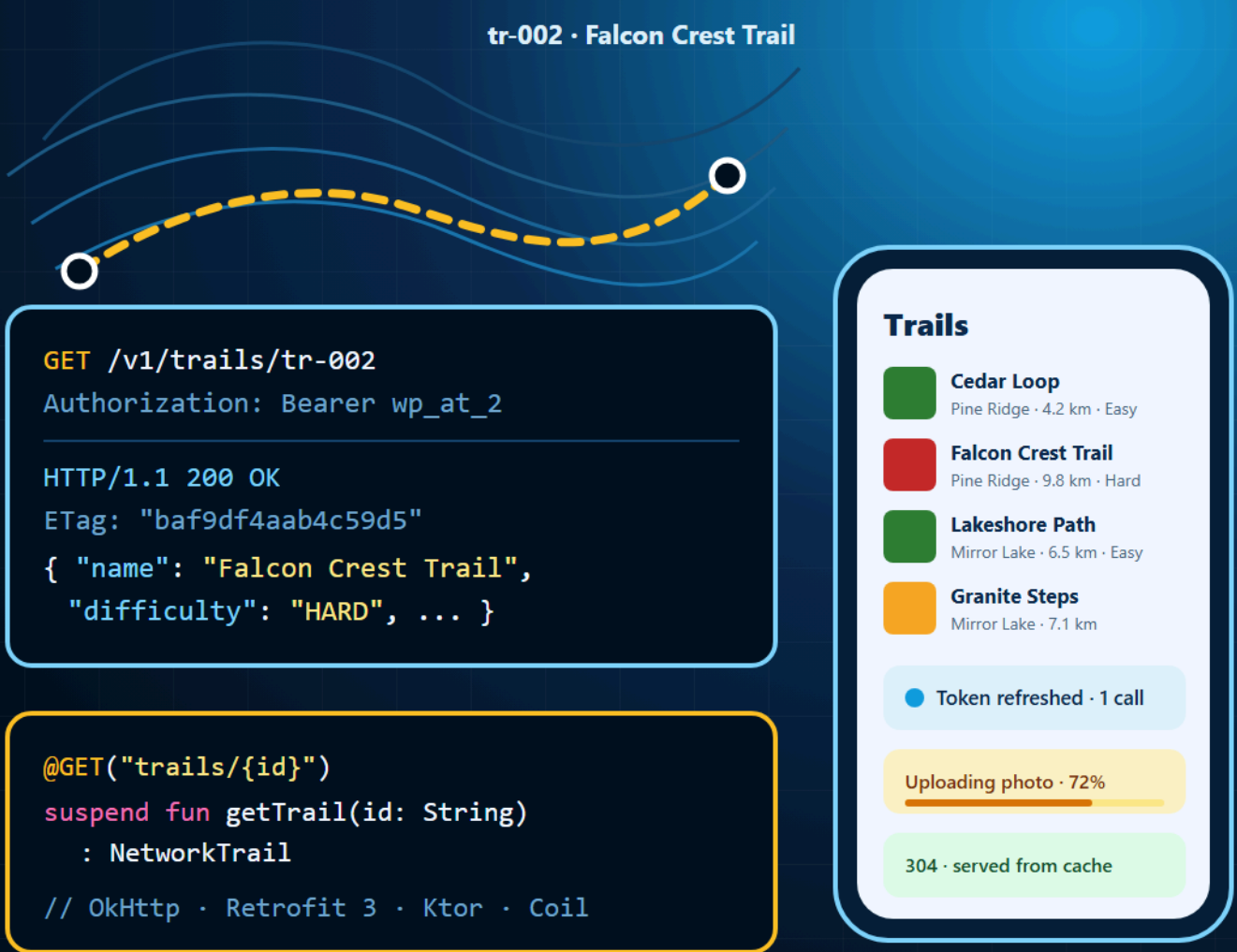


A PRACTICAL, IN-DEPTH GUIDE

Android Networking with Retrofit, Ktor, and OkHttp

REST Clients, Serialization, Authentication and Token Refresh, Caching, Errors, Uploads, Testing

HTTP · Interceptors · kotlinx.serialization · NetworkResult · Authenticator
ETags · Paging 3 · Multipart · WebSockets · SSE · MockWebServer

KOTLIN · ANDROID · WAYPOINT RUNNING EXAMPLE

THENEXBOOKS

Android Networking with Retrofit, Ktor, and OkHttp

REST Clients, Serialization, Authentication and Token Refresh, Caching, Errors, Uploads, and Testing
Build a production-quality network layer in Kotlin, one tested feature at a time

First Edition

THENEXBOOKS

Android Networking with Retrofit, Ktor, and OkHttp
REST Clients, Serialization, Authentication and Token Refresh, Caching, Errors, Uploads, 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. Libraries, the Android platform, and build tools change frequently; APIs, defaults, and behaviors can differ between versions of OkHttp, Retrofit, Ktor, kotlinx.serialization, Coil, the Android Gradle Plugin, and Android itself. Always consult the current documentation for the versions you use, and test security-sensitive code such as authentication, token refresh, and certificate pinning thoroughly before shipping it.

Android, Google Play, and Android Studio are trademarks of Google LLC. Kotlin, Ktor, OkHttp, Retrofit, Okio, Coil, GraphQL, Apollo Kotlin, and all other product, project, and company names mentioned, including debugging proxies, 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, Square, or any other company or open-source project mentioned.

Waypoint, the hiking app used as the running example in this book, is fictional, as are Northbeam Labs, its team members, the Waypoint API and its domains, the trails, regions, coordinates, user accounts, tokens, and all sample data. Domains use the reserved `.example` top-level domain and email addresses use `example.com`. Any resemblance to real companies, people, or places is coincidental.

Preface

Almost every Android app is a client for something. Trail maps, bank balances, chat messages, product catalogs, workout plans: the data lives on a server, and the app's job is to fetch it, show it, change it, and keep it in sync, quickly and safely, on a phone that drops from 5G to nothing at the trailhead. Getting a first request working takes a dozen lines of Kotlin. Getting networking *right* (typed clients, JSON that survives a changing server, errors the user understands, tokens that refresh safely when ten requests fail at once, caching, uploads that report progress, real-time updates, and tests you can trust) is what separates a demo from a product.

This book is about that second part. We build the complete network layer of one realistic app with the three libraries most Android teams reach for: **OkHttp** as the HTTP engine, **Retrofit** for declarative API interfaces, and the **Ktor client** as a Kotlin-first alternative, all with **kotlinx.serialization** for JSON. Every chapter adds a real capability, explains the HTTP behavior underneath it, and shows real traffic and real test output, so you understand not only which API to call but what happens on the wire when you do.

Who This Book Is For

- **Android developers** who can build an app with Kotlin and Jetpack Compose and now need to integrate seriously with REST APIs instead of copying fragile snippets.
- **Developers maintaining a Retrofit codebase** who want to understand interceptors, authenticators, caching, and error handling well enough to fix the hard bugs: duplicate token refreshes, stale data, silent retries, and leaking logs.
- **Teams evaluating Ktor** for Android or Kotlin Multiplatform who want a side-by-side comparison with Retrofit on the same API.
- **Team leads and backend developers** who want a reference architecture for an app's network layer, or want to understand how Android clients consume the APIs they build.

You should already be comfortable with Kotlin, the basics of coroutines, and the basics of Compose. We do not re-teach them; we use them.

What You Will Learn

- How HTTP really works for an app: methods, idempotency, status codes, headers, and caching semantics, seen through real requests and responses.
- How to use OkHttp well: one shared client, interceptors, logging with redaction, timeouts, connection reuse, event listeners, and the retries OkHttp performs on its own.
- How to model JSON with kotlinx.serialization: DTOs separate from domain models, defaults, dates, unknown keys, and unknown enum values.

- How to build typed API clients with Retrofit 3 and with the Ktor client, map failures into a `NetworkResult` type, and keep exceptions away from the UI.
- How to sign users in, attach bearer tokens, and refresh them safely with OkHttp's `Authenticator` and Ktor's `Auth` plugin, including single-flight refresh with rotating refresh tokens.
- How to secure traffic with HTTPS, the Network Security Configuration, and certificate pinning.
- How to make the app fast and frugal: HTTP caching with ETags, pagination with Paging 3, multipart uploads with progress, batched uploads, streaming and resumable downloads, and image loading with Coil.
- How to stay resilient and live: connectivity awareness, retries with exponential backoff and jitter, WebSockets, and Server-Sent Events.
- How to test all of it on the JVM with MockWebServer, Ktor's MockEngine, and a fake server, plus GraphQL basics and production debugging.

How This Book Is Organized

Part I — HTTP and OkHttp Foundations (Chapters 1–5) covers HTTP, the Waypoint API and its fake server, OkHttp essentials, interceptors, client configuration, and JSON with kotlinx.serialization. **Part II — Typed API Clients with Retrofit and Ktor** (Chapters 6–10) builds the Retrofit and Ktor clients, the repository layer, error handling with `NetworkResult`, and coroutine patterns. **Part III — Authentication and Security** (Chapters 11–15) covers login and bearer tokens, token refresh with OkHttp and Ktor, concurrency-safe refresh, and transport security. **Part IV — Data Transfer and Performance** (Chapters 16–20) covers HTTP caching, pagination, uploads, downloads, and images. **Part V — Resilience and Real-Time** (Chapters 21–24) covers connectivity, retries, WebSockets, and Server-Sent Events. **Part VI — Testing, Tooling, and Production** (Chapters 25–28) covers testing, GraphQL, debugging and observability, and a final architecture review. The appendices provide a networking cheat sheet, a complete API reference with code templates, a production checklist, and a glossary.

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

The Running Example: Waypoint

Throughout the book we build the network layer of **Waypoint**, a hiking trip planner and tracker made by **Northbeam Labs**, a small fictional product studio. Waypoint lets hikers browse and search trails such as Cedar Loop and Falcon Crest Trail, plan trips, record a hike as a stream of GPS track points, keep a private journal with photos, follow a trip live, and get trail-condition updates. It talks to the fictional Waypoint API at `https://api.waypoint.example/v1/`, which is specified exactly in Chapter 1 and Appendix B, down to every endpoint, header, and JSON field. Waypoint is also the shared running example of the thenexbooks Android series, so its models and sample data look the same in every book; this book is complete on its own.

To make every example runnable without a real backend, the book's starter project includes a **fake Waypoint server**: a deterministic implementation of the whole API that runs inside JVM tests on top of OkHttp's MockWebServer, and also plugs into Ktor's MockEngine and into the debug build of the app. Its clock is fixed at 7:30 AM UTC on Saturday, May 16, 2026, and its tokens and IDs count up predictably, so the responses you see match the book. It supports sign-in with refreshable tokens, uploads, ranged downloads, Server-Sent Events, a WebSocket, a small GraphQL endpoint, and fault injection for testing errors, timeouts, and retries. Every networking example in this book runs as a plain unit test with `./gradlew test`; no emulator, device, or server is required.

Versions and Conventions

At the time of writing, the code uses **Kotlin 2.4**, the **Android Gradle Plugin 9.4** with Gradle 9.8, **JDK 21**, compile SDK 37, target SDK 36, and a minimum SDK of 26, together with **OkHttp 5.5**, **Retrofit 3.0** with its kotlinx.serialization converter, **kotlinx.serialization 1.11**, **kotlinx.coroutines 1.11**, the **Ktor client 3.6**, **Coil 3.6**, and **Paging 3.5**. Libraries evolve quickly; if a signature or default looks slightly different in your version, the concepts still apply, so check the library's documentation for the exact spelling. All timestamps in the API are UTC.

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

Kotlin code, JSON, and HTTP traffic appear in `monospace`. A caption above a listing names the file it belongs to, such as `core/network/.../auth/Auth.kt`; terminal commands are captioned **Terminal**, raw traffic **HTTP exchange**, and results **Test output**. Long lines are wrapped to fit the page, and long JSON is sometimes shortened with an ellipsis (…), which the text always points out. MockWebServer listens on a random free port, shown in this book as `localhost:52114`; yours will differ. Output from the app on a device or emulator is shown as "you should see something like", because details such as timings vary from run to run.

Contents

Preface

PART I — HTTP AND OKHTTP FOUNDATIONS

1 How Android Apps Talk to Servers: HTTP, REST, and the Waypoint API

2 OkHttp Essentials: Clients, Requests, Responses, and the Fake Waypoint Server

3 Interceptors: Default Headers, Logging, and the Request Pipeline

4 Configuring OkHttp: Timeouts, Connection Pools, Dispatchers, and Event Listeners

5 JSON with kotlinx.serialization: DTOs, Defaults, Dates, and Unknown Values

PART II — TYPED API CLIENTS WITH RETROFIT AND KTOR

6 Retrofit 3 Fundamentals: Service Interfaces, Converters, and Suspend Functions

7 Designing the Waypoint API Layer: DTOs, Mappers, Repositories, and Dependency Injection

8 Errors as Values: HTTP Errors, Error Bodies, and NetworkResult

9 The Ktor Client: Engines, Plugins, and the Same API in Ktor

10 Networking with Coroutines: Dispatchers, Cancellation, Parallel Calls, and Flows

PART III — AUTHENTICATION AND SECURITY

11 Authentication Fundamentals: Login, Bearer Tokens, and Session State

12 Refreshing Tokens with OkHttp's Authenticator

13 Concurrency-Safe Token Refresh: Single-Flight Renewal and Rotating Refresh Tokens

14 Authentication in Ktor: The Auth Plugin and Bearer Refresh

15 Transport Security: HTTPS, Network Security Configuration, and Certificate Pinning

PART IV — DATA TRANSFER AND PERFORMANCE

16 HTTP Caching with OkHttp: Cache-Control, ETags, and Conditional Requests

17 Pagination: Page-Based APIs, Paging 3, and the Trails List

18 Uploads: Multipart Journal Photos, Progress, and Batched Track Points

19 Downloads and Streaming Bodies: GPX Tracks, Progress, and Resumable Transfers

20 Loading Images with Coil 3: Sharing OkHttp, Caching, and Compose

PART V — RESILIENCE AND REAL-TIME

21 Connectivity Awareness: Network Callbacks, Metered Networks, and Offline UX

22 Retries and Backoff: Idempotency, Retry-After, and Exponential Backoff with Jitter

23 Live Trips with WebSockets: OkHttp and Ktor

24 Server-Sent Events and Streaming Updates

PART VI — TESTING, TOOLING, AND PRODUCTION

25 Testing the Network Layer: MockWebServer, Ktor MockEngine, and the Fake Waypoint Server

26 GraphQL in Brief: Queries over Plain HTTP and When to Adopt a Client Library

27 Debugging and Observability: Logs, Event Listeners, Proxies, and Privacy

28 Case Study: Shipping Waypoint's Network Layer — Architecture Review and Production Checklist

APPENDICES

A Networking Cheat Sheet

B Waypoint API Reference and Code Templates

C Production Networking Checklist

D Glossary

PART I

HTTP and OkHttp Foundations

What actually happens when an Android app talks to a server, the Waypoint API and the fake server that answers every example, OkHttp's clients, calls, and interceptors, configuring timeouts and connections, and modeling JSON with kotlinx.serialization.

1 · How Android Apps Talk to Servers

2 · OkHttp Essentials

3 · Interceptors

4 · Configuring OkHttp

5 · JSON with kotlinx.serialization

CHAPTER 1 · PART I — HTTP AND OKHTTP FOUNDATIONS

How Android Apps Talk to Servers: HTTP, REST, and the Waypoint API

Every list of trails, every login, and every uploaded photo in a modern Android app is an HTTP exchange underneath. In this chapter we take one real exchange apart byte by byte, build the vocabulary the rest of the book relies on, meet the Waypoint API, and tour the starter project in which you will run every example on your own machine without a backend or an emulator.

Meet Waypoint and Northbeam Labs

Throughout this book we work on one app: **Waypoint**, a hiking trip planner and tracker built by a small fictional company called **Northbeam Labs**. Waypoint lets hikers browse trails, plan a trip for a date and a party size, record their track while they walk, and keep a journal with photos. A paid tier, Waypoint Plus, adds extras such as offline trail packs. The app's package is `com.northbeam.waypoint`.

Four people at Northbeam Labs appear now and then: **Lena Ortiz** leads Android development; **Sam Okafor** is the backend engineer who "owns" the Waypoint API; **Priya Natarajan** works on product and design and cares about what the user sees when the network misbehaves; and **Theo Brandt** runs QA and is the reason the network layer has so many tests. They are characters in our story who help us frame trade-offs.

Our job in this book is to build Waypoint's complete **network layer**: the code between the screens and the internet. That layer turns "show me the trails in Mirror Lake" into an HTTP request, turns the JSON answer into Kotlin objects, attaches the user's credentials, refreshes them when they expire, caches what can be cached, uploads photos with progress, survives flaky connections, streams live updates, and does all of that in a way you can test. We use three libraries that cover most Android networking today: **OkHttp** as the HTTP engine, **Retrofit** for typed REST clients on top of OkHttp, and the **Ktor client** as a Kotlin-first alternative that also runs on other platforms.

What Happens When Waypoint Loads a Trail

Suppose a hiker taps *Falcon Crest Trail* in the list. Waypoint needs the trail's details, so the app asks for `https://api.waypoint.example/v1/trails/tr-002`. From the app's point of view this is one function call that suspends for a moment and returns a `Trail`. Underneath, several distinct steps happen, and each one can fail in its own way. Knowing them by name is what lets you read an error message and know where to look.

Loading one trail (simplified sequence)



DNS resolution turns the host name into one or more IP addresses. A device can get both IPv6 and IPv4 answers, and the client tries them in turn. If DNS fails, you usually see an `UnknownHostException`, which on a phone most often means "no usable network right now" rather than "the server is gone".

TCP opens a reliable byte stream to the server's port, 443 for HTTPS. A failure here is a `ConnectException` or a connect timeout: the address was known, but nothing answered in time.

TLS wraps that stream in encryption. The client checks that the server's certificate is valid for `api.waypoint.example` and chains up to a trusted certificate authority, and both sides agree on keys. During the same handshake they also agree on the HTTP version through an extension called ALPN. Failures here are `SSLHandshakeException` or, if you pin certificates as we do in Chapter 15, `SSLPeerUnverifiedException`.

HTTP is the conversation on top: the client sends a request message, the server returns a response message. This is the layer we spend most of the book in, because it is where your code has the most influence: which method, which headers, which body, how to interpret the status code.

Finally, the client does not tear everything down after one response. Steps 1 to 3 are expensive compared with sending a few hundred bytes of request, so OkHttp keeps the connection in a **pool** and reuses it for the next request to the same host. Chapter 4 proves this with real connection numbers from the fake server. With HTTP/2, one connection can even carry many requests at once.

> **NOTE**
>
> OkHttp speaks HTTP/1.1 and HTTP/2. It does not implement HTTP/3, which runs over QUIC instead of TCP. If you need HTTP/3 on Android, Cronet (the Chromium network stack packaged for apps) is the usual option; we mention it again in Chapter 4 but do not use it in this book.

Anatomy of an HTTP Message

Enough diagrams. Here is the real exchange for that tap, captured from the starter project's test setup. The request was sent by Waypoint's own OkHttp client to the fake Waypoint server running on your machine, which is why the host is `localhost` and not `api.waypoint.example`. The port is chosen at random each time a test starts; we print it as `52114` throughout the book, and yours will differ.

HTTP exchange

```
GET /v1/trails/tr-002 HTTP/1.1
Accept: application/json
User-Agent: Waypoint/1.0.0 (Android)
Host: localhost:52114
Connection: Keep-Alive
Accept-Encoding: gzip

HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Cache-Control: public, max-age=60
ETag: "baf9df4aab4c59d5"
Content-Length: 206

{
  "id": "tr-002",
  "name": "Falcon Crest Trail",
  "region": "Pine Ridge",
  "distanceKm": 9.8,
  "elevationGainM": 640,
  "difficulty": "HARD",
  "rating": 4.8,
  "photoUrl": "https://api.waypoint.example/v1/images/trails/tr-002.png"
}
```

We pretty-printed the JSON body for the page; on the wire it is a single line of exactly 206 bytes, which is what `Content-Length` announces. Everything else is shown as it was sent.

The request

The first line is the **request line**: a **method** (`GET`), a **request target** (the path plus any query string, `/v1/trails/tr-002`), and the protocol version. Then come **header fields**, one per line, as `Name: value`. Header names are case-insensitive. A blank line ends the headers; a `GET` has no body, so the request ends there.

Two of these headers come from Waypoint's code. `Accept: application/json` tells the server which representation we want, and `User-Agent: Waypoint/1.0.0 (Android)` identifies the app and its version, which Sam's server logs use to tell which app versions are still in the wild. Both are added by a small interceptor we write in Chapter 3. The other three are added by OkHttp itself: `Host` (required by HTTP/1.1 because one IP address can serve many sites), `Connection: Keep-Alive` (please keep the

connection open), and `Accept-Encoding: gzip`, which offers to receive a compressed body. When OkHttp adds that header itself, it also decompresses the response for you transparently, so your code never sees gzip bytes.

The response

The response starts with a **status line**: version, a three-digit **status code** (`200`), and a human-readable reason phrase (`OK`). Code should only ever look at the number; reason phrases are informational and HTTP/2 does not even transmit them. Then headers, a blank line, and the **body**.

The response headers already tell a story. `Content-Type` says the body is JSON encoded as UTF-8. `Cache-Control: public, max-age=60` says any cache may keep this answer and consider it fresh for 60 seconds. `ETag` is a version label for this exact representation of the trail; later, the client can ask "has it changed since `"baf9df4aab4c59d5"`?" and get a tiny `304 Not Modified` instead of the whole body. Chapter 16 builds on exactly these two headers.

> **TIP**
> When something goes wrong in networking code, start by looking at the raw exchange: method, URL, status, headers, body. Most bugs are visible there in seconds. Chapter 3 shows how to log exchanges safely, and Chapter 27 builds a whole debugging toolbox around this habit.

Methods, Safety, and Idempotency

HTTP semantics are defined in RFC 9110. Two properties of methods matter enormously for app developers, because they decide what a client may do automatically when something fails.

A method is **safe** if it is read-only in intent: the client is not asking the server to change anything. `GET` and `HEAD` are safe. A method is **idempotent** if sending the same request twice has the same effect on the server as sending it once. All safe methods are idempotent, and so are `PUT` and `DELETE`: putting the same trip state twice leaves the trip in that state; deleting something twice leaves it deleted. `POST` is neither: posting "create a trip" twice can create two trips.

Method	Typical meaning	Safe	Idempotent	Waypoint example
`GET`	Read a resource	Yes	Yes	`GET trails/tr-002`
`HEAD`	Headers only	Yes	Yes	(not used)
`POST`	Create or process	No	No	`POST trips`
`PUT`	Replace a resource	No	Yes	`PUT trips/tp-1001`
`PATCH`	Partial update	No	Not by default	(not used)
`DELETE`	Remove a resource	No	Yes	(not used)

Why does this matter? Mobile connections drop, and the dangerous moment is when the request was sent but the response never arrived. For a `GET` or a `PUT`, just send it again; for a `POST`, you might create a second trip. This distinction drives the retry rules of Chapter 22 and explains an OkHttp behavior we observe in Chapter 4.

> **WARNING**
> Idempotency is a promise the *server* makes, not something the method name enforces. A badly designed API can make a `GET` that changes data or a `PUT` that appends. Before you add automatic retries to any call, confirm with whoever owns the API — at Northbeam Labs, Sam — that the endpoint really behaves the way its method promises.

Status Codes

Status codes come in five classes, and the first digit tells you most of what a client needs to know:

- **1xx informational** — interim responses; rare in app code (the WebSocket upgrade in Chapter 23 uses `101 Switching Protocols`).
- **2xx success** — the request worked. `200 OK`, `201 Created`, `202 Accepted`, `204 No Content`, `206 Partial Content`.
- **3xx redirection** — look elsewhere, or use your cached copy (`304 Not Modified`). OkHttp follows normal redirects automatically.
- **4xx client error** — the request was wrong in some way: bad input, missing credentials, unknown resource. Repeating the same request will usually fail the same way.
- **5xx server error** — the server failed. These are often temporary and are the classic candidates for a retry, carefully.

The Waypoint API uses a deliberate subset. Learning it now will make the error handling chapters read like a story you already know.

Code	Meaning in Waypoint	Where you meet it
200	Normal success with a body	Most reads; login; refresh
201	Created; `Location` header points to it	`POST trips`, journal
202	Accepted for processing	Track point upload
206	Partial content for a `Range`	GPX downloads (Ch 19)
304	Not modified since your `ETag`	Caching (Ch 16)
400	Malformed request (`invalid_request`)	Bad JSON, page size 500
401	Not authenticated	Missing or expired token
404	No such resource	`trail_not_found`

413 / 415	Too large / unsupported media type	Uploads (Ch 18)
422	Well-formed but invalid input	`validation_failed`
429	Rate limited; see `Retry-After`	Retries (Ch 22)
500 / 502 / 503 / 504	Server-side failures	Fault injection in tests

Two pairs are worth separating early. **400 versus 422**: Waypoint answers 400 when it cannot even understand the request (the body is not JSON) and 422 when it understands it but the content breaks a rule, such as a party size of 20. **401 versus 403**: 401 means "I do not know who you are", so signing in or refreshing a token may help; 403 means "I know who you are, and you may not do this", so new credentials will not help. Chapter 11 returns to that distinction.

One more point that surprises people: a 404 or a 500 is a *successful HTTP exchange*. The network worked perfectly; the server simply said no. At the OkHttp level such responses do not throw. Whether your app turns them into exceptions or into values is a design decision, and Chapter 8 makes it deliberately.

Headers That Matter

HTTP defines many headers; a mobile client needs to understand perhaps a dozen well. These are the ones you will see in almost every chapter:

Header	Direction	What it does
`Content-Type`	Both	Media type of the body, e.g. JSON, multipart, PNG
`Accept`	Request	Media types the client wants back
`Authorization`	Request	Credentials, here `Bearer <token>`
`WWW-Authenticate`	Response	How to authenticate after a 401
`Cache-Control`	Both	Freshness and storage rules
`ETag` / `If-None-Match`	Resp. / Req.	Version label and conditional request
`Location`	Response	URL of a newly created resource (201)
`Retry-After`	Response	How long to wait before trying again
`Range` / `Content-Range`	Req. / Resp.	Partial downloads and resumption

Waypoint uses `Retry-After` in its simplest form, a number of seconds: a 503 comes with `Retry-After: 5`, a 429 with `Retry-After: 2`. The standard also allows an HTTP date instead of seconds; our helper in Chapter 22 handles only the seconds form, and we say so there.

> **NOTE**
>
> Headers can carry secrets. `Authorization` holds a bearer token that is as good as a password until it expires. Any log you write, any crash report you send, and any screenshot you paste into a bug tracker should hide it. The logging setup in Chapter 3 prints it as `Authorization: ` ▮▮ .

REST Resources and the Waypoint API

"REST" gets used loosely. For our purposes it means an API organized around **resources** with stable URLs (a trail, a trip, the signed-in user), manipulated with standard HTTP methods, exchanging **representations** of those resources (JSON documents), and relying on HTTP's own machinery for status, caching, and content types instead of reinventing them inside the body. Waypoint's API is a pragmatic example of that style. Its base URL is `https://api.waypoint.example/v1/`; the `v1` segment is the API version, and Chapter 28 discusses how versioning and backward compatibility interact with apps that users update slowly.

Here is the whole API at a glance. Paths are relative to the base URL. "Bearer" means the call needs a signed-in user's access token.

Method and path	Auth	Result
`GET trails` (query, region, page, pageSize)	–	200 page of trails, cacheable
`GET trails/{id}`	–	200 trail; 404 `trail_not_found`
`GET trails/{id}/track.gpx`	–	200 GPX file; 206 for a range
`GET trails/{id}/conditions`	–	200 event stream
`GET images/trails/{id}.png`	–	200 small PNG
`POST auth/login`	–	200 tokens; 401 `invalid_credentials`
`POST auth/refresh`	–	200 rotated tokens
`GET me`	Bearer	200 user; never cached
`POST trips`	Bearer	201 trip + `Location`
`PUT trips/{id}`	Bearer	200 trip; 404 `trip_not_found`
`POST trips/{id}/points`	Bearer	202 accepted count
`POST trips/{id}/journal`	Bearer	201 entry (multipart upload)
`POST graphql`	–	One `TrailCard` operation
`GET live` (WebSocket)	–	Live trip positions

Notice the shapes. Trails are a **collection** (`trails`) with **items** (`trails/tr-002`) and **sub-resources** (`trails/tr-002/track.gpx`). Creating a trip is a `POST` to the collection, and the server chooses the new id, returning it in the body and in the `Location` header. Updating a trip is a `PUT` of its full editable state to the item URL, which is why it is idempotent. The API also deliberately includes things that are not plain JSON request/response — a file download, an event stream, a WebSocket, a GraphQL endpoint — because real apps meet them all, and each has its own chapter.

The data behind the API is small and fixed, which makes every output in this book reproducible. There are eight trails, `tr-001` to `tr-008`, in three regions: Pine Ridge (Cedar Loop, Falcon Crest Trail, Summit Spur), Mirror Lake (Lakeshore Path, Granite Steps, Brookside Way), and Fernhollow (Old Mill Ridge, Meadow Run). Two users exist: `u-100` Alex Rivera (`alex@example.com`) and `u-200` Jordan Lee (`jordan@example.com`, a Waypoint Plus member). Both use the test password `waypoint-demo`. The server's clock is frozen at `2026-05-16T07:30:00Z`, a Saturday morning, so every timestamp it creates is predictable.

JSON on the Wire

Waypoint speaks JSON (RFC 8259) with a few conventions that Sam documented for the team and that the serialization chapter relies on:

- Property names are **camelCase**: `distanceKm`, `elevationGainM`, `photoUrl`.
- Timestamps are ISO-8601 in UTC, such as `2026-05-16T07:30:00Z`; dates without a time are `yyyy-MM-dd`, such as `2026-06-06`.
- Enumerations travel as strings: `"HARD"`, `"PLANNED"`.
- Absent values may be `null`. Meadow Run (`tr-006`) has no photo, so its `photoUrl` is `null`, which gives us a natural test of null handling.
- Errors share one shape: an `error` code meant for programs and a `message` meant for people.

HTTP exchange

```
GET /v1/trails/tr-999 HTTP/1.1

HTTP/1.1 404 Not Found
Content-Type: application/json; charset=utf-8

{"error":"trail_not_found","message":"No trail with id 'tr-999'."}
```

We trimmed the request headers in that excerpt; the body is exact. That stable error shape is a gift to client developers. Code can branch on `trail_not_found` or `validation_failed` without parsing English, and the message can be shown to the user when it is meant for them, as with the 422 validation messages. An alternative standard format for error bodies exists, "problem details" (RFC 9457), and Chapter 8 compares it with Waypoint's simpler shape.

A point of design we will defend repeatedly: the JSON shape belongs to the API, not to your app. The app's own model classes — `Trail`, `Trip`, `User` — live in a module with no serialization annotations at all. Separate **DTO** classes (data transfer objects) mirror the JSON, and small mapper functions convert between the two. Chapter 5 explains why that extra layer pays for itself the first time the server adds a field or a new enum value.

Why We Test Against a Fake Server

`api.waypoint.example` does not exist; the `.example` domain is reserved for documentation. That is deliberate. A book whose examples depend on a live server breaks when the server changes, goes down, or rate-limits its readers. Instead, the starter project contains a complete **fake Waypoint backend** that implements every endpoint in the table above, with the frozen clock, the fixed trails, and predictable ids and tokens.

The fake is one class, `FakeWaypointBackend`, that knows nothing about sockets. It accepts a simple request object and returns a simple response object. Three thin **transports** expose it to clients in different ways:

- `FakeWaypointServer` serves it over **real HTTP** on `localhost`, using OkHttp's MockWebServer. Your client opens real TCP connections, so connection pooling, timeouts, and broken connections behave the way they do on a phone. Most tests in this book use it.
- `fakeWaypointEngine()` plugs it into the Ktor client as a `MockEngine`, with no sockets. Chapter 9 uses it.
- `FakeWaypointInterceptor` answers inside OkHttp's interceptor chain, again with no sockets. The debug build of the app uses it, so you can run Waypoint on a device without any backend at all.

The fake can also misbehave on command. A test can say "the next request gets a 503", "the next request is answered after 1.5 seconds", "the next connection is closed without a response", or "the next body is truncated JSON". These **faults** are how we produce real timeouts, real broken connections, and real parse errors on demand, instead of describing them in the abstract.

Everything runs on the **JVM**, in ordinary unit tests; no emulator or device is needed. The few things that really need Android at runtime, such as images drawn on a screen, are clearly marked, and their output is described as "you should see something like". JVM tests start in seconds and give the same answer every time.

> **TIP**
>
> A good fake is worth more than a test double per test. Because Waypoint's fake implements the real contract — status codes, headers, error bodies, token rotation — a test that passes against it says something meaningful about the client. Chapter 25 shows how the fake is designed and why it deliberately does not share DTO classes with the app.

The Toolchain

The starter project and every example were built and tested with the following versions, at the time of writing. The ideas in this book are stable across versions; exact API names occasionally move, and we point out the places where that is likely.

Component	Version (at the time of writing)
Kotlin	2.4.20
Android Gradle Plugin / Gradle	9.4.1 / 9.8.0
JDK (bytecode target)	21 (17)
compileSdk / targetSdk / minSdk	37 / 36 / 26
OkHttp	5.5.0
Retrofit (+ kotlinx converter)	3.0.0
kotlinx.serialization / coroutines	1.11.0 / 1.11.0
Ktor client	3.6.0
Coil / Paging	3.6.3 / 3.5.1
Compose BOM / Hilt	2026.09.00 / 2.60.1

We assume you are comfortable with Kotlin, coroutine basics, and Jetpack Compose basics. We do not re-teach them; when we use something less common, such as `callbackFlow` or virtual time in `runTest`, we explain it where it appears. The `minSdk` of 26 lets the app use `java.time.Instant` and `java.time.LocalDate` directly, which keeps date handling simple.

The INTERNET Permission

An Android app cannot open a network socket at all unless its manifest declares the `INTERNET` permission. It is a "normal" permission: the system grants it at install time without asking the user, but if you forget it, every connection attempt fails, typically with an exception whose message hints at a missing permission or a denied socket operation, which can be confusing the first time you see it. Waypoint also declares `ACCESS_NETWORK_STATE`, which Chapter 21 needs to observe connectivity.

app/src/main/AndroidManifest.xml (excerpt)

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android">

    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />

    <application
        android:name=".WaypointApplication"
        android:label="Waypoint"
```

```
        ...>
    </application>
</manifest>
```

The `core:network` library module declares the same two permissions in its own manifest, and the manifest merger copies them into the app. Declaring a permission in the library that needs it is a good habit: the requirement travels with the code.

A related default: apps that target API 28 or higher may not use cleartext `http://` by default. Production traffic should be HTTPS anyway. Our JVM tests are not affected, because Android's network security policy is not enforced in plain unit tests, but if you ever point a device build at a local `http://` server, you need a debug-only network security configuration. Chapter 15 sets one up for `localhost` and the emulator's host alias `10.0.2.2`.

Tour of the Starter Project

The starter project lives in the book's download as a normal Gradle project named `waypoint-network`. Open it in Android Studio or work from a terminal. Its modules follow a common modern layout:

Project structure

```
waypoint-network/
├── app             Hilt application, MainActivity + navigation, di/NetworkModule,
│                   DebugBackend (debug: fake backend in-process; release: none)
├── core/model      Pure Kotlin domain types: Trail, Difficulty, Trip, TripStatus,
│                   TrackPoint, JournalEntry, User
├── core/network    OkHttp, Retrofit, kotlinx.serialization, Ktor: WaypointApi,
│                   AuthApi, DTOs + mappers, NetworkResult, interceptors, auth,
│                   uploads, retries, realtime (SSE + WebSocket), connectivity
├── core/data       Repositories: DefaultTrailRepository (+ paging), AuthRepository
├── core/testing    Pure JVM: FakeWaypointBackend and its three transports
└── feature/trails  TrailsViewModel and the Compose TrailsScreen
```

Dependencies point in one direction: `feature:trails` depends on `core:data`, which depends on `core:network`, which depends on `core:model`. The `app` module wires everything together with Hilt. `core:testing` stands apart: it depends only on OkHttp, MockWebServer, Ktor's mock engine, and kotlinx.serialization, never on `core:network`. That separation mirrors reality, where the server team does not share classes with the app, and it means a bug in a DTO cannot hide itself by being wrong on both sides.

The project contains the *final* version of every class the book builds. Chapters sometimes start with a simpler version and evolve it; the version a chapter ends with always matches the project.

Before going further, run the whole test suite once. From the project folder:

Terminal

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