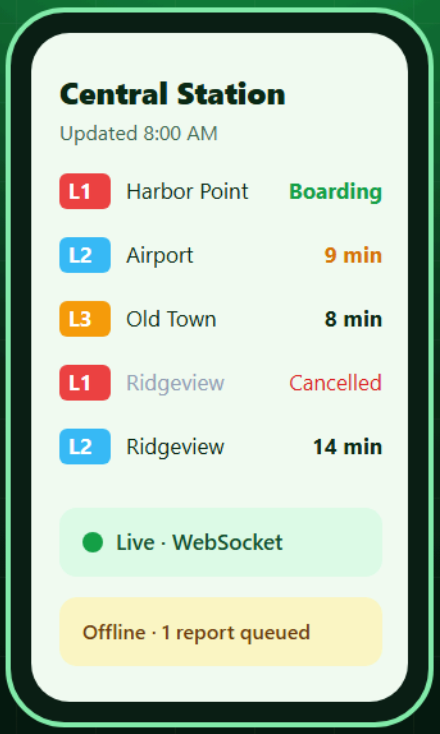



```
GET /v1/stations/CEN/departures
If-None-Match: "fdbc68956c13385e"

HTTP/1.1 200 OK
Cache-Control: max-age=15
{ "id": "D-1002",
  "status": "delayed", ... }
```

```
let board = try await client.send(
   .departures(stationID: "CEN"))
// retries · ETags · token refresh
// Swift 6 · Sendable · actors
```



A PRACTICAL, IN-DEPTH GUIDE

iOS Networking and API Integration

URLSession, Codable, Authentication, Real-Time Data, and Offline-First Apps

HTTP · async/await · Typed API Client · Caching · OAuth + PKCE
Keychain · WebSockets · SSE · Push · Sync Queue · Testing

SWIFT 6 · iOS 18+ · RAILBIRD RUNNING EXAMPLE

THENEXBOOKS

iOS Networking and API Integration

URLSession, Codable, Authentication, Real-Time Data, and Offline-First Apps
Build a production-quality API layer in Swift 6, one real feature at a time

First Edition

THENEXBOOKS

iOS Networking and API Integration
URLSession, Codable, Authentication, Real-Time Data, and Offline-First Apps

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. Apple platforms, SDKs, and developer tools change frequently; APIs, build settings, and behaviors can differ between versions of iOS, Xcode, and Swift. Always consult Apple's current documentation for the versions you use, and test security-sensitive code such as authentication and certificate pinning thoroughly before shipping it.

Apple, iPhone, iPad, iOS, iPadOS, macOS, Mac, Swift, SwiftUI, Xcode, Instruments, Keychain, Safari, and App Store are trademarks of Apple Inc., registered in the U.S. and other countries and regions. Other product and company names mentioned, including tools such as proxies and code generators, 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 Apple Inc.

Railbird, the transit app used as the running example in this book, is fictional, as are the city of Larkhaven, its transit lines and stations, the Railbird API and its domains, the user accounts, and all sample data. Domains use the reserved `.example` top-level domain and email addresses use `example.com`.

Preface

Almost every iOS app is a client for something. Departure times, 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 device that loses its connection in tunnels and elevators. Getting a first request working takes ten lines of Swift. Getting networking *right* — typed requests, meaningful errors, retries that do not make outages worse, caching, sign-in with refreshable tokens, live updates, and an app that still works offline — is what separates a demo from a product.

This book is about that second part. We build the complete networking layer of one realistic app, starting from HTTP itself and finishing with an architecture you can defend in a code review. Every chapter adds a real capability, explains the protocol behavior underneath it, and shows the actual HTTP traffic, so you understand not only which API to call but what happens on the wire when you do.

Who This Book Is For

- **iOS developers** who can build a SwiftUI app and now need to integrate with real REST or GraphQL backends without copying fragile snippets.
- **Developers moving to Swift 6** who want networking code that compiles cleanly under strict data-race checking, with actors, `Sendable` types, and structured concurrency.
- **Team leads and senior engineers** who want a reference architecture for an app's API layer: authentication, caching, offline support, testing, and production readiness.
- **Backend and web developers** who build APIs consumed by iOS apps and want to understand how the client side behaves.

You should already be comfortable with Swift and the basics of SwiftUI (views, state, navigation). We do not re-teach them; we use them.

What You Will Learn

- How HTTP really works for an app: methods, status codes, headers, idempotency, content negotiation, and caching semantics.
- How to use `URLSession` with async/await, build requests correctly, and decode real-world JSON with Codable: custom keys, dates, unknown values, and polymorphic payloads.
- How to design a typed API client with endpoints, a single request pipeline, meaningful errors, retries with exponential backoff, cancellation, and pagination.
- How to make it fast and efficient: `URLCache`, ETags, image loading pipelines, uploads with progress, resumable and background downloads, and GraphQL over plain HTTP.
- How to sign users in with OAuth 2.0 and PKCE, keep tokens in the Keychain, refresh them safely under concurrency, and protect traffic with App Transport Security and certificate pinning.

- How to deliver live data with WebSockets, Server-Sent Events, and push notifications, and how to build an offline-first app with path monitoring, local snapshots, and a durable sync queue.
- How to test, debug, and generate networking code: `URLProtocol` mocks, Swift Testing, proxies, metrics, and the Swift OpenAPI Generator.

How This Book Is Organized

Part I — HTTP and URLSession Foundations (Chapters 1–5) covers HTTP, the Railbird API, URLSession with async/await, building requests, and Codable in depth. **Part II — A Production API Client** (Chapters 6–10) builds the typed client with errors, retries, pagination, and Swift 6 concurrency. **Part III — Performance and Data Transfer** (Chapters 11–15) covers caching, images, uploads, downloads and background sessions, and GraphQL. **Part IV — Authentication and Security** (Chapters 16–20) covers tokens, OAuth 2.0 with PKCE, the Keychain, transport security, and privacy. **Part V — Real-Time and Offline** (Chapters 21–24) covers WebSockets, Server-Sent Events, push notifications, and offline-first sync. **Part VI — Testing, Tooling, and Shipping** (Chapters 25–28) covers testing, debugging, OpenAPI code generation, 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: Railbird

Throughout the book we build **Railbird**, a departures app for the fictional city of **Larkhaven**, whose transit network has two rail lines, a tram line, and a crosstown bus. Railbird shows live departure boards for stations such as Central Station and University, lists service alerts, lets riders save favorite stations, sign in, report problems with photos, download the network map, and keep working when the connection drops. It talks to the fictional Railbird API, which is specified exactly in Chapter 1 and Appendix B, down to every endpoint, header, and JSON field.

To make every example runnable without a real backend, the book uses a small **Railbird mock server** written in Python with only the standard library. You run it on your Mac, point the app in the Simulator at `http://127.0.0.1:8080/v1`, and get deterministic responses: the data clock is fixed at 8:00 AM on Monday, March 16, 2026, so the departure board you see matches the book. The mock server also supports OAuth sign-in, uploads, resumable downloads, WebSockets, Server-Sent Events, and fault injection for testing errors and retries. The complete server, `railbird_mock_server.py`, and the OpenAPI document `railbird-openapi.yaml` are included with this book's download; Chapter 26 also prints a compact version of the server that you can type in yourself.

Versions and Conventions

The code uses the **Swift 6 language mode** with complete data-race checking, targets **iOS 18** as the minimum deployment version, and was written for **Xcode 26-era tooling**. Where an API requires iOS 26, the text says so and the code checks availability. Apple's SDKs evolve quickly; if a signature or build setting looks slightly different in your version of Xcode, the concepts still apply — check Apple's documentation for the exact spelling. All timestamps in the API are UTC; the app displays them in Larkhaven's time zone (`America/Chicago`).

> **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.

Swift code, JSON, and HTTP traffic appear in `monospace`. A caption above a listing names the file it belongs to, such as `Railbird/Networking/APIClient.swift`; terminal commands are captioned **Terminal**. Long lines are wrapped to fit the page, and long JSON is sometimes shortened with an ellipsis (…), which the text always points out. Runtime output from the app is shown as "you should see something like", because details such as timings and generated tokens vary from run to run.

Contents

Preface

PART I — HTTP AND URLSESSION FOUNDATIONS

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

2 URLSession with async/await: Requests, Responses, and Status Codes

3 Building Requests Right: URLs, Query Items, Headers, and Bodies

4 Codable in Depth: Coding Keys, Nested Containers, and Custom Decoding

5 Real-World JSON: Dates, Optional Fields, Unknown Values, and Polymorphic Payloads

PART II — A PRODUCTION API CLIENT

6 Designing a Typed API Client: Endpoint, APIClient, and Protocols

7 Errors That Make Sense: HTTP Errors, API Error Bodies, and User Messages

8 Resilient Requests: Timeouts, Retries, Exponential Backoff, and Cancellation

9 Pagination, Search, and Loading State in SwiftUI

10 Networking and Swift 6 Concurrency: Actors, MainActor Models, and Task Groups

PART III — PERFORMANCE AND DATA TRANSFER

11 HTTP Caching: URLCache, Cache-Control, and ETags

12 Loading, Caching, and Displaying Remote Images

13 Uploads: JSON, Binary Bodies, Multipart Forms, and Progress

14 Downloads, Resumable Transfers, and Background Sessions

15 GraphQL over URLSession: Queries, Variables, and Partial Errors

PART IV — AUTHENTICATION AND SECURITY

16 Authentication Fundamentals: API Keys, Sessions, and Bearer Tokens

17 Sign In with OAuth 2.0, PKCE, and ASWebAuthenticationSession

18 Tokens in the Keychain: Storage, Refresh, and Single-Flight Renewal

19 Transport Security: App Transport Security, TLS, and Certificate Pinning

20 Protecting Users on the Wire: Secrets, Logging, and Privacy

PART V — REAL-TIME AND OFFLINE

21 Live Updates with WebSockets and URLSessionWebSocketTask

22 Server-Sent Events and Streaming Responses

23 Push Notifications as a Data Trigger

24 Offline-First Railbird: Path Monitoring, Local Snapshots, and a Sync Queue

PART VI — TESTING, TOOLING, AND SHIPPING

25 Testing Networking Code: URLProtocol Mocks, Fixtures, and Swift Testing

26 Debugging Network Traffic: Proxies, Logs, Metrics, and the Mock Server

27 Generating Clients from OpenAPI with Swift OpenAPI Generator

28 Case Study: Shipping Railbird — Architecture Review and Production Checklist

APPENDICES

A Networking Cheat Sheet

B Railbird API Reference and Code Templates

C Production Networking Checklist

D Glossary

PART I

HTTP and URLSession Foundations

What actually happens when an app talks to a server, the Railbird API and its mock server, URLSession with async/await, building requests that are correct down to the last percent-encoded character, and decoding real-world JSON with Codable.

1 · How Apps Talk to Servers
2 · URLSession with async/await
3 · Building Requests Right
4 · Codable in Depth
5 · Real-World JSON

CHAPTER 1 · PART I — HTTP AND URLSESSION FOUNDATIONS

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

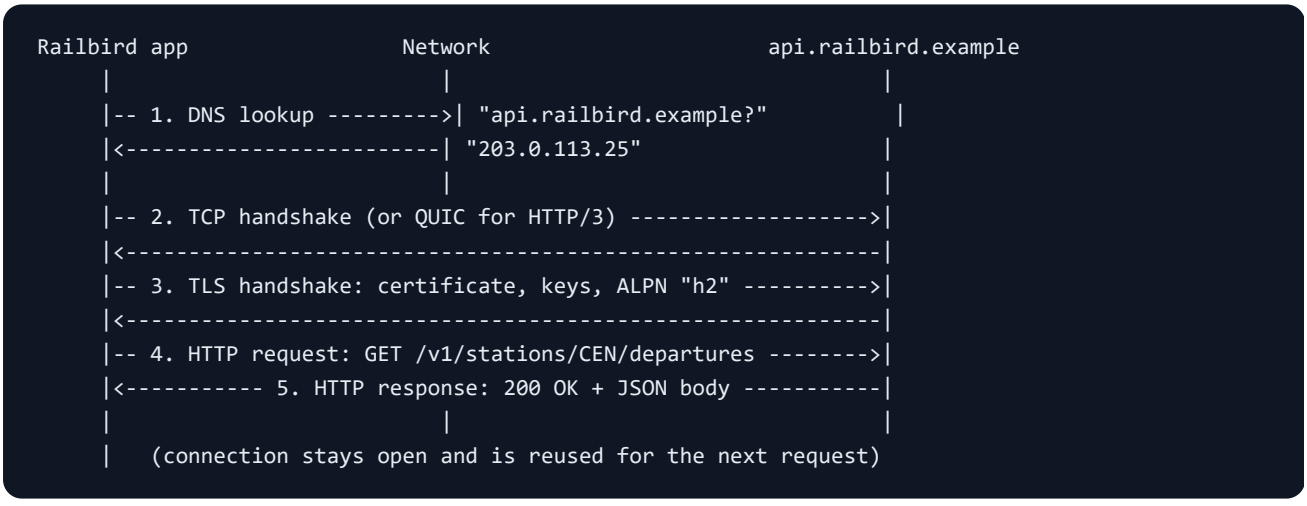
Before we write a single line of URLSession code, we look at what actually travels over the wire when an app loads data: the connection, the request, the response, and the conventions that make an API predictable. Then we meet Railbird, start its mock server on your Mac, and explore the API with nothing but `curl`.

Most networking bugs in iOS apps are not Swift bugs. They are misunderstandings of HTTP: a status code that was never checked, a query string built by hand that broke on an ampersand, a retry that created a duplicate order. URLSession is an excellent tool, but it only behaves well when we understand the protocol it speaks. That is why this book starts with the protocol, and why every chapter shows real requests and responses next to the Swift code that produces or consumes them.

What Happens When Railbird Loads a Departure Board

Picture a commuter in the fictional city of Larkhaven opening Railbird at Central Station. The screen needs the next departures, so the app asks the server for `https://api.railbird.example/v1/stations/CEN/departures`. From the app's point of view this is a single `await`. Underneath, several distinct steps happen, and each one can fail in its own way.

One request, step by step



1. Name resolution. DNS turns the host name into an IP address. The system caches answers, so most requests skip this step. When DNS fails you meet `URLError(.cannotFindHost)` or `URLError(.dnsLookupFailed)`. (The address in the diagram comes from a range reserved for documentation.)

2. Transport connection. HTTP/1.1 and HTTP/2 run over TCP, which costs a handshake round trip before any data moves. HTTP/3 runs over QUIC, a UDP-based transport that folds the transport and encryption handshakes together. You do not choose in code: URLSession negotiates with the server and, in recent iOS versions, uses HTTP/3 when the server advertises support. If nothing is listening on the port, you get `URLError(.cannotConnectToHost)`, as you will see the first time you forget to start the mock server.

3. TLS. For an `https` URL the server presents a certificate chain, the device decides whether it trusts that chain for this host name, and both sides agree on keys. In the same handshake, ALPN settles whether to speak HTTP/2 (`h2`) or HTTP/1.1. App Transport Security (Chapter 19) requires TLS by default.

4 and 5. The HTTP exchange. Only now does the app send the request it cares about and receive a response.

Two consequences follow. First, the expensive part of a fresh request is often the setup, not the payload. URLSession keeps connections open and reuses them, which is one reason to share a few long-lived sessions instead of creating one per request. Second, a failure can happen at any layer: "no network", "server unreachable", "server answered with an error", and "server sent something we cannot read" are four different situations with four different user messages. Chapter 7 builds exactly that taxonomy.

> **NOTE**
>
> The Railbird mock server you run on your Mac speaks plain HTTP/1.1 without TLS, on the loopback address `127.0.0.1`. That keeps the traffic readable and lets you inspect every byte with `curl`. The production configuration uses `https`, and the code we write works identically against both; only the base URL changes.

Anatomy of an HTTP Message

HTTP is a request–response protocol. Its semantics (methods, status codes, header meanings) are defined in RFC 9110, and the HTTP/1.1 wire format in RFC 9112. HTTP/2 (RFC 9113) and HTTP/3 (RFC 9114) encode the same messages in binary frames, but the semantics do not change, so it is perfectly reasonable to think in terms of the readable HTTP/1.1 text form. This is what `curl -v` shows for a request to the mock server's health endpoint:

HTTP request

```
GET /v1/health HTTP/1.1
Host: 127.0.0.1:8080
User-Agent: curl/8.13.0
Accept: */*
```

The **request line** holds the method, the request target (path plus query string), and the protocol version. Then come the **header fields**: `Host` names the server, `User-Agent` identifies the client, and `Accept` lists the response formats the client handles. A blank line ends the headers, and an optional **body** follows. A `GET` has no body; a `POST` that creates a report carries JSON and announces it with `Content-Type: application/json`.

The response has the same shape: a **status line**, headers, a blank line, and a body.

HTTP response

```
HTTP/1.1 200 OK
Server: RailbirdMock/1.0.0 Python/3.12.0
Date: Mon, 16 Mar 2026 13:00:00 GMT
X-Request-ID: req_000001
Content-Type: application/json; charset=utf-8
Content-Length: 76

{
  "status": "ok",
  "version": "1.0.0",
  "now": "2026-03-16T13:00:00Z"
}
```

The status line carries a three-digit **status code** and a reason phrase that clients must ignore; HTTP/2 does not even transmit the phrase. `X-Request-ID` is a Railbird convention: every response carries a unique identifier that also appears in the server's logs, so a support screen showing it turns a vague complaint into a searchable log entry. Chapter 7 keeps that ID alive through the whole error pipeline.

Header names are case-insensitive, and HTTP/2 always sends them in lowercase. `HTTPURLResponse.value(forHTTPHeaderField:)` performs a case-insensitive lookup, so use it instead of comparing names in a dictionary you built yourself.

Methods, Safety, and Idempotency

The method tells the server what the client intends to do with the resource named by the path. RFC 9110 defines two properties that matter enormously for client code. A method is **safe** if the client does not request any change on the server: reading is safe, writing is not. A method is **idempotent** if sending the same request twice has the same intended effect as sending it once. Setting a favorite with `PUT` is idempotent: after one or five identical requests, Central Station is a favorite. Creating a report with `POST` is not: five requests could create five reports.

Method	Safe	Idempotent	Railbird example
`GET`	Yes	Yes	`GET /v1/stations/CEN`
`HEAD`	Yes	Yes	Headers of the network map PDF

`POST`	No	No	`POST /v1/reports`
`PUT`	No	Yes	`PUT /v1/me/favorites/MKT`
`DELETE`	No	Yes	`DELETE /v1/me/favorites/CEN`
`PATCH`	No	Not guaranteed	Not used by Railbird

Why does an app developer care? Because networks fail in the middle of requests. Suppose the app sends `POST /v1/reports`, the server stores the report, and the connection drops before the response arrives. The app sees an error, but the server did the work; an automatic retry turns one complaint into two. With an idempotent method the same retry is harmless. That observation drives Chapter 8: Railbird retries `GET`, `PUT`, and `DELETE` automatically, and retries a `POST` only when it carries an `Idempotency-Key` header that lets the server recognize a duplicate. That header is still an IETF draft, but many APIs implement the idea, and Railbird's does.

Idempotency is about effect, not identical responses: deleting a favorite twice leaves the same server state whether the second call returns `204` or `404`. Railbird answers `204` both times.

Status Codes: Five Classes, a Few Dozen Codes

The first digit of the status code defines its class, and a client that understands the classes can handle codes it has never seen. RFC 9110 explicitly requires that: an unknown code such as `299` must be treated like `200`, and an unknown `4xx` like `400`.

Class	Meaning	Codes Railbird uses	Typical client reaction
1xx	Informational	101 (WebSocket upgrade)	Handled by URLSession
2xx	Success	200, 201, 204, 206	Decode the body (if any)
3xx	Redirection or cache	302 (OAuth), 304	Mostly handled by URLSession
4xx	Client error	400, 401, 404, 413, 415, 422, 429	Fix the request; usually no retry
5xx	Server error	500, 503	Retry safe requests with backoff

A few codes shape the whole book. `201 Created` answers a successful `POST`, usually with a `Location` header. `204 No Content` has an empty body, so a client must not decode JSON from it. `401` really means "unauthenticated" (token missing, invalid, or expired), while `403` means the server knows who you are and still says no. `422` means the request failed validation. `429` and `503` often carry a `Retry-After` header that tells a polite client how long to wait.

> **WARNING**
>
> URLSession does **not** throw for `404`, `500`, or any other HTTP status. A `404` with an error body is a successful network transfer as far as URLSession is concerned. If your code decodes the body without checking the status first, it either crashes into a confusing `DecodingError` or, worse, silently treats an error message as data. Chapter 2 shows the check, and Chapter 7 turns it into a proper error type.

REST, Resources, and the Shape of a Good API

In everyday usage "REST" means an HTTP API organized around **resources** named by URLs, manipulated with the standard methods, and represented as JSON. A well-designed one is predictable: once you have seen three endpoints, you can guess the fourth. The Railbird API follows the common conventions, and each has a client-side consequence:

- **Nouns in paths, verbs in methods.** `/stations` is a collection, `/stations/CEN` one member, `/stations/CEN/departures` a sub-resource. A path is data built from typed values, the idea behind the `Endpoint` type of Chapters 3 and 6.
- **Query parameters filter collections.** `/stations?query=canal&line=L3&limit=20` narrows the list without new paths. Their encoding rules trip up hand-built URLs (Chapter 3).
- **Cursor pagination.** Railbird returns an opaque `next_cursor` that the client sends back for the next page and must never parse (Chapter 9).
- **A version in the path.** A breaking change would ship as `/v2` while old app versions keep using `/v1` (Chapter 28).
- **One error format.** Every error with status 400 or higher uses one JSON envelope with a machine-readable `code`, a `message`, and the `request_id`. Only the OAuth endpoints differ, using the RFC 6749 format (Chapter 17).
- **Caching metadata.** `Cache-Control` and `ETag` say how long a response stays fresh: 15 seconds for the board, an hour for the lines (Chapter 11).

Not every API is this tidy. Some use `POST` for everything or return `200` with an error inside, and GraphQL (Chapter 15) deliberately uses one endpoint for all queries. Build your client around the semantics the server actually implements, and learn them by looking at real traffic.

JSON on the Wire

JSON has six value types: objects, arrays, strings, numbers, booleans, and `null`. Railbird uses **snake_case** keys such as `line_id`, while Swift prefers camelCase. Dates are not a JSON type; they travel as ISO 8601 strings like `"2026-03-16T13:06:00Z"`, sometimes with fractional seconds or as a calendar day like `"2026-03-18"`. And a field can be absent, `null`, or present, which may mean three different things.

Here is one departure exactly as the server sends it:

One departure object

```
{
  "id": "D-1002",
  "line_id": "L2",
  "destination": "Airport Terminal",
  "platform": "3",
  "scheduled_at": "2026-03-16T13:06:00Z",
  "expected_at": "2026-03-16T13:09:00Z",
  "status": "delayed",
  "delay_seconds": 180,
  "accessible": true
}
```

The platform is a string because real stations have platforms called "3a". The status comes from an open-ended set (`on_time`, `delayed`, `boarding`, `departed`, `cancelled`), and an app already on users' phones must not crash when a future server adds a value. A cancelled departure has `"expected_at": null`. Such details are why decoding gets two full chapters.

Meet Railbird and Larkhaven

Railbird is the app we build across the book: a departures app for Larkhaven, a fictional city with two rail lines, a tram line, and a bus route. Every name and timetable is invented. Railbird is small enough to understand completely and rich enough to need every production technique: pagination, a live board, polymorphic alerts, sign-in, uploads, downloads, real-time updates, and offline support.

Line	Name	Mode	Color	Stations in order
L1	Harbor Line	rail	`#D7263D`	HBR, MKT, CEN, RDG
L2	Ridge Line	rail	`#1B98E0`	AIR, CEN, UNI, RDG
L3	Canal Line	tram	`#F4A259`	OLD, CNL, MKT, CEN
B7	Crosstown 7	bus	`#6A4C93`	OLD, MKT, UNI

The eight stations are Airport Terminal (AIR), Canal & 5th (CNL), Central Station (CEN), Harbor Point (HBR), Market Street (MKT), Old Town (OLD), Ridgeview (RDG), and University (UNI). Central Station is the hub where three lines meet, and its departure board is the worked example you will see in almost every chapter. Old Town is the only station without step-free access. Canal & 5th has an ampersand in its name on purpose, because it is the perfect test case for URL encoding in Chapter 3.

A frozen clock

A book is read long after it is written, so the Railbird API lives on a **fixed data clock**: every response is generated for Monday, March 16, 2026, at 13:00:00 UTC, which is 8:00 AM in Larkhaven (time zone `America/Chicago`, CDT, UTC−5 on that date). Each departures response states its generation time in `generated_at`, and Railbird computes "minutes until departure" from that server time rather than the device clock. That is a good habit anyway, because device clocks drift, and it keeps the board looking live whenever you run the code.

This is the board the finished app shows for Central Station:

Departure board as Railbird shows it

```
Central Station                        Updated 8:00 AM
L1  Harbor Point        Platform 1     Boarding
L2  Airport Terminal    Platform 3     8:09 AM  +3 min   9 min
L3  Old Town            Platform 5     8:08 AM           8 min
L1  Ridgeview           Platform 2     Cancelled
L2  Ridgeview           Platform 4     8:14 AM          14 min
L1  Harbor Point        Platform 1     8:19 AM          19 min
```

The minutes column is the whole number of minutes from `generated_at` to the expected time (or the scheduled time if there is no expected time), rounded down. The Ridge Line train to the airport was scheduled for 8:06 AM, is now expected at 8:09 AM, and therefore shows `+3 min` and `9 min`.

The endpoints

The production base URL is `https://api.railbird.example/v1`. The `.example` top-level domain is reserved for documentation, so it can never resolve to a real server; that is deliberate. The mock server serves the same API at `http://127.0.0.1:8080/v1`. Paths below are relative to the base URL.

Request	Auth	Purpose	Chapter
`GET /health`	–	Server status and clock	1
`GET /stations`	–	Search and page stations	3, 9
`GET /stations/{id}`	–	One station	4
`GET /lines`	–	All lines	4
`GET /stations/{id}/departures`	–	Departure board	2, 6
`GET /stations/{id}/departures/events`	–	Live updates (SSE)	22
`GET /alerts`	–	Service alerts	5
`GET /images/stations/{id}.png`	–	Station image	12
`GET /maps/network-map.pdf`	–	Large download	14
`POST /graphql`	–	`StationBoard` query	15

`GET /live`	–	WebSocket updates	21
`GET /me`	Bearer	Signed-in user	16
`PUT`, `DELETE /me/favorites/{id}`	Bearer	Favorites	16, 24
`POST /reports`	Bearer	Create a trip report	13, 24
`PUT /reports/{id}/photo`	Bearer	Attach a photo	13
`POST /uploads`	Bearer	Multipart upload	13
`POST /devices`	Bearer	Register for push	23

The OAuth 2.0 endpoints for signing in live outside the versioned API at `/oauth/authorize`, `/oauth/token`, and `/oauth/revoke` (Chapter 17). Appendix B lists every endpoint with its parameters and sample payloads.

Running the Railbird Mock Server

A fictional API has no production server, so the companion files include `railbird_mock_server.py`, a single-file Python program that implements the entire Railbird API: JSON endpoints, caching headers, pagination, OAuth 2.0 with PKCE, uploads, a resumable download, Server-Sent Events, WebSockets, and fault injection. It uses only Python's standard library. The companion files also include `railbird-openapi.yaml`, a machine-readable description of the API for Chapter 27.

The server needs Python 3.10 or later. Check what your Mac has:

Terminal

```
python3 --version
```

If the command is missing or reports an older version, install a current Python from python.org or with a package manager such as Homebrew. Then, in the folder that contains the companion files, start the server:

Terminal

```
python3 railbird_mock_server.py
Railbird mock API on http://127.0.0.1:8080/v1  (Ctrl+C to stop)
```

Leave that Terminal window open; the server runs until you press Control-C. Each request it handles is logged as one line with the method, the path, and the status code, which is surprisingly useful when you want to know whether your app sent a request at all:

Terminal (server log)

```
[16/Mar/2026 13:00:00] "GET /v1/health HTTP/1.1" 200 -
[16/Mar/2026 13:00:00] "GET /v1/stations/CEN HTTP/1.1" 200 -
```

```
[16/Mar/2026 13:00:00] "GET /v1/stations/XYZ HTTP/1.1" 404 -
```

The log timestamps use your Mac's real clock; only the API data is frozen. Options such as `--port`, `--token-ttl`, and `--quiet` are listed in Appendix B and introduced in the chapters that need them.

> **TIP**
> If port 8080 is already taken by another tool, start the server with `--port 9090` and change the base URL in `APIConfiguration.localMock` to match. Everything else in the book is port-independent.

Fault injection

Real servers fail rarely when you want them to; the mock fails on demand. Add the header `X-Mock-Fault` (or query parameter `mock_fault`) with `429`, `500`, or `503` for that error with a proper body, `slow` for a five-second delay, `drop` to close the connection without a response, or `badjson` for `200 OK` with truncated JSON. Chapters 4, 7, and 8 use these faults to prove the client behaves when things go wrong.

Exploring the API with curl

`curl` ships with macOS: `-s` hides the progress meter and `-i` includes the status line and headers. In a second Terminal window, ask for Central Station:

Terminal

```
curl -si http://127.0.0.1:8080/v1/stations/CEN
```

HTTP response

```
HTTP/1.1 200 OK
Server: RailbirdMock/1.0.0 Python/3.12.0
Date: Mon, 16 Mar 2026 13:00:00 GMT
X-Request-ID: req_000002
Content-Type: application/json; charset=utf-8
Content-Length: 246
ETag: "368c74c6e03faf50"
Cache-Control: public, max-age=3600

{
  "id": "CEN",
  "name": "Central Station",
  "lines": [
    "L1",
    "L2",
    "L3"
  ],
  "location": {
    "latitude": 44.9778,
    "longitude": -93.265
```

```
  },
  "accessible": true,
  "zone": 1,
  "image_url": "/v1/images/stations/CEN.png"
}
```

Read the headers first. `Cache-Control: public, max-age=3600` lets any cache keep this response for an hour. The `ETag` fingerprints this exact representation so a client can later ask "has it changed?" The `image_url` is a path the app must resolve against its server (Chapter 3). Your request IDs will differ, because the counter grows with every request. Now the departure board, shortened:

Terminal

```
curl -si http://127.0.0.1:8080/v1/stations/CEN/departures
```

HTTP response (body shortened)

```
HTTP/1.1 200 OK
Server: RailbirdMock/1.0.0 Python/3.12.0
Date: Mon, 16 Mar 2026 13:00:00 GMT
X-Request-ID: req_000003
Content-Type: application/json; charset=utf-8
Content-Length: 1797
ETag: "fdbc68956c13385e"
Cache-Control: max-age=15

{
  "station_id": "CEN",
  "generated_at": "2026-03-16T13:00:00Z",
  "departures": [
    {
      "id": "D-1001",
      "line_id": "L1",
      "destination": "Harbor Point",
      …
```

The board is fresh for only 15 seconds, the right trade-off for data that changes minute by minute.

Errors look like data

Ask for a station that does not exist:

Terminal

```
curl -si http://127.0.0.1:8080/v1/stations/XYZ
```

HTTP response

```
HTTP/1.1 404 Not Found
Server: RailbirdMock/1.0.0 Python/3.12.0
Date: Mon, 16 Mar 2026 13:00:00 GMT
X-Request-ID: req_000004
```

```
Content-Type: application/json; charset=utf-8
Content-Length: 121

{
  "error": {
    "code": "not_found",
    "message": "No station with id 'XYZ'.",
    "request_id": "req_000004"
  }
}
```

This is a complete, well-formed HTTP response with a JSON body. Nothing about it is a transport failure, which is exactly why URLSession does not throw for it. Notice that the `request_id` in the body matches the `X-Request-ID` header. Invalid parameters produce a `400` with field details; try `?limit=99` on the departures endpoint, which accepts 1 to 30:

HTTP response (headers shortened)

```
HTTP/1.1 400 Bad Request
Content-Type: application/json; charset=utf-8

{
  "error": {
    "code": "invalid_request",
    "message": "'limit' must be an integer from 1 to 30.",
    "request_id": "req_000017",
    "details": [
      {
        "field": "limit",
        "issue": "out_of_range"
      }
    ]
  }
}
```

Protected resources

The account endpoints require a bearer token. Without one, the server answers `401` and explains how to authenticate in the `WWW-Authenticate` header:

HTTP response

```
HTTP/1.1 401 Unauthorized
Server: RailbirdMock/1.0.0 Python/3.12.0
Date: Mon, 16 Mar 2026 13:00:00 GMT
X-Request-ID: req_000007
Content-Type: application/json; charset=utf-8
Content-Length: 121
WWW-Authenticate: Bearer realm="railbird"

{
  "error": {
    "code": "invalid_token",
```