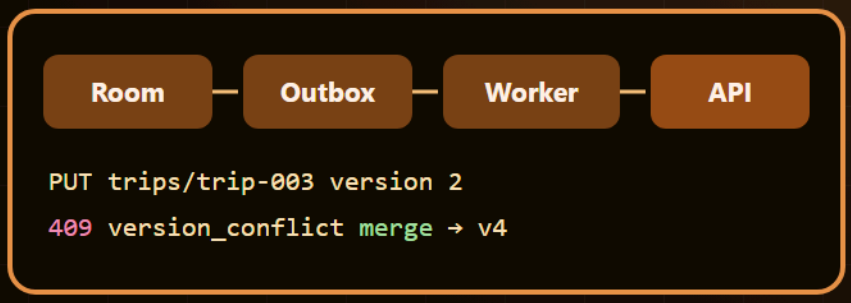
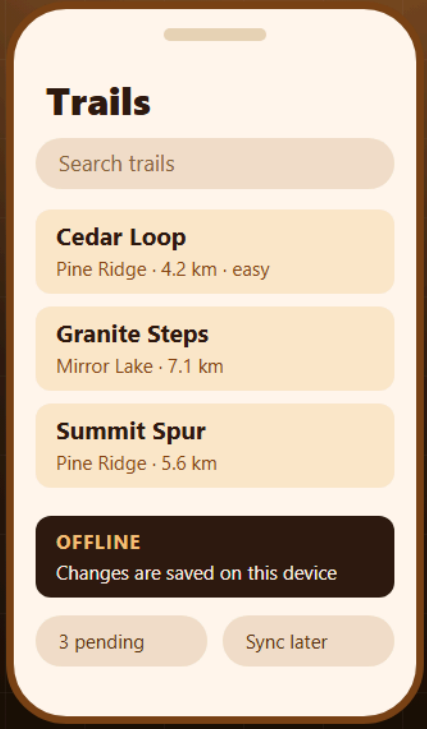



```kotlin
@Dao interface TrailDao {
  @Query("SELECT * FROM trails")
  fun observeAll(): Flow<List<TrailEntity>>
  @Upsert
  suspend fun upsertAll(t: List<TrailEntity>)
}
```





A COMPLETE, PRACTICAL GUIDE

Room, DataStore, and Offline-First Android

Local Databases, Migrations, Reactive Queries, Preferences, Sync, and Conflict Resolution

Room · FTS Search · AutoMigrations · Flow · Paging 3 · DataStore
Outbox · WorkManager Sync · Conflict Rules · JVM Testing

STORE · MIGRATE · OBSERVE · SYNC · TEST

THENEXBOOKS

Room, DataStore, and Offline-First Android

Local Databases, Migrations, Reactive Queries, Preferences, Sync, and Conflict Resolution
Build an Android app that keeps working when the network does not

First Edition

THENEXBOOKS

Room, DataStore, and Offline-First Android
Local Databases, Migrations, Reactive Queries, Preferences, Sync, and Conflict Resolution

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, SQL, and build scripts 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, Jetpack libraries, Kotlin, Gradle, and Android Studio change often; APIs, defaults, and tool behavior can differ between versions. The versions named in this book were current at the time of writing. Always consult the official documentation for the versions you use, and back up user data before you ship a database migration.

Android, Google Play, Jetpack Compose, and Android Studio are trademarks of Google LLC. Kotlin is a trademark of JetBrains s.r.o. and the Kotlin Foundation. Java is a trademark of Oracle and/or its affiliates. SQLite is in the public domain; the name is used for identification. Gradle is a trademark of Gradle, Inc. Other product names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement. This book is independent and is not affiliated with, sponsored by, or endorsed by Google, JetBrains, or any other company mentioned.

Northbeam Labs, the Waypoint app, its team members, trails, trips, journal entries, users, and the backend at `api.waypoint.example` are fictional and used as a running example only. All sample data is invented; email addresses and host names use reserved example domains.

Preface

Mobile apps live on unreliable networks. A phone moves from fast Wi-Fi to a weak cellular signal to no signal at all, often within minutes, and the people holding it do not stop using the app while that happens. An app that shows a spinner, an error dialog, or an empty screen every time the connection drops feels broken, even if the code behind it is technically correct. An app that keeps working, saves what the user did, and quietly catches up later feels trustworthy.

That second kind of app is called **offline-first**. Its local storage is not a cache bolted on at the end; it is the place where the truth lives on the device. The user interface reads from local storage, every change is written locally first, and synchronization with a server is a background job that can fail, retry, and resolve disagreements without bothering anyone. Building that well requires a solid grasp of several tools that Android developers often learn only in fragments: SQLite and Room for structured data, schema migrations that never lose a user's data, reactive queries that keep screens up to date, DataStore for settings and small typed state, files for photos, Paging for long lists, WorkManager for reliable background sync, and a clear policy for what happens when two devices change the same thing.

This book teaches all of those pieces together, in the order you need them, by building the data layer of one realistic app. We explain *why* each tool works the way it does before we show *how* to use it, we show the failure cases as well as the happy path, and we test everything on the JVM so you can run the examples on any development machine.

Who This Book Is For

- **Android developers** who can build screens with Kotlin and Jetpack Compose and now need data that survives process death, app updates, and missing networks.
- **Developers who use Room or DataStore already** but want to understand migrations, transactions, invalidation, paging, and testing deeply enough to stop being surprised by them.
- **Teams planning offline support** who need a concrete, tested design for an outbox, background sync, and conflict resolution rather than a whiteboard sketch.
- **Developers coming from other platforms** who know SQL or local databases elsewhere and want the idiomatic Android way.

You should be comfortable with Kotlin, including classes, lambdas, and the basics of coroutines and `Flow`. You do not need any previous experience with SQL, Room, or DataStore; Chapter 2 teaches the SQL we use.

What You Will Learn

- How SQLite works on Android, and how Room turns it into type-safe, compile-time-checked Kotlin: entities, DAOs, relationships, type converters, projections, upserts, and full-text search.

- How to observe the database with `Flow`, keep multi-step writes atomic with transactions, and understand Room's threading and invalidation model.
- How to evolve a shipped schema safely: schema export, AutoMigrations with specs, manual migrations with data backfills, and migration tests that run on every build.
- How to store settings with Preferences DataStore, typed state with kotlinx.serialization, and photos with app-private files and MediaStore.
- How to page large lists from Room and from the network with Paging 3 and `RemoteMediator`.
- How to design an offline-first repository, record local changes in an outbox, sync with WorkManager, detect conflicts with versions, and resolve them with explicit rules.
- How to test the whole data layer on the JVM with Robolectric, Turbine, fakes, and the WorkManager and Paging testing libraries, and how to investigate performance, backups, privacy, and data deletion before release.

How This Book Is Organized

Part I — Local Data Foundations (Chapters 1–4) explains offline-first thinking, introduces SQLite, and builds the first Room database and its queries. **Part II — Room in Depth** (Chapters 5–9) covers type converters, relationships, reactive queries, transactions, and full-text search. **Part III — Evolving the Schema** (Chapters 10–13) ships version 1 and then evolves it with automatic and manual migrations, all covered by tests. **Part IV — Preferences, Files, and Paging** (Chapters 14–18) adds Preferences and typed DataStore, journal photos, and paging from the database and from the network. **Part V — Offline-First Sync** (Chapters 19–23) builds the offline-first repository, the outbox, WorkManager sync, conflict resolution, and sync status in the UI. **Part VI — Testing and Production** (Chapters 24–28) covers testing strategy, end-to-end sync tests, performance and debugging, data safety, and a complete case study. The appendices provide a cheat sheet, templates and reference tables, an offline-first 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 build the data layer of **Waypoint**, a hiking trip planner and tracker made by **Northbeam Labs**, a small fictional product studio. Hikers browse and search trails, plan trips with a date, party size, and packing checklist, record hikes as GPS track points, keep a private journal with photos, and expect all of it to work on a mountain with no signal. When the phone is back online, Waypoint syncs trips and journal entries with its backend — and sometimes discovers that the same trip was edited on the web in the meantime.

The book starts from a small, tested starter project that already contains Waypoint's models, eight sample trails, a fake backend, and the first version of the database. Chapter by chapter we add migrations, settings, photos, paging, an outbox, background sync, and conflict resolution. The sample

data is small enough to check by hand, and every query result and test output printed in this book comes from running the code. There is no real Waypoint server: a fake backend that behaves like the documented API lets you run everything on your own machine.

Versions and Conventions

At the time of writing, the examples use **Room 2.8**, **DataStore 1.2**, **Paging 3.5**, **WorkManager 2.12**, **Kotlin 2.4** with KSP, the **Android Gradle Plugin 9.4**, and Gradle 9.8, with `minSdk 26`, `targetSdk 36`, and `compileSdk 37`. Exact versions are listed in Chapter 1 and Appendix B. Libraries evolve quickly; when a detail may differ in recent versions, a Note says so. Tests run on the JVM with Robolectric, so you do not need an emulator to follow along, although a device or emulator is useful for the few features that only exist there, which we label clearly.

> **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 data loss, crashes, or security problems.

Code, SQL, file names, and identifiers appear in `monospace`; long lines are wrapped to fit the page. File captions above listings show where the code lives in the project, such as `core/database/.../Daos.kt`, where `...` stands for `src/main/kotlin/com/northbeam/waypoint`. Output blocks show real output from running the code unless they are explicitly labeled "you should see something like".

Contents

Preface

PART I — LOCAL DATA FOUNDATIONS

1 Offline-First Android: Why Local Data Comes First

2 SQLite on Android: The Engine Under Room

3 Your First Room Database: Entities, DAOs, and the Database Class

4 DAO Queries in Depth: Parameters, Projections, Upserts, and Indices

PART II — ROOM IN DEPTH

5 Type Converters, Enums, and Embedded Values

6 Relationships: One-to-Many, Many-to-Many, and Foreign Keys

7 Reactive Queries with Flow

8 Transactions and Concurrency

9 Full-Text Search for Trails with FTS4

PART III — EVOLVING THE SCHEMA

10 Schema Export and a Versioning Strategy

11 AutoMigrations: Added Columns, Renames, and Specs

12 Manual Migrations and Data Transformations

13 Testing Migrations with MigrationTestHelper

PART IV — PREFERENCES, FILES, AND PAGING

14 Preferences DataStore: Settings Done Right

15 Typed DataStore with kotlinx.serialization (and Proto)

16 Files, MediaStore, and Journal Photos

17 Paging 3 from Room

18 Paging from the Network with RemoteMediator

PART V — OFFLINE-FIRST SYNC

19 The Offline-First Repository: Room as the Source of Truth

20 The Outbox Pattern: Recording Local Changes

21 Syncing with WorkManager

22 Conflict Resolution

23 Sync Status, Connectivity, and Errors in the UI

PART VI — TESTING AND PRODUCTION

24 Testing the Data Layer on the JVM

25 Testing Sync End to End

26 Performance and Debugging: Query Plans, Indices, and Inspection

27 Data Safety: Backup, Encryption, Privacy, and Deleting User Data

28 Case Study: Waypoint's Offline-First Data Layer, End to End

APPENDICES

A Room, DataStore, and Paging Cheat Sheet

B Templates and Reference

C Offline-First Production Checklist

D Glossary

PART I

Local Data Foundations

Why offline-first apps keep the truth on the device, how SQLite stores and queries data on Android, and how Room turns tables and SQL into type-safe Kotlin: entities, DAOs, the database class, and the queries Waypoint needs every day.

1 · Offline-First Android

2 · SQLite on Android

3 · Your First Room Database

4 · DAO Queries in Depth

CHAPTER 1 · PART I — LOCAL DATA FOUNDATIONS

Offline-First Android: Why Local Data Comes First

A hiking app that needs a signal to show you your own trip is broken exactly when it matters most. In this chapter we define what offline-first really means, meet the Waypoint app and its team, decide where every piece of Waypoint's data should live, and take a guided tour of the starter project you will extend for the rest of the book.

A Hike Without Signal

Picture a Saturday morning at the Summit Spur trailhead. Alex has planned the hike for a week: the trip is in the app, the packing checklist is ticked off, and the trail details were downloaded on Thursday. The parking lot has one bar of signal. Two hundred meters up the trail it has none. Alex opens the app to start recording and sees a spinner, then an error: "Could not load your trip. Check your connection."

Nothing about that moment requires a server. The trip was created on this phone, the trail data was already downloaded, and GPS works without a network. The app failed because it was *designed* to ask the server first, which is the simplest thing to write: call an API, show the result, show an error when the call fails.

For a news reader that might be acceptable. For a hiking app it is a design flaw: trails go through valleys and canyons, and even in a city a train tunnel can turn a fast connection into a useless one. If the main features depend on the network, the app is only as reliable as the weakest signal on the route.

This book is about building the other kind of app: one where the phone holds the data the user needs, the user can read and change it at any time, and the network is used to keep that data in step with a server whenever a connection happens to be available. That style is called **offline-first**, and the tools Android gives us for it are the subject of every chapter that follows: SQLite and Room for structured data, DataStore for settings and small typed values, the file system for photos, Paging for long lists, and WorkManager for reliable background sync.

What Offline-First Means

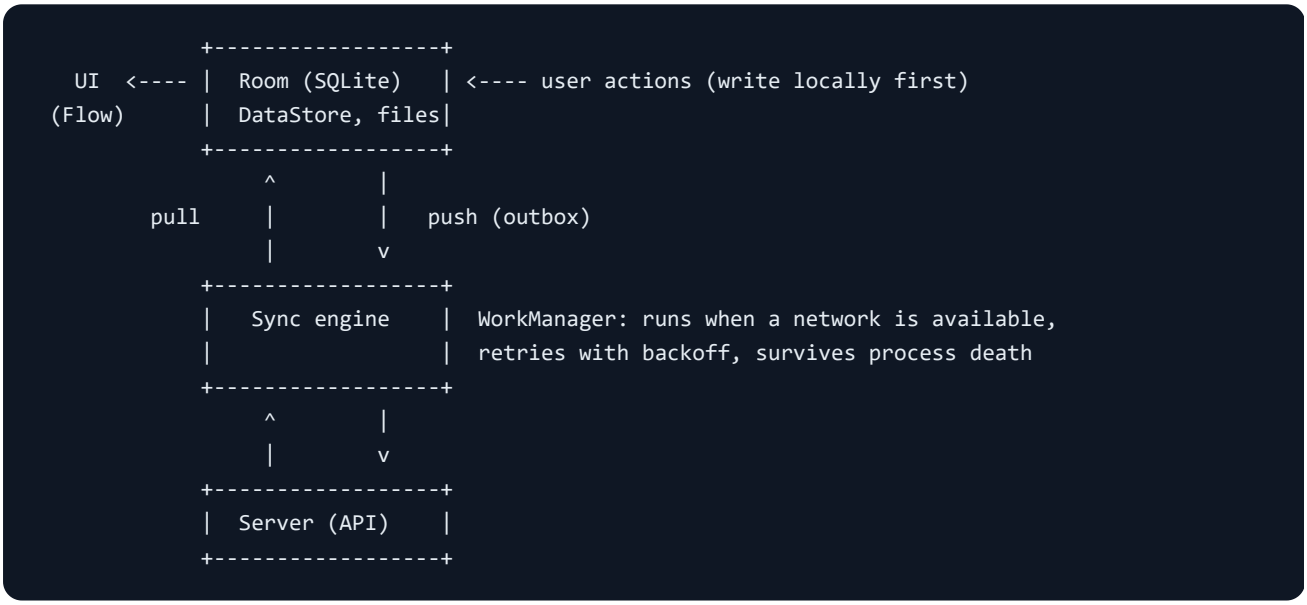
"Offline support" is a vague phrase. Some teams use it to mean "we show a nice error message when there is no network." Others mean "we cache the last response and show it read-only." Offline-first is a more specific and more demanding idea, and it is worth stating precisely because every design decision in this book follows from it.

In an offline-first app, the local database is the source of truth for everything the user sees, and the network is a synchronization mechanism. The UI never reads from the network directly. It reads from the local database and re-renders whenever the database changes. User actions write to the local database first, immediately, and succeed even without a connection. A separate process, usually running in the background, pushes local changes to the server and pulls remote changes into the local database.

Three consequences follow from that definition:

- **Reads are always local.** Screens observe queries on the device. If the server has newer data, it reaches the screen by first being written into the database, never by bypassing it.
- **Writes are always local first.** Planning a trip, finishing a hike, or editing a journal entry is a database write that completes in milliseconds. Whether the server has heard about it yet is a separate, visible state ("3 changes waiting to sync"), not a reason to block the user.
- **Sync is a first-class part of the design.** Because both the device and the server can change data independently, we need a way to record pending changes, retry them reliably, and resolve conflicts when two edits collide.

The following diagram shows the shape we will build. Arrows point in the direction data flows.



Compare that with the two designs most apps start from. The table below is a useful way to explain the difference to a product manager or a new teammate.

Design	Where the UI reads	Where writes go	Without a network
Online-only	The API response	Straight to the API	Errors and spinners
Cache-aside	API, falling back to a cache	Straight to the API	Stale, read-only data
Offline-first	The local database, always	Local database, then synced	Fully usable; sync waits

The cache-aside design is a common halfway house, and it is better than nothing, but it has two problems. First, it has two code paths for every screen (network result or cached result), which doubles the number of states to test. Second, it does not help with writes at all: if creating a trip requires a successful API call, Alex still cannot plan a trip at the trailhead. Offline-first has one read path and one write path, both local, and pushes all the complexity of the network into a single place: the sync layer.

> **NOTE**
> Offline-first does not mean "never online." Waypoint still downloads trails, uploads trips, and shows data from other devices. It means the app's correctness does not depend on the network being available at the moment the user acts. The network makes the data more complete; it is not required for the app to work.

The Honest Costs

Offline-first is not free, and it would be misleading to present it as a pure win. Before committing to it, a team should understand what it is signing up for.

- **More code in the data layer.** You need a schema, migrations, a sync engine, an outbox of pending changes, and conflict rules. An online-only app needs none of these.
- **Conflicts become your problem.** If Alex edits a trip on the phone while offline and also edits it on the web, which version wins? An online-only app avoids the question because the server serializes every write. An offline-first app has to answer it explicitly. Chapter 22 is entirely about that answer.
- **Schema evolution is permanent.** Once version 1.0 ships, every user has a database file with your schema on their phone. You can never again change a table casually; every change needs a migration that is tested against real old data. Part III of this book exists because of this.
- **Stale data is possible.** The trail list on the phone may be a day old. The app must make freshness visible where it matters and refresh in the background where it can.
- **Storage and privacy matter more.** Data on the device can be backed up, copied, or left behind. A private hiking journal deserves careful handling, which Chapter 27 covers.

For Waypoint, the trade is clearly worth it: the core use case happens where the network is worst.

Meet Northbeam Labs and Waypoint

Throughout this book we follow a small fictional product studio, **Northbeam Labs**, as it builds the data layer of its hiking app, **Waypoint**. The team is four people: Lena Ortiz leads Android development, Sam Okafor owns the backend, Priya Natarajan handles product and design, and Theo Brandt runs QA. They will appear now and then when a decision or a bug report makes a concept concrete.

Waypoint helps hikers do four things:

1. **Browse and search trails.** The app downloads a catalog of trails (name, region, distance, elevation gain, difficulty, rating) and lets users search it by name or region.
2. **Plan and record trips.** A trip belongs to a trail, has a planned date and a party size, and moves through the states PLANNED, ACTIVE, FINISHED, or CANCELLED. While a trip is active, the app records GPS track points. Finishing a trip computes the distance and elevation gain from those points.
3. **Keep a private journal.** Each trip can have journal entries with text and an optional photo.
4. **Sync.** Trips and journal entries are written on the device and synced to Northbeam's backend so they appear on the web and on other devices.

This book builds Waypoint's data layer. The user interface stays deliberately small (a trail list screen in the starter).

Priya's product brief for version 1.0 contains one sentence that shapes everything: *"Everything you can do on a trail, you can do without signal."* Lena turned it into three engineering rules that you will see enforced chapter after chapter:

- Screens observe Room, never the API.
- Every user action is a local write that cannot fail because of the network.
- Anything that must reach the server goes through one sync path that retries until it succeeds or is clearly marked as needing attention.

What Waypoint Stores

Before choosing tools, list the data. It is tempting to say "we will put everything in Room," but different kinds of data have very different shapes, sizes, and lifetimes, and Android offers different storage for each. Here is the inventory for Waypoint.

- **Trail cache.** Around a few dozen to a few thousand rows of structured data downloaded from the server. Searched by text, filtered by region, sorted by name, shown in long scrolling lists. The server owns it; the device keeps a copy plus a little local-only data such as favorites (added in Chapter 11).
- **Trips.** Structured, relational (a trip points to a trail), created and edited on the device, synced both ways.
- **Track points.** Potentially thousands per trip, append-only, written in bursts while recording, read in order when finishing a trip.
- **Journal entries.** Small text rows linked to a trip, with an optional photo. Private and sensitive.
- **Journal photos.** Binary files, potentially several megabytes each. Not something to store inside a database row.

- **Checklists and collections.** Small structured lists linked to trips or trails ("Water (2 L each)", "Weekend Hikes").
- **User settings.** A handful of values: distance unit, sync on Wi-Fi only, the last region filter, whether onboarding is complete.
- **Sync metadata.** A device ID, a pull cursor, the time of the last successful sync, a failure counter. Small, typed, read and written by the sync engine.
- **Screen state.** The current search query, scroll position, whether a dialog is open. Short-lived; never written to disk by the data layer.

Choosing the Right Storage

With the inventory in hand, the choice of tool for each item becomes much less arbitrary. The rule of thumb we follow throughout the book: **relational or queryable data goes into Room; a small set of independent values goes into Preferences DataStore; a small typed object goes into a typed DataStore; large binary content goes into files with a path stored in Room; transient UI state stays in memory.**

Data	Shape	Where it lives	Chapter
Trail cache	Rows, searched and paged	Room (`trails` + FTS)	3, 9, 17, 18
Trips, track points	Related rows, many writes	Room (`trips` , `track_points`)	6, 8, 19
Journal entries	Rows linked to trips	Room (`journal_entries`)	6, 16, 19
Journal photos	Binary files	App files + path in Room	16
Settings	A few key-value pairs	Preferences DataStore	14
Sync metadata	One small typed object	Typed DataStore (JSON)	15
Pending changes	Queue of rows	Room (`outbox`)	20
Screen state	Transient values	Memory (ViewModel)	7

A few of these choices deserve a sentence of justification now, because they are frequently done differently in real projects.

Why not put settings in Room? You could, but settings have no relations, are read as a single unit, and benefit from DataStore's simple transactional `edit { }` API and its migration from the old `SharedPreferences` . Using a full database table for four values adds schema and migration work with no benefit.

Why not put track points in DataStore or a JSON file? DataStore rewrites its entire file on every update. That is perfect for a few values and a poor fit for thousands of rows that grow during a hike. Room appends a row in a transaction and lets us query points by trip in timestamp order.

Why not store photos as BLOBs in Room? Large blobs make the database file grow quickly, slow down queries that touch the table, and complicate backups. A file per photo with a relative path in the row keeps the database small and lets us stream the bytes. Chapter 16 also shows how to keep files and rows consistent when one is deleted.

Why is the outbox in Room and not in memory? Pending changes must survive the app being killed, the phone rebooting, or the battery dying on the trail. Anything that must survive process death and must be updated atomically with the data it describes belongs in the same database as that data.

> **TIP**
>
> When you are unsure where something should live, ask two questions. Does it need to survive process death? If not, keep it in memory. Do you need to query it, relate it to other data, or update part of it atomically with other rows? If yes, use Room. Otherwise, DataStore or a file is usually simpler.

A Tour of the Starter Project

The book comes with a starter project: a small but complete version of Waypoint whose data layer already contains the version 1 database that Part I and Part II explain piece by piece. You can open it in Android Studio, build it, and run its tests on your computer without a device. From Chapter 10 on you will change your own copy as Waypoint's schema evolves through versions 2 to 6.

The project is split into Gradle modules, one per layer. Splitting the data layer from the UI is not just tidiness; it forces the rule "screens observe Room through repositories" to be visible in the dependency graph.

```
:app                Application class, MainActivity, navigation
:feature:trails     TrailsScreen (Compose) + TrailsViewModel
:core:data          repositories (TrailRepository), Hilt bindings, later: sync
:core:database      Room: entities, DAOs, WaypointDatabase, exported schemas
:core:network       WaypointApi (Retrofit), FakeWaypointApi
:core:model         plain Kotlin models (Trail, Trip, ...), SampleData, TrackMath
```

The dependency direction is strictly downward: `:feature:trails` depends on `:core:data`, which depends on `:core:database` and `:core:network`, both of which depend on `:core:model`. The feature module never sees a DAO or a Retrofit interface. It sees `TrailRepository` and the `Trail` model.

Here are the files you will meet most often (the `...` stands for `src/main/kotlin/com/northbeam/waypoint`):

```
app/.../WaypointApp.kt                @HiltAndroidApp + Configuration.Provider
app/.../MainActivity.kt               NavHost -> TrailsRoute
core/model/.../core/model/Models.kt   SampleData.kt   TrackMath.kt
core/database/.../core/database/      Entities.kt  Relations.kt  Daos.kt
                                      Converters.kt  Mappers.kt  FtsQuery.kt
```

```
                                        WaypointDatabase.kt (+ DatabaseModule)
core/database/schemas/com.northbeam.waypoint.core.database.WaypointDatabase/1.json
core/network/.../core/network/          WaypointApi.kt  FakeWaypointApi.kt
                                        di/NetworkModule.kt
core/data/.../core/data/                TrailRepository.kt  di/DataModule.kt
feature/trails/.../feature/trails/      TrailsViewModel.kt  TrailsScreen.kt
```

The models

`:core:model` holds plain Kotlin data classes with no Android or library annotations: `Trail`, `Trip`, `TrackPoint`, `JournalEntry`, `User`, plus `ChecklistItem` and `TrailCollection`. Times use `java.time.Instant` and dates use `java.time.LocalDate`, which are available without desugaring because Waypoint's minimum SDK is 26. Room entities and network DTOs are separate classes mapped to and from these models explicitly, so a column name or a JSON field can change without touching a single screen.

The same module contains `SampleData`, the canonical data set used by the fake backend, the tests, and this book's examples: eight trails from Cedar Loop (`tr-001`) to Brookside Way (`tr-008`) in three regions (Pine Ridge, Mirror Lake, Fernhollow), three trips in different states, five track points recorded on the Summit Spur hike, two journal entries, a three-item checklist, and one collection called "Weekend Hikes." Because these values are fixed, every output you see in this book can be reproduced exactly in your copy.

The fake backend

Waypoint's backend lives at `https://api.waypoint.example/v1/`, and that address does not exist. The `.example` domain is reserved for documentation, so there is no risk of the book's code talking to someone's real server. Instead, `:core:network` contains two implementations of the same interface:

- `WaypointApi` is a Retrofit interface. `createWaypointApi(baseUrl, client)` builds a real HTTP client for it, which the network tests point at a local MockWebServer.
- `FakeWaypointApi` implements `WaypointApi` in memory, using the sample trails. It counts requests (`requestCount`) and can be told to fail the next call (`failNextWith = FakeWaypointApi.offline()`), which is exactly what we need to simulate a hike without signal in a unit test.

The app's Hilt module provides `FakeWaypointApi`, so the starter runs on a device with no server at all. If Northbeam had a real backend, one line in `NetworkModule` would change and nothing else in the app would notice. That is a small, early proof that the repository boundary is doing its job.

The database module

`:core:database` already contains Waypoint's complete version 1 schema: eight tables for trails, the trail search index, trips, track points, journal entries, checklist items, collections, and the collection-trail join table. Chapters 3 through 9 explain how each piece is built and why, so for now it is enough to know that it is there and that it is tested. Notice the `schemas/` folder: Room writes a JSON description of each database version into it, and Part III of the book depends on those files.

Building and Running the Tests

Open the starter's root folder in Android Studio and let Gradle sync. The project targets the versions listed in the front matter at the time of writing (among them Room 2.8.5, DataStore 1.2.1, Kotlin 2.4.20, and AGP 9.4.1). Then run the unit tests from a terminal in the project folder:

```
./gradlew test          # macOS / Linux
gradlew.bat test        # Windows
```

The tests do not need an emulator or a device. Most of the data-layer tests use **Robolectric**, a library that runs Android framework code on the regular JVM. Its SQLite support uses a real SQLite engine, so Room, DataStore, WorkManager, and Paging behave as they do on a phone, while the test runs on your computer.

Gradle prints a line per test. On our machine, the database module's tests produced lines like these (the indented lines are values the tests print with `println`; one long line is cut with `...`):

Output (test run, database module, excerpt)

```
    SQLite version under Robolectric: 3.44.3
SchemaV1Test > createVersion1_fromExportedSchema PASSED
TrailDaoTest > observeAll_isSortedByName_andReEmitsAfterWrites PASSED
    Fernhollow | 2 | 14.700000000000001
    Mirror Lake | 3 | 21.6
    Pine Ridge | 3 | 19.6
TrailDaoTest > regionSummaries_groupAndSum PASSED
TrailDaoTest > ftsSearch_prefixMatchesNameAndRegion PASSED
TripDaoTest > deletingTrip_cascadesToChildren PASSED
    FK violation message: FOREIGN KEY constraint failed (code 787 SQLITE_CONSTRAINT...
TripDaoTest > foreignKey_rejectsPointForUnknownTrip PASSED
TripDaoTest > duplicateTrackPoints_areIgnored PASSED
```

The whole starter has 26 tests in 12 classes, all passing. A few of those outputs are teasers for later chapters: the odd `14.700000000000001` is a floating-point sum we discuss in Chapter 4, and the foreign key message comes from Chapter 6. The tests in `core/data/src/test/.../toolchain/` are small working templates for DataStore, WorkManager, and Paging; later chapters copy their setup code.

To run one module's tests, or one class, narrow the task:

```
./gradlew :core:database:testDebugUnitTest
./gradlew :core:data:testDebugUnitTest --tests "*OfflineFirstTrailRepositoryTest*"
```

> **WARNING**
>
> Gradle caches test results. If nothing changed since the last run, a test task is reported as up to date or loaded from the build cache, the tests do not run again, and nothing they print appears. When you want to see the output of a test again, add `--rerun` after the test task (for example `:core:database:testDebugUnitTest --rerun`).

In the IDE, every test class has a green run arrow in the gutter; Robolectric tests run as local unit tests.

A Map of the Book

The book is organized in six parts, each building on the previous one.

- **Part I — Local Data Foundations** (Chapters 1–4). Why offline-first, the SQL and SQLite you need, your first Room database, and DAO queries in depth.
- **Part II — Room in Depth** (Chapters 5–9). Type converters, relationships and foreign keys, reactive queries with Flow, transactions and concurrency, and full-text search for trails.
- **Part III — Evolving the Schema** (Chapters 10–13). Schema export, a versioning strategy, automatic and manual migrations, and testing migrations so that Waypoint's users never lose a trip when they update the app.
- **Part IV — Preferences, Files, and Paging** (Chapters 14–18). Preferences DataStore for settings, a typed DataStore for sync metadata, journal photos on disk and in the gallery, and Paging 3 from both Room and the network.
- **Part V — Offline-First Sync** (Chapters 19–23). The offline-first repository, the outbox pattern, sync with WorkManager, conflict resolution, and showing sync status in the UI.
- **Part VI — Testing and Production** (Chapters 24–28). Testing the data layer and the sync engine end to end, performance and debugging, data safety, and a case study that ties everything together.

The schema grows with the parts: version 1 ships in Waypoint 1.0, and by Chapter 20 Waypoint is at version 6, with every step covered by a migration test.

Walk-Through: `OfflineFirstTrailRepository.refresh()`

The best way to see the offline-first idea in code is the one repository the starter already has. Here is the interface and the implementation, from `core:data`:

core/data/.../core/data/TrailRepository.kt

```
interface TrailRepository {
    /** All cached trails, sorted by name.
     *  Emits again whenever the trails table changes. */
    fun observeTrails(): Flow<List<Trail>>

    fun observeTrail(id: String): Flow<Trail?>
```

```
    /** Full-text search over the cache. A blank query returns every trail. */
    fun search(input: String): Flow<List<Trail>>

    /** Downloads every trail page into Room. Returns the number of trails written.
     *  Throws on network errors. */
    suspend fun refresh(): Int
}

/** Offline-first: the UI only ever reads Room;
 *  [refresh] copies the network into Room. */
class OfflineFirstTrailRepository @Inject constructor(
    private val api: WaypointApi,
    private val trailDao: TrailDao,
    private val clock: Clock,
) : TrailRepository {

    override fun observeTrails(): Flow<List<Trail>> =
        trailDao.observeAll().map { rows -> rows.map(TrailEntity::asModel) }

    override fun observeTrail(id: String): Flow<Trail?> =
        trailDao.observeById(id).map { it?.asModel() }

    override fun search(input: String): Flow<List<Trail>> {
        val fts = ftsQueryOf(input)
        val rows = if (fts.isEmpty()) trailDao.observeAll() else trailDao.search(fts)
        return rows.map { list -> list.map(TrailEntity::asModel) }
    }

    override suspend fun refresh(): Int {
        val now = clock.instant()
        var page = 1
        var written = 0
        while (true) {
            val response = api.getTrails(page = page, pageSize = REFRESH_PAGE_SIZE)
            trailDao.upsertAll(
                response.items.map { it.asModel().asEntity(fetchedAt = now) },
            )
            written += response.items.size
            if (response.items.isEmpty() || written >= response.total) break
            page++
        }
        return written
    }

    companion object {
        const val REFRESH_PAGE_SIZE = 50
    }
}
```

(Three KDoc comments and the `upsertAll` call are wrapped here to fit the page; imports are omitted.) Read it with the offline-first rules in mind.

The read functions never touch the network. `observeTrails()`, `observeTrail()`, and `search()` all return a `Flow` that comes straight from a DAO. Room re-runs those queries whenever the `trails` table changes and emits the new result. The repository only maps database entities to domain models. There is no "if online, call the API" branch anywhere in the read path.

The refresh function writes into Room and returns. `refresh()` does not return a list of trails for the UI to show. It downloads pages from the API, converts each `NetworkTrail` to a domain `Trail` and then to a `TrailEntity`, and upserts the rows into the database. The UI does not need to know that a refresh happened: the `Flow` it is already collecting emits again because the table changed. That is the whole trick. The network result reaches the screen *through* the database.

Time is injected. `fetchedAt` records when each row was downloaded. The value comes from an injected `java.time.Clock` rather than from `Instant.now()`, so a test can use `Clock.fixed(...)` and assert exact values. Every repository in this book follows the same rule.

The loop handles paging on the server side. The API returns trails in pages with a `total` count. The loop requests pages of 50 until it has written `total` rows or receives an empty page. With the eight sample trails, that is a single request. Chapter 18 replaces this "download everything" approach for very large catalogs with Paging's `RemoteMediator`, which downloads pages as the user scrolls.

Errors propagate. If the network fails, `api.getTrails` throws (an `IOException` for a dropped connection, an `HttpException` for an error status), and so does `refresh()`. Crucially, nothing in the database is deleted or changed by a failed refresh, so the screen keeps showing the cached trails. The caller decides how to report the failure.

That caller is `TrailsViewModel`, which turns the outcome into a short message:

feature/trails/.../feature/trails/TrailsViewModel.kt (excerpt)

```
fun refresh() {
    viewModelScope.launch {
        refreshing.value = true
        message.value = runCatching { repository.refresh() }
            .fold(
                onSuccess = { "Updated $it trails" },
                onFailure = { "Offline - showing saved trails" },
            )
        refreshing.value = false
    }
}
```

(The starter writes the `fold` call on one line; we split it here for the page.) "Offline - showing saved trails" is the entire user-visible cost of having no signal on the trail list. The list itself never disappears.

The starter proves this behavior with a test that simulates the hike without signal. It refreshes once successfully, tells the fake API to fail the next call, refreshes again, and checks that the cache still answers queries: