

THE COMPLETE GUIDE

REST API Design

The Complete Guide to Building APIs Developers Love

Resources · Status codes · Errors · Pagination · Idempotency
OpenAPI · Versioning · Security · Webhooks

MODEL · SPECIFY · SECURE · EVOLVE · DELIGHT

THENEXBOOKS

REST API Design

The Complete Guide to Building APIs Developers Love

First Edition

THENEXBOOKS

REST API Design
The Complete Guide to Building APIs Developers Love

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, OpenAPI documents, and configuration examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. Standards evolve, and some specifications mentioned in this book were drafts at the time of writing; always check the current status of a specification before relying on it. The publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

Parcelwise and all people, companies, and data associated with it are fictional; the domain `parcelwise.example` is reserved for documentation. Product names such as OpenAPI, FastAPI, Express, Redis, PostgreSQL, and Pact are trademarks of their respective owners and are used for identification only.

Preface

Every API is a promise. When you publish an endpoint, other people write code against it, deploy that code, and build businesses on top of it. They will not read your source code, attend your planning meetings, or know what you meant. They will only see what the API actually does: the URLs, the methods, the status codes, the bodies, the errors, and the headers. If those are consistent and honest, integrations are quick and quiet. If they are not, every client pays the price — and so do you, in support tickets, workarounds you can never remove, and breaking changes you cannot make.

Most API problems are not caused by missing features. They come from small design decisions made in a hurry: a `200 OK` that carries an error, a list endpoint without pagination, a field that is sometimes a string and sometimes a number, a `POST` that charges a customer twice when the network hiccups, or a version change that silently breaks a partner's integration on a Friday night. Each of these is easy to avoid at design time and very hard to fix afterward.

This book is a practical, opinionated guide to designing HTTP APIs that developers **love** to use. We explain the reasoning behind each recommendation from first principles and the relevant standards, show many **bad → better** examples, and are honest about trade-offs where reasonable teams choose differently.

Who This Book Is For

- **Backend developers** who build and maintain HTTP APIs and want to get the details right.
- **API designers and product owners** responsible for public, partner, or internal APIs.
- **Tech leads and architects** who set API standards, review designs, and plan evolution.
- **Frontend, mobile, and integration developers** who want to understand why good APIs look the way they do.

What You Will Learn

- The HTTP semantics that matter for API design, and what REST actually requires.
- How to model resources, design URLs, and choose methods and status codes precisely.
- How to design bodies, errors with Problem Details, headers, and content negotiation.
- Pagination, filtering, partial updates, bulk operations, idempotency, concurrency control, async jobs, and webhooks.
- OpenAPI and design-first development, versioning without breaking clients, documentation, SDKs, hypermedia, and APIs for AI agents.
- Authentication, authorization, the OWASP API Security Top 10, rate limiting, caching, testing, observability, and governance.
- Two complete case studies you can adapt to your own APIs.

How This Book Is Organized

Part I — Foundations (Chapters 1–5) covers why design matters, HTTP, REST, resource modeling, and naming. **Part II — Requests and Responses** (Chapters 6–10) covers methods, status codes, bodies, errors, and headers. **Part III — Collections and Operations** (Chapters 11–15) covers pagination, partial and bulk updates, idempotency, long-running operations, and webhooks. **Part IV — Contracts, Evolution, and Developer Experience** (Chapters 16–20) covers OpenAPI, versioning, documentation and SDKs, hypermedia, and designing for AI agents. **Part V — Security, Performance, and Operations** (Chapters 21–26) covers authentication, authorization, rate limiting, caching, testing, and observability. **Part VI — Case Studies** (Chapters 27–28) brings everything together. The appendices provide a cheat sheet, templates, a design review checklist, and a glossary.

A Note on Standards

Wherever possible we cite the relevant IETF standards — for example RFC 9110 for HTTP semantics and RFC 9457 for Problem Details — so you can check the source yourself. Some widely used mechanisms, such as the `Idempotency-Key` header, the `RateLimit` header fields, and OAuth 2.1, were still drafts when this book was written; we label them as drafts. Code examples use Python (FastAPI) and TypeScript (Express) with stable, widely used APIs, but the design principles apply to any language or framework.

The Running Example: Parcelwise

Throughout the book we design the API of **Parcelwise**, a fictional shipping and parcel-tracking platform at `https://api.parcelwise.example`. Merchants create shipments, compare rates, buy labels, schedule pickups, and track parcels across many carriers; partners receive webhooks; internal apps and AI agents automate the rest. Parcelwise gives us every hard case a real API meets: buying a label exactly once, generating labels asynchronously, quotes that expire, events from many sources, multi-tenant data isolation, and moving customers from a legacy v1 to a clean v2.

Conventions Used in This Book

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

> **NOTE**
>
> Background information, standards details, or a useful aside.

> **WARNING**
>
> A common mistake that can break clients, leak data, or cause outages.

Code, HTTP messages, and field names appear in `monospace`. HTTP examples are shortened to the lines that matter. Numbers labeled *hypothetical* or *illustrative* are examples, not measurements.

Contents

Preface

PART I — FOUNDATIONS

1 Why API Design Matters: APIs as Products

2 HTTP Fundamentals for API Designers

3 REST Principles and Constraints — and What Actually Matters

4 Resource Modeling: Nouns, Relationships, and Boundaries

5 URL and Naming Design

PART II — REQUESTS AND RESPONSES

6 HTTP Methods: Safe, Idempotent, and Everything in Between

7 Status Codes That Tell the Truth

8 Designing Request and Response Bodies

9 Errors That Help: Problem Details and Validation

10 Headers, Content Negotiation, and Media Types

PART III — COLLECTIONS AND OPERATIONS

11 Pagination, Filtering, Sorting, and Search

12 Partial Updates, Bulk Operations, and Batch Requests

13 Idempotency, Concurrency Control, and Safe Retries

14 Long-Running Operations and Asynchronous APIs

15 Webhooks and Event-Driven Integration

PART IV — CONTRACTS, EVOLUTION, AND DEVELOPER EXPERIENCE

16 OpenAPI and Design-First Development

17 Versioning and Evolution Without Breaking Clients

18 Documentation, SDKs, and Developer Experience

19 Hypermedia, Links, and Discoverability

20 Designing APIs for AI Agents and Automation

PART V — SECURITY, PERFORMANCE, AND OPERATIONS

21 Authentication: API Keys, OAuth 2.0, and OpenID Connect

22 Authorization, Multi-Tenancy, and the OWASP API Security Top 10

23 Rate Limiting, Quotas, and Fair Use

24 Caching and Performance

25 Testing APIs: Contract, Integration, and Consumer-Driven Tests

26 Observability, Governance, and the API Lifecycle

PART VI — CASE STUDIES

27 Case Study: The Parcelwise Public Shipping API

28 Case Study: Evolving a Legacy API to v2

APPENDICES

A REST API Cheat Sheet

B Templates: OpenAPI, Errors, and Webhooks

C API Design Review Checklist

D Glossary

PART I

Foundations

Why API design deserves real attention, the HTTP you actually need, what REST really means, and how to model resources and name them well.

1 · Why API Design Matters
2 · HTTP Fundamentals for API Designers
3 · REST Principles and Constraints
4 · Resource Modeling
5 · URL and Naming Design

CHAPTER 1 · PART I — FOUNDATIONS

Why API Design Matters: APIs as Products

An API is a promise you make to people you will never meet, and it is far easier to make that promise well than to take it back. This chapter explains why design deserves deliberate effort, introduces the principles that guide the rest of the book, compares the major API styles honestly, and sets up Parcelwise, the shipping platform we will design together from the first endpoint to the v2 migration.

An API Is a Contract

Every time a client sends a request to your API, it relies on a long list of assumptions. It assumes the URL still exists. It assumes the field called `tracking_number` is still a string and still means what it meant yesterday. It assumes that a `201 Created` means the thing was actually created, that an error body will have the same shape as last week, and that retrying a failed request will not charge the customer twice. None of these assumptions are written in the client's code as explicit checks. They are baked into its logic, its database schema, its dashboards, and its support scripts. That web of assumptions is the real contract, whether or not you wrote any of it down.

This is the first idea to internalize: **the contract is whatever your clients depend on, not whatever you documented**. If your API happens to return shipments sorted by creation date, some client will eventually rely on that order even if your documentation never promised it. If an error message contains the phrase "address invalid", someone will match on that string. Designing an API well means deciding, deliberately and early, which behaviors you are promising and which you are not — and then making the promised ones obvious, consistent, and stable.

A contract has two parties, and the asymmetry between them is what makes API design different from ordinary software design. Inside your own codebase, when you rename a function, the compiler finds every caller and you fix them in the same commit. Across an API boundary, the callers live in other companies' repositories, deployed on schedules you do not control, maintained by developers who may have left the company that wrote the integration. You cannot refactor them. You can only avoid breaking them, or break them politely and slowly with a migration plan. Chapter 17 is devoted to that second path; this chapter is about why the first path — getting it right early — is so much cheaper.

Bad → better: an accidental contract

Consider a first draft of a Parcelwise endpoint that returns a shipment. It was written quickly, by serializing the database row:

Response (bad)

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "ShipmentID": 88213,
  "cust_id": 4411,
  "status": 3,
  "addr_to": "221B Baker St|London|NW1 6XE|GB",
  "created": "09/27/2026 14:03",
  "label_pdf_blob": null,
  "internal_route_code": "LHR-EDGE-7"
}
```

Every field here leaks an implementation decision. The sequential integer ID reveals volume and invites enumeration. The status is a magic number whose meaning lives only in the server code. The address is a pipe-delimited string that clients must parse, and the timestamp uses an ambiguous regional format with no time zone. `internal_route_code` was never meant for customers, but now that it is in the response, someone will build a report on it, and you can never remove it without breaking them. A designed version makes deliberate choices:

Response (better)

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "id": "shp_7Hq2Xk9pLm",
  "object": "shipment",
  "status": "label_purchased",
  "from_address": { "id": "adr_2Lw8Rt5Qc" },
  "to_address": {
    "line1": "221B Baker Street",
    "city": "London",
    "postal_code": "NW1 6XE",
    "country": "GB"
  },
  "created_at": "2026-09-27T14:03:11Z"
}
```

Opaque, prefixed identifiers; a status expressed as a readable enumeration; a structured address; an RFC 3339 timestamp in UTC; and nothing internal. None of this required more engineering effort than the first version. It required a decision, made before the first client arrived.

An API Is a Product

Calling an API a product is not marketing language. It is a practical statement about who the API is for and how its success should be measured. A product has users, and those users have goals that are not your goals. A merchant integrating Parcelwise does not want to "create a shipment resource"; they

want the parcel on the customer's doorstep with the least possible code and the fewest surprises. The API is the means, not the end.

Treating the API as a product changes several habits at once:

- **It has an owner.** Someone is accountable for its coherence across teams, not just for individual endpoints.
- **It has a lifecycle.** Features are introduced, evolved, deprecated, and retired on a published schedule rather than whenever a backend team refactors.
- **It has a user experience.** Naming, error messages, documentation, and examples are designed with the same care a product team gives to screens and flows.
- **It is measured by outcomes.** Time to first successful call, number of support tickets per integration, and the share of integrations that reach production are more meaningful than raw request counts.

This holds even for purely internal APIs. An internal API consumed by five teams in your organization is still a product for those five teams. The difference is only in how politely you can negotiate a breaking change; the cost of the change is still real.

What Bad Design Actually Costs

It is tempting to justify API design with dramatic numbers. We will not do that, because the honest answer is that costs vary enormously by organization and nobody can hand you a universal figure. What we can do is describe where the costs come from, so you can recognize them in your own systems.

Breaking changes

A breaking change is any change that causes a previously working client to fail or misbehave. Removing a field, renaming it, changing its type, tightening validation, changing a default, altering the meaning of a status code — all can break someone. The cost is paid twice. First by your consumers, who must discover the breakage (often in production), diagnose it, and ship a fix. Then by you, in incident handling, support conversations, lost trust, and often an emergency rollback that leaves your own team with two code paths to maintain. A poorly designed API tends to accumulate pressure for breaking changes because its early mistakes are painful to live with. A well-designed one leaves room to grow additively.

Support load

Every ambiguity in an API becomes a support ticket. "What does status 3 mean?" "Why did I get a 200 with an error message inside?" "Is the weight in grams or ounces?" "Can I safely retry this request?" Each of these questions is a design defect that has been converted into a recurring human cost. Worse, the answers usually live in the heads of a few engineers, so the support load lands on exactly the people you most need building things.

Integration time

For a platform business like Parcelwise, integration time is directly tied to revenue: a merchant who cannot ship through you this week will ship through a competitor. Inconsistent naming, surprising pagination, and cryptic errors each add friction. Individually they cost minutes; together they turn a one-day integration into a two-week one, and some fraction of prospective customers simply give up. You rarely see those customers in your metrics, because they never become customers.

Internal drag

Bad API design also slows down the team that owns it. When the contract is accidental, every refactor risks changing observable behavior, so engineers become afraid to touch the code. Features get bolted on in whatever shape is expedient, which makes the next feature harder still. Good design is partly a gift to your own future self: a clear contract tells you exactly what you are free to change behind it.

> **NOTE**
> You do not need statistics to make the case for design inside your organization. Count the support tickets that ask what a field means, list the endpoints nobody dares to change, and measure how long your last three partners took to go live. Your own numbers are more persuasive than anyone else's.

Who Consumes APIs

Good design starts with knowing the audience. Parcelwise has at least five distinct kinds of consumers, and most real APIs have a similar mix. Their needs overlap, but they are not identical, and a design that serves only one of them well will frustrate the others.

Merchants

E-commerce merchants range from a solo developer running a small shop to a large retailer with a dedicated integrations team. The small end values simplicity above all: a single request that quotes, buys, and returns a printable label. The large end values control and scale: bulk operations, precise error handling, webhooks instead of polling, and strong guarantees about retries. A good design offers a simple path for the first group without closing doors for the second.

Partners

Carriers and marketplace partners integrate more deeply. A marketplace might create shipments on behalf of thousands of sellers, which raises questions of delegated access and multi-tenant isolation. A carrier might push tracking updates into Parcelwise in bulk. Partners tend to build long-lived integrations and are particularly sensitive to breaking changes, because a single change can affect all of their own customers at once.

Internal teams

Parcelwise's own dashboard, mobile app, and billing system use the same API. Internal consumers are often the first to ask for shortcuts — "just add a field with the data we need for this screen" — and a disciplined API owner must balance their velocity against the coherence of the public contract. A common and healthy rule is that internal apps use the public API wherever possible, so that the company feels its own design decisions.

AI agents

Increasingly, API calls are made by software agents built on large language models, acting on instructions like "ship these three orders by the cheapest carrier that arrives before Friday". Agents read your documentation and OpenAPI descriptions, choose operations, construct requests, and interpret responses — including errors — with no human in the loop for each step. They reward exactly the qualities human developers reward, but more strictly: descriptive names, consistent patterns, explicit enumerations, machine-readable errors that explain how to recover, and safe retry semantics. An operation that is ambiguous to a human is often unusable to an agent. Chapter 20 covers this audience in depth; for now, notice that designing for clarity helps every consumer at once.

Tools and intermediaries

Finally, there are consumers that are not applications at all: API gateways, caches, SDK generators, monitoring tools, security scanners, and HTTP client libraries. These intermediaries understand HTTP semantics, not your business logic. When you use HTTP the way it was designed — correct methods, honest status codes, standard headers — these tools work for you. When you tunnel everything through `POST` with a `200 OK`, they cannot help, and some will actively misbehave.

Design Principles for the Whole Book

Specific rules about URLs, methods, and status codes come later. Underneath them sit four principles that we will keep returning to. When a specific rule seems not to fit your situation, these are what you should reason from.

Consistency

Consistency means that once a consumer has learned one part of your API, they can predict the rest. If one collection paginates with `cursor` and `limit`, every collection does. If one resource uses `created_at`, none use `creationDate`. If one error response follows the Problem Details format of RFC 9457, all of them do. Consistency is the single cheapest way to make an API feel well designed, and it compounds: a consistent API with a few mediocre choices is easier to use than an inconsistent one where each endpoint is individually brilliant.

Bad: three conventions in one API

```
GET /v2/shipments?page=2&per_page=50
GET /v2/getLabels?offset=100&count=50
GET /v2/tracking_events?cursor=abc123&max=50
```

Better: one convention everywhere

```
GET /v2/shipments?limit=50&cursor=eyJpZCI6InNocF8xIn0
GET /v2/labels?limit=50&cursor=eyJpZCI6ImxibF8xIn0
GET /v2/tracking-events?limit=50&cursor=eyJpZCI6InRya18xIn0
```

Predictability

Predictability is consistency's twin, applied to behavior rather than naming. A predictable API behaves the same way under the same conditions: `GET` never changes anything, a `DELETE` that succeeds once leaves the resource gone no matter how many times it is repeated, validation errors always arrive as `422` with the same structure, and rate limiting always arrives as `429 Too Many Requests` with a `Retry-After` header. Predictability is what lets a consumer write generic code — one retry helper, one error handler, one pagination iterator — instead of special-casing every endpoint.

Least surprise

The principle of least surprise says that an API should do what a reasonable, informed developer expects. Informed matters: the expectation is shaped by HTTP itself and by widespread conventions, not by your internal architecture. A developer expects that `POST /shipments` creates a shipment and returns `201 Created` with a `Location` header. They do not expect it to also purchase a label and charge their account unless the request clearly asked for that. They expect that a field named `weight` comes with a unit. Surprises are usually symptoms of an API that mirrors the server's internals rather than the consumer's mental model.

Bad: a surprise hidden in a GET

```
GET /v2/shipments/shp_7Hq2Xk9pLm/label
# Side effect: buys a label and charges the account if none exists yet.
```

Better: the costly action is explicit

```
POST /v2/shipments/shp_7Hq2Xk9pLm/label
Idempotency-Key: 5b0e6c7a-7d0e-4c61-9b8f-2f1a0c9d4e11

GET /v2/labels/lbl_9Tc3Vb6Nw
# Safe: only reads a label that already exists.
```

We will unpack why `GET` must be safe in Chapter 6 and how idempotency keys work in Chapter 13. For now, notice how much a reader can infer from the better version without reading any documentation.

Evolvability

Evolvability is the ability to change the API over years without breaking existing clients. It is achieved mostly through small early choices: using objects rather than bare arrays at the top level of a response so you can add metadata later; using string enumerations and documenting that new values may appear; making new request fields optional; never reusing a field name with a new meaning; and telling clients to ignore fields they do not recognize. An evolvable API grows by addition. An API that is not evolvable grows by versions, and every version is a fork you must maintain.

Bad: a bare array leaves no room to grow

```
[
  { "id": "rate_4Kp9", "carrier": "ups", "amount": "12.40" },
  { "id": "rate_8Qm2", "carrier": "dhl", "amount": "15.10" }
]
```

Better: an envelope that can gain fields safely

```
{
  "data": [
    { "id": "rate_4Kp9", "carrier": "ups", "amount": "12.40", "currency": "USD" },
    { "id": "rate_8Qm2", "carrier": "dhl", "amount": "15.10", "currency": "USD" }
  ],
  "expires_at": "2026-09-27T15:03:11Z",
  "next_cursor": null
}
```

The second shape later gained `expires_at` without breaking anyone, because clients were already reading `data`. Rate quotes that expire are one of Parcelwise's recurring hard cases, and this small structural choice made room for them.

> **TIP**
>
> When two design options seem equally good, choose the one that is easier to extend later. You can almost always add; you can rarely take away.

Choosing a Style: REST, RPC, GraphQL, gRPC, and Messaging

This is a book about REST API design, but it would be dishonest to pretend REST is the right answer for every problem. Mature organizations use several styles, each where it fits. Understanding the alternatives also sharpens your understanding of what REST is good at.

REST over HTTP

REST models a system as a set of *resources* identified by URIs and manipulated through a small, uniform set of methods whose meaning is defined by HTTP. Its strengths are ubiquity (every language and platform speaks HTTP), a rich ecosystem of intermediaries that understand its semantics, cacheability, and loose coupling suitable for public APIs consumed by strangers. Its weaknesses show

up when clients need many related resources in one round trip, or when the domain is dominated by complex operations that do not map naturally onto resources. Chapter 3 examines what REST actually requires.

RPC over HTTP

Remote procedure call style exposes operations rather than resources: `POST /createShipment`, `POST /cancelShipment`, often with every call as `POST` and every response as `200`. JSON-RPC is a formal version of this idea. RPC is easy to design because it mirrors function calls in code, and it can be perfectly reasonable for internal, tightly coupled services. Its cost is that HTTP intermediaries learn nothing from the messages: nothing is cacheable, nothing is safely retryable by default, and every operation must be learned individually.

GraphQL

GraphQL exposes a typed schema and lets clients send queries that specify exactly the fields they want, typically to a single endpoint over `POST`. It shines when many different clients — especially user interfaces — need different slices of a rich, interconnected data graph and want to avoid multiple round trips. The trade-offs are real: HTTP caching becomes harder, query cost must be analyzed and limited to prevent abuse, errors often arrive with a `200 OK`, and authorization must be enforced per field rather than per URL.

gRPC

gRPC defines services and messages in Protocol Buffers, generates client and server code, and runs over HTTP/2 with a compact binary encoding. It supports streaming in both directions. It excels for high-throughput, low-latency communication between internal services where both sides are under your control and can share generated code. It is less convenient for public APIs: browsers cannot call it directly without a proxy layer, the payloads are not human-readable, and ad-hoc debugging with everyday tools is harder.

Asynchronous messaging

Event streams, message queues, and webhooks invert the direction of communication: instead of a client asking, the producer tells interested parties that something happened. This is the natural model for tracking updates, label-ready notifications, and anything where the client would otherwise poll. Messaging decouples producers from consumers in time, but it introduces its own design problems — delivery guarantees, ordering, duplicates, schema evolution — which Chapter 15 addresses for webhooks.

Style	Contract	Best fit	Main weakness
REST	URIs, methods, media types	Public, long-lived APIs	Chatty for deep graphs
RPC over HTTP	Named operations	Simple internal calls	Ignores HTTP semantics

GraphQL	Typed schema, queries	Varied UI data needs	Caching, query cost
gRPC	Protobuf services	Service-to-service	Browsers, readability
Messaging	Event schemas	Notifications, streams	Ordering, duplicates

These styles combine well. Parcelwise itself uses REST for its public API, webhooks to notify merchants of tracking and label events, and gRPC between some internal services such as the rate engine and carrier adapters. A partner portal might one day add a GraphQL layer for its dashboard. The mistake is not choosing one style over another; the mistake is choosing a style by habit rather than by the needs of the consumers.

Bad → better: the same operation in two styles

RPC style (hard for intermediaries to understand)

```
POST /api/ShipmentService.GetShipment HTTP/1.1
Host: api.parcelwise.example
Content-Type: application/json

{ "shipmentId": "shp_7Hq2Xk9pLm" }
```

Resource style (safe, cacheable, self-describing)

```
GET /v2/shipments/shp_7Hq2Xk9pLm HTTP/1.1
Host: api.parcelwise.example
Accept: application/json
```

Both requests fetch the same data. Only the second tells every cache, proxy, retry library, and log analyzer that this is a read with no side effects, safe to repeat and eligible for caching. That is the kind of leverage REST gives you when you use it properly.

Design-First vs Code-First

There are two broad ways to arrive at an API contract. In **code-first** development, engineers implement handlers and let a framework generate the API description from the code — for example, FastAPI generating OpenAPI from Python type hints. In **design-first** development, the team writes the contract first, usually as an OpenAPI document, reviews it with consumers and stakeholders, and then implements the server (and often generates mocks, SDKs, and tests) from it.

Code-first

Code-first is fast and keeps the description in sync with the implementation automatically. Its danger is that the contract becomes a side effect of implementation decisions: the database model becomes the response shape, the framework's defaults become your error format, and nobody reviews the design as a design because there was never a moment when it existed separately.

shipments.py (code-first, FastAPI)

```
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()


class ShipmentRow(BaseModel):
    ShipmentID: int
    cust_id: int
    status: int


@app.get("/getShipment")
def get_shipment(id: int) -> ShipmentRow:
    # The public contract is now whatever this table looks like.
    return ShipmentRow(ShipmentID=id, cust_id=4411, status=3)
```

Design-first

Design-first inserts a deliberate step: the contract is proposed, discussed, and agreed before code exists. The result can be reviewed by people who will never read the server code — partner engineers, technical writers, support staff — and mocks let consumers start integrating in parallel with the implementation.

openapi.yaml (design-first excerpt)

```
openapi: 3.1.0
info:
  title: Parcelwise API
  version: 1.0.0
paths:
  /v2/shipments/{shipment_id}:
    get:
      operationId: getShipment
      summary: Retrieve a shipment
      parameters:
        - name: shipment_id
          in: path
          required: true
          schema:
            type: string
            pattern: "^shp_[A-Za-z0-9]+$"
      responses:
        "200":
          description: The shipment.
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Shipment"
        "404":
          description: No shipment with this ID is visible to the caller.
components:
```

```
  schemas:
    Shipment:
      type: object
      required: [id, object, status, created_at]
      properties:
        id:
          type: string
        object:
          type: string
          const: shipment
        status:
          type: string
          enum: [draft, rated, label_purchased, in_transit, delivered, cancelled]
        created_at:
          type: string
          format: date-time
```

In practice, the dichotomy is softer than it sounds. Many teams sketch the contract design-first, then implement in a framework that generates the final OpenAPI from code, and use continuous integration to check that the generated description matches the approved design. What matters is not which file is written first; it is that there is a moment when humans review the contract as a contract. This book recommends design-first for any API with external consumers, and Chapter 16 shows how to run the process without slowing teams down.

> **WARNING**
>
> "We will clean up the API before we publish it" rarely happens. The first external consumer freezes whatever shape exists on that day. If a design review is going to happen, it has to happen before the first key is issued.

Meet Parcelwise

Throughout this book we will design the public REST API of **Parcelwise**, a fictional shipping and parcel-tracking platform. Parcelwise sits between merchants who need to ship goods and the carriers who move them. A merchant sends Parcelwise the details of a shipment; Parcelwise asks many carriers for prices, lets the merchant choose one, buys the postage, produces a printable label, schedules a pickup if needed, and then aggregates tracking updates from the carrier until the parcel is delivered — or returned. It bills the merchant monthly for labels and services.

The API lives at `https://api.parcelwise.example`. We chose this domain deliberately; like all our examples, it uses the reserved `.example` top-level domain so that nothing in this book points at a real service.

The resources

Here is the first-pass inventory of Parcelwise resources. Chapter 4 refines it into a proper resource model with relationships and boundaries.