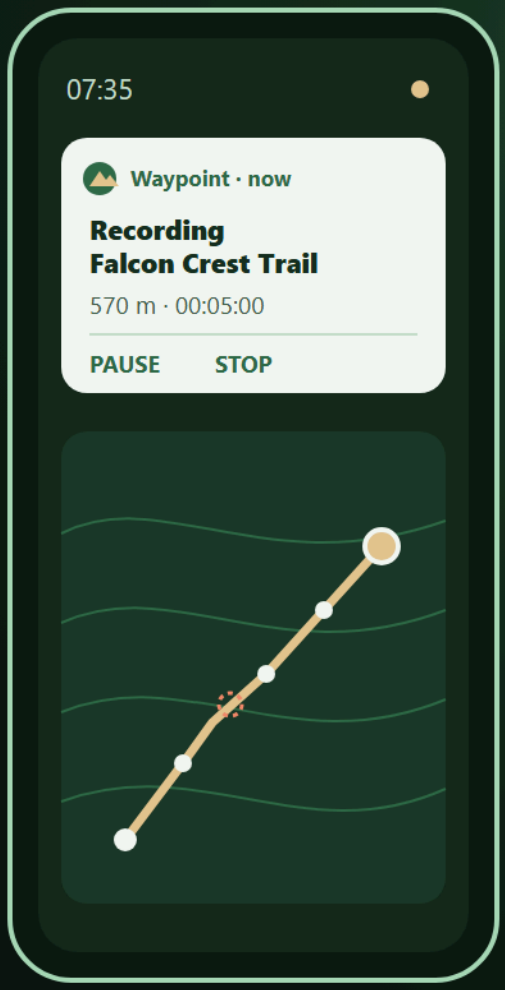



```
OneTimeWorkRequestBuilder
  <TripSyncWorker>()
  .setConstraints(NETWORK)
  .setBackoffCriteria(
      EXPONENTIAL, 30, SECONDS)
  .build()
// unique: "trip-sync", KEEP
```

FinishTripWorker
570.44 m · +16 m

→

TripSyncWorker
500 points / batch

needs: CONNECTED

Tomorrow: Granite Steps
Party of 3 · 7.1 km · Mirror Lake
`setAndAllowWhileIdle(18:00)`

A PRACTICAL, MODERN GUIDE

Android Background Work and System Integration

WorkManager, foreground services, notifications, location, and alarms that follow the platform's rules

Doze · Standby Buckets · Constraints · Retries · Location FGS
Channels · Exact Alarms · Receivers · Share Sheet · App Links · Glance

KOTLIN · JETPACK · ANDROID 8 TO 16

THENEXBOOKS

Android Background Work and System Integration

WorkManager, Foreground Services, Notifications, Location, Alarms, and Working with the Platform's Rules

From "why did my service die?" to a background layer that respects battery, privacy, and every Android version you support

First Edition

THENEXBOOKS

Android Background Work and System Integration
WorkManager, Foreground Services, Notifications, Location, Alarms, and Working with the Platform's Rules

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's background-execution rules, permissions, and Google Play policies change from release to release, and device manufacturers may add their own power-management behavior. Library versions named in this book were current at the time of writing. Always check the official behavior-changes documentation for the API levels you target, and test on real devices before you ship.

Android, Google Play, Google Play services, Jetpack Compose, Kotlin, and related marks are trademarks of their respective owners (Android and Google Play are trademarks of Google LLC; Kotlin is a trademark of the Kotlin Foundation). 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 LLC, JetBrains, or any other company.

Northbeam Labs, the Waypoint app, its team, trails, trips, users, and all sample data in this book are fictional. The API host `api.waypoint.example`, the web domain `waypoint.example`, and email addresses use reserved example domains; no real server is needed to run any example.

Preface

Every Android developer eventually ships a feature that works perfectly on their own phone and then fails quietly in the field. An upload never finishes. A reminder arrives an hour late, or not at all. A location tracker stops recording the moment the screen turns off. A notification never appears because a permission was never granted. The code is not obviously wrong; it simply ignores the rules the platform enforces on work that happens when the user is not looking at your app.

Those rules exist for good reasons. Every app that wakes the device, holds the network, or reads the GPS in the background spends the user's battery and touches their privacy. Over the years Android has added Doze, App Standby, background execution limits, foreground-service types, notification permissions, exact-alarm restrictions, and much more. Each change made phones last longer and made users safer, and each one broke apps that relied on the old behavior.

This book teaches you to work *with* those rules instead of fighting them. We start with how Android decides which processes live and which die, walk through the limits version by version, and then build a complete background layer for one realistic app: guaranteed work with WorkManager, a location foreground service that records a hike, notifications people actually want, alarms that respect the exact-alarm rules, receivers, sharing, deep links, shortcuts, and a home-screen widget. Every important piece is tested on the JVM with Robolectric, so you can prove it works before you ever plug in a device.

Who This Book Is For

- **Android developers** who can build screens with Jetpack Compose and write Kotlin with coroutines, and now need features that keep working when the app is in the background.
- **Developers maintaining older apps** that still rely on background services, repeating alarms, or implicit broadcasts and need to bring them up to modern targetSdk requirements.
- **Tech leads and reviewers** who want a clear mental model for choosing between WorkManager, foreground services, alarms, and doing nothing at all.
- **QA engineers and testers** who want to know which background behaviors can be tested automatically and how to reproduce Doze, standby buckets, and process death on a device.

What You Will Learn

- How processes, priorities, and component lifecycles determine what keeps running, and why threads and `GlobalScope` are not background work.
- The background execution limits from Android 8 through Android 16, Doze, App Standby Buckets, and battery restrictions — and what they mean for an app that targets a recent API level.
- WorkManager in depth: constraints, backoff and retries, unique and periodic work, chains, expedited and long-running work, progress, and Hilt-injected workers for offline-first sync.

- Foreground services the modern way: service types, permissions, the Android 14+ requirements, and a complete location-recording service built on the Fused Location Provider.
- Location permissions done right: precise versus approximate, one-time grants, and why many apps never need background location.
- Notifications that users keep enabled: channels, the Android 13 permission, actions, progress, groups, and deep links from a tap.
- Alarms and the exact-alarm rules, broadcast receivers that survive reboots, and handing work off safely to WorkManager.
- Integrating with the rest of the system: intents, the share sheet, FileProvider, deep links and Android App Links, app shortcuts, and Glance widgets.
- Testing and debugging background work: work-testing, Robolectric, Hilt testing, adb and `dumpsys`, and the Background Task Inspector.

How This Book Is Organized

Part I — How Android Runs Your App (Chapters 1–5) explains processes and lifecycles, background execution limits across versions, Doze and standby buckets, and how to choose the right tool. **Part II — WorkManager** (Chapters 6–10) covers workers, constraints, retries, unique, periodic, chained, expedited, and long-running work, and builds offline-first sync. **Part III — Foreground Services and Location** (Chapters 11–15) covers foreground-service types and rules, location permissions, the Fused Location Provider, a complete trip-recording service, and battery-aware recording that survives process death. **Part IV — Notifications and Alarms** (Chapters 16–19) covers channels and permissions, rich notifications, alarms, and broadcast receivers. **Part V — Integrating with the System** (Chapters 20–24) covers intents and activity results, sharing, deep links and App Links, app shortcuts, and Glance widgets. **Part VI — Testing, Debugging, and Shipping** (Chapters 25–28) covers automated tests, on-device debugging, Play policies and platform changes, and a complete case study. The appendices provide a cheat sheet, templates and reference tables, a production checklist, and a glossary.

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

The Running Example: Waypoint

Throughout the book we work on **Waypoint**, a hiking trip planner and tracker made by **Northbeam Labs**, a small fictional product studio. Lena Ortiz leads Android development, Sam Okafor owns the backend, Priya Natarajan shapes the product, and Theo Brandt keeps everyone honest in QA. Hikers browse trails such as Cedar Loop and Falcon Crest Trail, plan trips, record their hikes with GPS, and sync everything when they are back in coverage. That makes Waypoint a perfect stress test for

background work: a recording must survive a three-hour hike with the screen off, uploads must wait for a network that may not exist on the mountain, reminders must arrive the evening before a trip, and a safety notification must appear after three hours of moving.

The book's sample project contains Waypoint's complete background layer in three modules, with tests that run on your computer without an emulator. The sample data is small enough to check by hand: a five-minute track whose distance you can compute yourself, three trips, and eight trails. Waypoint also appears in the other books of the thenexbooks Android series, but this book is completely self-contained.

Versions and Conventions

The examples use Kotlin 2.4, Jetpack Compose (BOM 2026.09.00), WorkManager 2.12, Hilt, Room, Google Play services location 21.4, and Glance 1.2, built with Android Gradle Plugin 9.4 and Gradle 9.8 on JDK 21. The app compiles against API level 37, targets API level 36 (Android 16), and supports devices from API level 26 (Android 8.0) — so every rule in Chapter 3 applies to it. These were current stable versions at the time of writing; newer versions may differ in details. Where behavior depends on the Android version or on your targetSdk, the text says so. Features that were still in preview at the time of writing are labeled as such.

Some things cannot be shown by an automated test: a real GPS fix, a device entering Doze, the permission dialog, or a notification on a lock screen. When the book shows output from a device or from `adb`, it is introduced as "you should see something like" — your exact output will vary by device and version.

> **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 drain batteries, lose data, crash your app, or break a store policy.

Code appears in `monospace`; long lines are wrapped to fit the page, and import statements are omitted unless they matter. File captions show the path inside the sample project, such as `app/src/main/kotlin/com/northbeam/waypoint/work/WorkScheduler.kt`. Canonical names — `TripSyncWorker`, `TripRecordingService`, `WaypointNotifications`, `TripReminderScheduler` — are introduced in Chapter 1 and summarized in Appendix B.

Contents

Preface

PART I — HOW ANDROID RUNS YOUR APP

1 Meet Waypoint: The Background Problems a Hiking App Must Solve

2 Processes, Priorities, and Lifecycles: How Android Decides What Lives

3 Background Execution Limits from Android 8 to Android 16

4 Doze, App Standby Buckets, and Battery Restrictions

5 Choosing the Right Tool: WorkManager, Foreground Services, Alarms, or Nothing

PART II — WORKMANAGER

6 WorkManager Fundamentals: Workers, Requests, and Results

7 Constraints, Backoff, and Retries

8 Unique Work, Periodic Work, and Chains

9 Expedited and Long-Running Work: Progress and setForeground

10 Offline-First Sync with Hilt Workers

PART III — FOREGROUND SERVICES AND LOCATION

11 Foreground Services Today: Types, Permissions, and the Android 14+ Rules

12 Location Permissions: Precise, Approximate, and Background

13 The Fused Location Provider: Requests, Accuracy, and Updates

14 Building TripRecordingService: Recording a Hike End to End

15 Battery-Aware Recording and Surviving Process Death

PART IV — NOTIFICATIONS AND ALARMS

16 Notification Channels and the Android 13 Permission

17 Rich Notifications: Actions, Progress, Groups, and Tapping Through

18 Alarms: Inexact, Exact, and the Exact Alarm Restrictions

19 Broadcast Receivers: Boot, System Events, and Your Own

PART V — INTEGRATING WITH THE SYSTEM

20 Intents and Activity Results: Talking to Other Apps

21 Sharing: The Share Sheet, FileProvider, and Receiving Shares

22 Deep Links and Android App Links

23 App Shortcuts: Static, Dynamic, and Pinned

24 Home-Screen Widgets with Glance

PART VI — TESTING, DEBUGGING, AND SHIPPING

25 Testing Background Work: Workers, Services, Receivers, and Notifications

26 Debugging Background Work: adb, dumpsys, and the Background Task Inspector

27 Play Policies, Permissions, and Keeping Up with Platform Changes

28 Case Study: Waypoint's Background Layer from Plan to Upload

APPENDICES

A Background Work Cheat Sheet

B Templates and Reference

C Production Checklist

D Glossary

PART I

How Android Runs Your App

Before writing a single worker, we learn the rules: how Android ranks and kills processes, how background execution limits evolved from Android 8 to Android 16, what Doze and App Standby Buckets do to your work, and how to pick the right tool for every Waypoint feature.

1 · Meet Waypoint

2 · Processes, Priorities, and Lifecycles

3 · Background Execution Limits

4 · Doze, Standby Buckets, and Battery

5 · Choosing the Right Tool

CHAPTER 1 · PART I — HOW ANDROID RUNS YOUR APP

Meet Waypoint: The Background Problems a Hiking App Must Solve

A hiking app looks simple on the screen and hard everywhere else: it has to keep working while the phone sits in a backpack, reconnect when the trail ends, and speak up at exactly the right moment. In this chapter we meet Waypoint, list the background jobs it must do, explain why Android makes those jobs deliberately hard, and get the sample project building on your machine.

"It Works on My Phone"

Every Android team has a bug tracker full of reports that nobody can reproduce at a desk. At Northbeam Labs, the small product studio that builds Waypoint, the list looked like this the week before the team decided to rebuild the app's background layer:

- Theo Brandt (QA) filed: "Recorded a three-hour hike, screen off the whole time. The map shows a straight line from the trailhead to the summit and nothing after that."
- A beta tester wrote: "My trip from Saturday never showed up on the website. The app said 'Saved'."
- Priya Natarajan (product and design) noticed that the "Your hike is tomorrow" reminder arrived on her phone at 23:40 instead of early evening, and on one test phone it never arrived at all.
- Sam Okafor (backend) saw the same trip uploaded twice, a few seconds apart, with slightly different point counts.
- Lena Ortiz (Android lead) reproduced a crash on a newer test device when an experimental "auto-start at the trailhead" feature tried to begin recording while another app was in front: a stack trace mentioning a foreground service that was "not allowed".

None of these bugs showed up with the phone plugged into a laptop, screen on, debugger attached. That is the defining feature of background bugs: the conditions that trigger them are exactly the ones you remove when you sit down to debug. A charging phone does not enter deep power-saving states, and an app you just launched from Android Studio is the most important app on the device. This book is about the other situation: the phone is in a pocket, the screen is off, the network comes and goes, and the user has not looked at your app for an hour.

What "Background Work" Means for Waypoint

"Background work" is a loose phrase. For some developers it means "anything off the main thread". For others it means "anything that runs when the app is not on screen". In this book we use the second, stricter meaning: background work is any work your app does, or promises to do, **when the user is not actively looking at it**. Moving a database query to a background thread while the user watches a loading spinner is not background work in this sense; it is ordinary asynchronous code, and the Kotlin Coroutines and Flow for Android book in this series covers it well. What we care about is the work that has to survive the user pressing Home, turning off the screen, or walking out of network coverage.

Waypoint is a hiking trip planner and tracker. You browse trails, plan a trip on a date with a party size, record the hike while you walk, and look back at a summary of distance, duration and elevation gain. Out of that simple feature list come six background jobs that the rest of the book builds, one by one:

1. **Record a three-hour hike with the screen off.** The phone samples its location every few seconds, the app stores every fix, and the distance keeps growing even though the user never looks at the screen. This is user-visible, ongoing work: the user started it and expects it to continue until they stop it.
2. **Upload trips when the phone is back online.** Waypoint is offline-first: every trip and every track point lands in a local Room database first. Later, when there is a network connection, the app uploads whatever is pending. The user does not need to watch; they only need it to happen eventually and exactly once.
3. **Remind the user the evening before a planned trip.** "Tomorrow: Granite Steps" should appear around 18:00 the day before, even if the user has not opened the app for two weeks and the phone was rebooted in between.
4. **Alert after three hours of moving.** A safety nudge, "You've been moving for 3 hours", posted once per recording. It is small but time-sensitive, and it must work while the phone is in a pocket.
5. **Share a hike.** Send a short summary to a friend through the system share sheet, or export a GPX file another app can open. This is a hand-off to another app, which brings its own rules about permissions and file access.
6. **Open Waypoint from outside the app.** A link a friend sent, a launcher shortcut that starts a hike in one tap, or a home-screen widget that shows the next planned trip. Here the "background" part is the system calling into your app when the user has not opened it.

The table below adds the second half of each story: what the user expects, and the specific way it fails when the code ignores the platform's rules. Every failure in the right-hand column is one we will explain and fix.

Job	What the user expects	Typical failure	Chapters
Record a hike	Continuous track, live distance	Updates stop with the screen off; the process is killed	11–15

Upload trips	Data appears online eventually, once	Upload lost on process death, or sent twice	6–10
Evening reminder	Notification around 18:00 the day before	Late, missing after reboot, or blocked	16–19
Three-hour alert	One timely safety nudge	Never shown: no channel or no permission	14, 16, 17
Share a hike	Friend gets text or a file	Crash on a `file://` URI; receiver cannot read the file	20, 21
Links, shortcuts, widget	One tap to the right screen	Link opens the browser; widget never refreshes	22–24

Notice that the jobs are very different from each other. Recording is long, continuous and user-visible. Uploading is deferrable but must be guaranteed. The reminder is tied to a wall-clock time. The share is an immediate hand-off. Android has a different tool for each shape, and much of the pain in background code comes from using the wrong one. Chapter 5 turns this into a decision guide.

Why Background Work Is Hard on Android

It is tempting to read Android's background rules as a list of obstacles invented to annoy developers. They are easier to follow once you see the problems they solve. There are three, and each one shows up in Waypoint.

Battery: the phone is a shared, finite resource

A phone runs dozens of installed apps on one battery. Each app, seen on its own, wants "just a little" background time: a quick sync, a periodic check, a location fix. Add them up and the device never gets to sleep. The radio stays powered, the CPU keeps waking, and the user blames the phone. Over many releases Android has moved from trusting each app to manage itself toward the system deciding when background work runs, batching it together, and postponing it while the device is idle. Doze, App Standby, background execution limits and job quotas all come from this one idea: *the system, not the app, owns the schedule*. Chapters 3 and 4 walk through these mechanisms in detail.

For Waypoint, battery is not an abstract concern. A hiker relies on the phone for maps and emergencies. An app that drains the battery on a long trail is not just annoying; it can be a safety problem. That is why recording will adapt to the battery level (Chapter 15) and why we refuse to keep the device awake for work that can wait.

Privacy: the user decides what the app may know

Location is among the most sensitive data a phone holds. Recent Android versions let the user grant only approximate location or access "only this time", and background location needs a separate grant and, on Google Play, a declaration and review. Notifications became opt-in on Android 13. Each rule

exists because users were surprised by what apps did when nobody was watching. Waypoint therefore **never asks for background location**: recording runs in a foreground service the user starts from a visible screen, with an ongoing notification. Chapter 12 explains why this choice works.

Device diversity: not every phone follows the same script

Android runs on devices from many manufacturers, and some of them add their own battery management on top of the platform's rules. Those additions can stop background work more aggressively than stock Android does, and they differ between brands and even between software updates. We will not make claims about specific manufacturers in this book, because such behavior changes often and is hard to verify. Instead we will design Waypoint so that it does the right thing under the platform's documented rules, degrades gracefully when the device is stricter, and tells the user what is going on when it matters. Chapter 4 returns to this topic.

> **NOTE**
>
> Throughout the book we explain *why* a rule exists before showing *how* to comply. Rules change between Android versions; the reasons behind them change far more slowly. If you understand that the system protects battery and privacy, you can usually predict what a new restriction will look like before you read the release notes.

The Versions This Book Targets

Background behavior depends heavily on two numbers: the Android version running on the device, and the `targetSdk` your app declares. Many restrictions apply only "when your app targets" a certain level, so the same device can treat two apps differently. Here are the versions the sample project uses at the time of writing:

Setting	Value	What it means for Waypoint
`compileSdk`	37 (Android 17)	We can call the newest APIs; no behavior change by itself
`targetSdk`	36 (Android 16)	Opts in to every behavior change up to Android 16
`minSdk`	26 (Android 8.0)	Oldest supported devices; background limits always apply
WorkManager	2.12.0	Deferrable, guaranteed work
Play services location	21.4.0	Fused Location Provider
Glance	1.2.0	Home-screen widget

The rest of the toolchain is current as well: Android Gradle Plugin 9.4.1, Gradle 9.8.0, Kotlin 2.4.20, Compose BOM 2026.09.00, Hilt 2.60.1 with `androidx.hilt` 1.4.0 for workers, Room 2.8.5, core-ktx 1.19.1, kotlinx-coroutines 1.11.0, and Robolectric 4.17 for JVM tests. The project builds with JDK 21 and produces Java 17 bytecode.

Three numbers deserve a closer look. `compileSdk` only decides which APIs the compiler can see; raising it never changes how your app behaves on a device. `targetSdk` is a promise to the system: "I have tested with the rules of this version, apply them to me." Most of the background restrictions in Chapter 3 are keyed to it. `minSdk` is the oldest Android version you install on. Waypoint's minimum is Android 8.0, API 26, and that has a pleasant consequence: **Doze, App Standby, background execution limits and notification channels exist on every device Waypoint supports**. We never write a code path for a world without them.

> **WARNING**
>
> We compile against Android 17 but target Android 16. Do not raise `targetSdk` to 37 just because the SDK is installed. A higher target opts your app into new behavior changes; read the behavior-changes page for that level and re-test every background feature first. Chapter 27 gives a checklist for that upgrade.

A Tour of the Sample Project

The book comes with a finished sample project: Waypoint's background and system-integration layer. Each chapter builds a piece of it step by step, and the sample shows where every piece ends up. Whenever a chapter presents the final version of a class, it matches the sample exactly; earlier, simpler versions are clearly marked as such.

The UI in the sample is deliberately small: one Compose screen with a "Start hike" button, a "Sync now" button, and a line that shows the live recording. The interesting code is everything around that screen.

Three modules

The project has three Gradle modules. The settings file is short enough to show in full:

settings.gradle.kts

```
pluginManagement {
    repositories {
        google()
        mavenCentral()
        gradlePluginPortal()
    }
}

dependencyResolutionManagement {
    repositoriesMode.set(RepositoriesMode.FAIL_ON_PROJECT_REPOS)
    repositories {
        google()
        mavenCentral()
    }
}

rootProject.name = "waypoint-background"
```

```
include(":app")
include(":core:model")
include(":core:data")
```

Each module has one job, and the dependencies point in one direction only:

```
+-------------------------------------------------------------+
| :app  (Android application, com.northbeam.waypoint)         |
|   work/  recording/  location/  notifications/  alarms/     |
|   receivers/  share/  deeplink/  shortcuts/  widget/        |
+-----------------------------+-------------------------------+
                              | depends on
+-----------------------------v-------------------------------+
| :core:data  (Android library: Room, repositories, sync)     |
|   WaypointDatabase, TripRepository, TripSyncer, WaypointApi |
+-----------------------------+-------------------------------+
                              | api(...)
+-----------------------------v-------------------------------+
| :core:model  (pure Kotlin/JVM: no Android at all)           |
|   Trail, Trip, TrackPoint, TrackMath, WaypointFormat        |
+-------------------------------------------------------------+
```

- **`:core:model`** is a plain Kotlin/JVM module. It holds the canonical models (`Trail`, `Trip`, `TrackPoint` and friends), the track calculations in `TrackMath`, the display helpers in `WaypointFormat`, and the `SampleData` object. Because it has no Android dependency, its tests run in milliseconds.
- **`:core:data`** is an Android library with Room. It owns `WaypointDatabase` (file `"waypoint.db"`, tables `trails`, `trips` and `track_points`), the `TripRepository` interface and its `RoomTripRepository` implementation, `TrailRepository`, the `TripSyncer` that uploads pending data, and the fictional `WaypointApi` with its in-memory `FakeWaypointApi`.
- **`:app`** is the application. It contains every piece of code this book is really about: workers, the recording service, location, notifications, alarms, receivers, sharing, deep links, shortcuts and the widget.

The backend is fictional: `https://api.waypoint.example/v1/` uses a reserved example domain. The sample talks to the `WaypointApi` interface, and tests use `FakeWaypointApi`, which can be told to fail a number of times in a row so we can exercise retries.

The package map

Inside `:app`, every concern lives in its own package under `com.northbeam.waypoint`. The names below are canonical: you will see them in every chapter, in the tests, and in the final case study.

Package	Main classes	Built in
(root)	`WaypointApp`, `MainActivity`	1, 10, 22

`.work`	`WorkScheduler`, `WorkNames`, `TripSyncWorker`, `TrailRefreshWorker`, `FinishTripWorker`	6–10
`.recording`	`TripRecordingService`, `RecordingState`, `RecordingProfile`	11, 14, 15
`.location`	`LocationClient`, `FusedLocationClient`, `LocationFix`, `LocationPermissions`	12, 13
`.notifications`	`WaypointNotifications`	16, 17
`.alarms`	`TripReminderScheduler`, `TripReminderReceiver`	18
`.receivers`	`BootReceiver`, `SyncRetryReceiver`	19
`.share`	`TripShare`, `GpxWriter`	21
`.deeplink`	`DeepLink`, `DeepLinkParser`	22
`.shortcuts`	`WaypointShortcuts`	23
`.widget`	`NextTripWidget`, `NextTripWidgetReceiver`, `WidgetEntryPoint`	24

Dependency injection uses Hilt. Three modules wire things together: `DatabaseModule` and `RepositoryModule` in `core.data.di`, and `LocationModule` in the location package. The location module matters for testing: the real `FusedLocationClient` needs Google Play services, which do not exist in a JVM test, so the service depends on the small `LocationClient` interface and tests replace it with a `FakeLocationClient`. We will meet that design properly in Chapter 13; it is a good habit anyway, because it keeps a third-party API at the edge of the app.

The application class

The entry point is short, but it already hints at two topics of the book: notification channels and on-demand WorkManager initialization.

app/src/main/kotlin/com/northbeam/waypoint/WaypointApp.kt

```
@HiltAndroidApp
class WaypointApp : Application(), Configuration.Provider {

    @Inject lateinit var workerFactory: HiltWorkerFactory

    override val workManagerConfiguration: Configuration
        get() = Configuration.Builder()
            .setWorkerFactory(workerFactory)
            .build()

    override fun onCreate() {
        super.onCreate()
        WaypointNotifications.createChannels(this)
    }
}
```

Two things happen here. First, the app creates its four notification channels every time the process starts; that is cheap and safe, and Chapter 16 explains why channels must exist before any notification is posted. Second, the class implements `Configuration.Provider` so that WorkManager builds workers through Hilt. That requires removing WorkManager's default initializer in the manifest, a detail Chapter 10 covers in full. For now, remember that this one file is where the app process "wakes up" for every component: an activity, a worker, a receiver or a service. Chapter 2 explains why that matters.

The sample data

The models are shared by every book in the series. Eight sample trails, `tr-001` to `tr-008`, carry fixed names and numbers: for example `tr-001` is Cedar Loop in Pine Ridge (4.2 km, EASY), `tr-002` is Falcon Crest Trail (9.8 km, HARD), and `tr-004` is Granite Steps at Mirror Lake (7.1 km, MODERATE). The sample user is `u-100`, Alex Rivera.

This book adds three trips, one for each state a trip can be in when background code touches it:

Trip	Trail	Date and party	Status	Used for
`tp-300`	Cedar Loop	Sat 2026-09-26, 2 people	FINISHED, 4310.0 m, +118 m	Summaries, sharing
`tp-301`	Granite Steps	Sat 2026-10-17, 3 people	PLANNED	Reminder, alarm, widget
`tp-302`	Falcon Crest Trail	Sat 2026-10-10, 1 person	ACTIVE while recording	Recording, sync

The finished hike, `tp-300`, started at 08:00 UTC and ended at 09:32 UTC, so its share text reads "4.3 km in 1 h 32 min, +118 m". The planned trip, `tp-301`, gets a reminder on Friday 2026-10-16 at 18:00, titled "Tomorrow: Granite Steps". The recording trip, `tp-302`, started at 07:30 UTC; the sample holds its first five minutes as `SampleData.sampleTrack`, six location fixes one minute apart. One of those fixes has a 30-meter accuracy radius and is ignored for distance, which is why the five-minute track measures 570 m with 16 m of elevation gain. Chapter 13 works through those numbers in detail.

When the app starts for the first time, `SampleSeeder.seedIfEmpty()`, called from `MainActivity`, inserts the eight trails and the three trips into Room, all marked as already synced. It stores `tp-302` as PLANNED with no start time, so that tapping "Start hike" turns it into an active recording. Tests skip the seeder and use `SampleData.recordingTrip`, which is already ACTIVE.

> **TIP**
>
> Throughout the book, "today" is Tuesday 2026-10-06. Tests pin time with `Clock.fixed(...)` at 09:00 UTC on that day, and they run in the UTC time zone with a US locale. Pinning the clock and the zone is the single most effective way to make time-dependent background code testable; you will see it again in the alarm chapter.

Getting the Project Running

You need a recent stable Android Studio with JDK 21 (the bundled JetBrains Runtime works) and an Android SDK that includes platform 37. The sample does not need an emulator to build or to run its tests; almost every test in this book runs on the JVM with Robolectric. You will want a device or an emulator later to watch notifications, services and widgets for real.

1. **Make a working copy.** Copy the `sample` folder to a place of your own, for example `waypoint-background` in your projects folder. Keep the original untouched so that you can always compare your version with the finished one.
2. **Skip the build outputs.** If you copy a folder that has been built before, leave out the `build`, `.gradle` and `.idea` folders and any `local.properties` file. Android Studio recreates them for your machine.
3. **Open the copy in Android Studio.** Use File, Open and choose the folder that contains `settings.gradle.kts`. Studio writes a `local.properties` pointing at your SDK and starts a Gradle sync. The first sync downloads the dependencies, which can take a while.
4. **Run the tests.** In the Project view, right-click the root project or a module and choose "Run tests", or open the Gradle tool window and run the `test` task. From a terminal in the project folder you can run the wrapper directly.

Terminal (project folder)

```
# macOS / Linux
./gradlew test assembleDebug

# Windows
gradlew.bat test assembleDebug
```

The `test` task runs the unit tests of all three modules; `assembleDebug` builds an APK you can install later. When everything is in place, you should see something like this at the end of the output:

Gradle output (something like)

```
> Task :core:model:test
> Task :core:data:testDebugUnitTest
> Task :app:testDebugUnitTest
> Task :app:assembleDebug

BUILD SUCCESSFUL
```

The sample contains 23 tests in 9 classes, and all of them pass. Gradle writes an HTML report for each module under `build/reports/tests`; open the one for `:app` to see the test classes by name. Here is what they cover, so you know what "green" means before you change anything:

Test class	Tests	What it proves
`TrackMathTest`	3	Distance, gain and duration of the sample track

`TripSyncerTest`	2	Batched upload against an in-memory Room database
`TripSyncWorkerTest`	3	Success, retry, and giving up with a notification
`WorkSchedulerTest`	3	Constraints, periodic refresh, the finish chain
`TripRecordingServiceTest`	1	The foreground service records a fake track
`WaypointNotificationsTest`	3	Channels and notification content
`TripReminderSchedulerTest`	2	The reminder alarm time and type
`SyncRetryReceiverTest`	2	The Retry action re-enqueues sync
`PureLogicTest`	4	Deep links, share text, recording profile, GPX

A foreground service, notifications and alarms can all be tested without a device if you keep platform calls at the edges and inject the parts that depend on Google Play services. Things that truly need a device, such as real GPS fixes, real Doze, or a widget on a home screen, are marked clearly, and their output is shown as "you should see something like" because exact results vary by device.

> **WARNING**
>
> If the build fails with a message about the SDK location, the copy has no `local.properties` or it points at another machine's SDK. Let Android Studio regenerate it, or create it yourself with a single `sdk.dir=...` line pointing at your Android SDK folder. Never commit that file; it is machine-specific and the sample's `.gitignore` already excludes it.

One Hike, End to End

Before we take the system apart, it helps to see how the pieces cooperate during a single trip. Here is the life of `tp-301`, the Granite Steps trip, as the finished app handles it:

```
Plan trip (UI)                      Room: trip saved, needsSync = true
   |                                   |
   +--> TripReminderScheduler          +--> WorkScheduler.enqueueTripSync()
   |      alarm: Fri 18:00 local              unique work "trip-sync", needs network
   |
Fri 18:00  TripReminderReceiver --> "Tomorrow: Granite Steps"
   |
Sat  "Start hike" (visible UI) --> TripRecordingService (location FGS)
   |      every fix --> Room track_points; notification "Recording ..."
   |      3 hours of moving --> "You've been moving for 3 hours"
   |
"Stop" --> WorkScheduler.enqueueFinishTrip(id)
   |      unique chain "finish-trip-<id>": FinishTripWorker --> TripSyncWorker
   |
Offline? work waits for a network, retries with backoff, gives up after 5 tries
   |
Online --> trips PUT, points POSTed in batches of 500 --> synced
```

A few design decisions are already visible in this diagram, and each one is a rule we will justify later:

- **Room is the source of truth.** The UI, the service and the workers all read and write the local database. The network is a destination, never a dependency for saving data.
- **Only one class enqueues work.** `WorkScheduler` owns every unique work name, constraint and policy. When we need to know why something ran twice, there is one file to read.
- **Recording starts only from visible UI.** That single rule satisfies the foreground-service restrictions of Android 12 and later and keeps location access in the "while in use" category.
- **Time-of-day features use alarms, not workers.** WorkManager is excellent at "eventually"; it is not an alarm clock. The reminder uses `AlarmManager`.
- **Every hand-off is explicit.** Notification actions, alarms and receivers use explicit intents to non-exported components, so other apps cannot trigger them.

Not every feature is in the diagram. Sharing, deep links, shortcuts and the widget sit around this core and reuse its pieces: the share text and the widget subtitle use the same `WaypointFormat` helpers as the notifications, and links, shortcuts and notification taps all go through one `DeepLinkParser`.

How This Book Is Organized

The 28 chapters are grouped into six parts. You can read them in order, which is how the sample is built, or jump to the part you need today; each chapter points back to the earlier material it relies on.

Part	Chapters	What you will be able to do
I. How Android Runs Your App	1–5	Predict when the system will run, delay or kill your code, and pick the right tool
II. WorkManager	6–10	Build reliable, testable, offline-first sync with Hilt workers
III. Foreground Services and Location	11–15	Record a hike end to end, within the Android 14+ rules, without draining the battery
IV. Notifications and Alarms	16–19	Post the right notification at the right time and survive reboots
V. Integrating with the System	20–24	Share, open from links and shortcuts, and show a widget
VI. Testing, Debugging, and Shipping	25–28	Test it on the JVM, debug it on a device, and pass Play review

Part I contains almost no feature code, on purpose: Chapters 2 to 4 explain how Android decides which processes live, which background rules apply, and how Doze and App Standby work, and Chapter 5 turns that into a decision guide. If you know the platform rules well, skim to Chapter 5 and continue with WorkManager in Chapter 6. The appendices hold a cheat sheet, templates, a production checklist and a glossary.