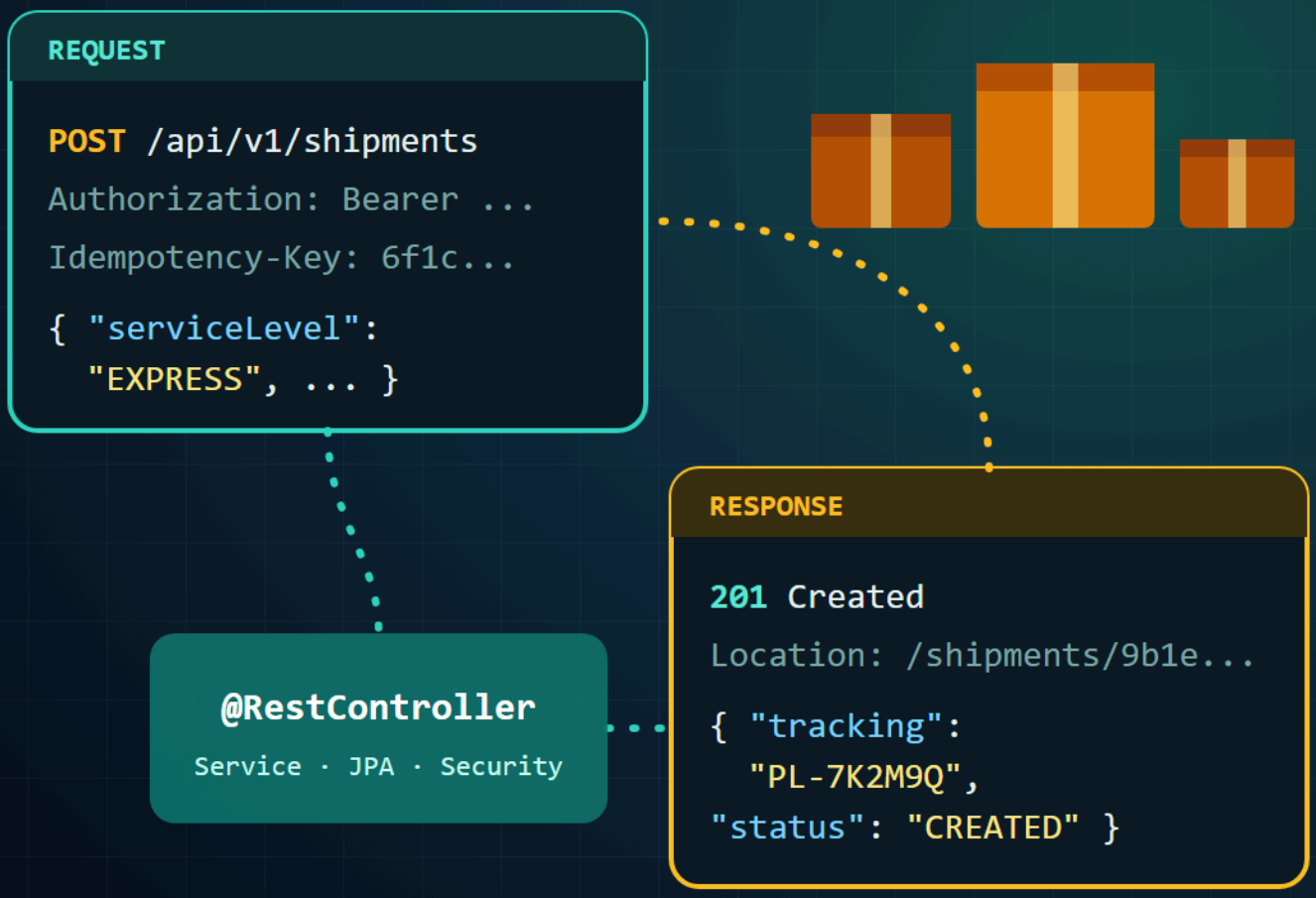


GET · POST · PUT · PATCH · DELETE · 200 · 201 · 409

THE COMPLETE PRACTICAL GUIDE

Spring Boot REST Services

Design, build, secure, and ship production APIs with Java

For Spring Boot 3 on Java 21 · PostgreSQL, Spring Security, Testcontainers, and Kubernetes

REST · JPA · OAUTH2 · RESILIENCE · OBSERVABILITY

THENEXBOOKS

Spring Boot REST Services

The Complete Practical Guide to Designing, Building, Securing, and Shipping Production APIs with Java
For Spring Boot 3 on Java 21

First Edition

THENEXBOOKS

Spring Boot REST Services
The Complete Practical Guide to Designing, Building, Securing, and Shipping Production APIs with Java

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. Spring Boot, its ecosystem, and the third-party libraries discussed in this book evolve quickly; always consult the official reference documentation for the versions you use.

Spring, Spring Boot, and Spring Framework are trademarks of Broadcom Inc. and/or its subsidiaries. Java and OpenJDK are trademarks or registered trademarks of Oracle and/or its affiliates. PostgreSQL, Docker, Kubernetes, and other product names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement. This book is not affiliated with, sponsored by, or endorsed by any of these companies or projects.

PortLoop, the running example in this book, is fictional, as are all merchants, carriers, and people mentioned in its examples. Any resemblance to real companies or products is coincidental.

Preface

Most of the software the world runs on today talks to other software over HTTP. Mobile apps call backends, single-page web apps call backends, partners integrate through public APIs, internal teams split systems into services, and increasingly AI agents call tools that are nothing more than well-described REST endpoints. Behind a large share of those endpoints sits Java — and, very often, Spring Boot.

Spring Boot makes the first endpoint astonishingly easy. A single annotated class returns JSON in minutes. The hard part comes later: designing resources that stay stable as clients multiply, validating input and reporting errors consistently, persisting data without surprise queries, securing every endpoint with the right identity and permissions, evolving the API without breaking anyone, calling other services without cascading failures, and running the whole thing in production where you can see what it is doing. That is where most teams spend their time, and that is where most of this book lives.

This book is a complete, practical guide to building REST services with Spring Boot the way experienced teams do. We start from HTTP and Spring fundamentals, build a real API step by step, and then go deep on persistence, security, API evolution, integration, resilience, testing, performance, observability, and deployment. We explain *why* each technique works — and where it breaks — not just *how* to type it.

Who This Book Is For

- **Java developers** who know the language and want to build professional REST services with Spring Boot.
- **Backend engineers** coming from other stacks who need to become productive in the Spring ecosystem quickly.
- **Spring developers** who can ship endpoints but want solid answers on error handling, security, versioning, testing, and operations.
- **Tech leads and architects** who set conventions for API design, resilience, and production readiness across teams.

What You Will Learn

- HTTP and REST fundamentals, Spring's core container, configuration, and profiles, and how a Spring Boot application is put together.
- Controllers, request and response bodies with Jackson and Java records, Bean Validation, and consistent errors with `ProblemDetail`.
- Resource design: URIs, pagination, filtering, sorting, and hypermedia.

- Spring Data JPA, relationships and fetching, transactions and optimistic locking, Flyway migrations, `JdbcClient`, and caching.
- Spring Security for APIs, JWT and OAuth2 resource servers, and fine-grained authorization.
- Versioning, idempotency, ETags, and OpenAPI documentation.
- Calling other services with `RestClient` and HTTP interfaces, resilience patterns, events, messaging, and webhooks.
- Testing with slices and Testcontainers, virtual threads and performance, observability, container images, and Kubernetes deployment.

How This Book Is Organized

Part I — Foundations (Chapters 1–5) sets up your tools, explains HTTP, REST, and Spring's core ideas, and builds a first working API end to end. **Part II — Building the API** (Chapters 6–10) covers controllers, payloads, validation, error handling, and resource design in depth. **Part III — Data and Persistence** (Chapters 11–15) connects the API to PostgreSQL with Spring Data JPA, transactions, migrations, and lighter-weight data access. **Part IV — Security and API Evolution** (Chapters 16–20) protects the API and lets it change safely over time. **Part V — Integration, Testing, and Performance** (Chapters 21–25) connects PortLoop to other systems, makes it resilient, tests it thoroughly, and makes it fast. **Part VI — Production and Case Study** (Chapters 26–28) makes the service observable, deploys it, and walks through taking the whole platform to production. The appendices provide a cheat sheet, templates and an annotation reference, a production checklist, and a glossary.

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

The Running Example: PortLoop

Throughout the book we build **PortLoop**, a fictional parcel delivery and freight platform. Merchants request quotes and book shipments, couriers scan parcels at depots and doorsteps, customers follow deliveries on a public tracking page, and partner carriers push status updates through webhooks. That gives us everything a real API faces: resources with state machines, high-volume event streams, writes that must be idempotent, several kinds of clients with different permissions, and integrations that fail in interesting ways. The code lives in the package `com.portloop`, stores data in PostgreSQL, and exposes its API under `/api/v1`.

Versions and Conventions

Examples target Spring Boot 3.x on Java 21, with Spring Framework 6 and Spring Security 6 and the Jakarta EE (`jakarta.*`) namespaces. Where a feature arrived in a specific 3.x release, we say so. Spring Boot 4 and Spring Framework 7 continue in the same direction; where it matters, a Note points you to the migration guide rather than guessing at details.

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

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

> **WARNING**
> A common mistake or a behavior that can cause bugs, data loss, or outages in production.

Code appears in `monospace`. Long lines are wrapped to fit the page, DTOs are written as Java records, and dependencies are injected through constructors.

Contents

Preface

PART I — FOUNDATIONS

1 Why Spring Boot for REST Services: The Big Picture

2 Setting Up: JDK, Maven and Gradle, Spring Initializr, and Project Structure

3 HTTP and REST Fundamentals for Spring Developers

4 Spring Core Essentials: Beans, Dependency Injection, Configuration, and Profiles

5 Your First REST Service: The PortLoop Shipments API End to End

PART II — BUILDING THE API

6 Controllers in Depth: Mappings, Path Variables, Query Parameters, and Headers

7 Request and Response Bodies: Jackson, DTOs, and Java Records

8 Validation with Jakarta Bean Validation

9 Error Handling: @RestControllerAdvice, ProblemDetail, and RFC 9457

10 Resource Design: URIs, Pagination, Filtering, Sorting, and Hypermedia

PART III — DATA AND PERSISTENCE

11 Spring Data JPA Fundamentals: Entities, Repositories, and Queries

12 Relationships, Fetching Strategies, and the N+1 Problem

13 Transactions, Concurrency, and Optimistic Locking

14 Schema Migrations with Flyway and Working Well with PostgreSQL

15 Beyond JPA: JdbcClient, Projections, and Caching

PART IV — SECURITY AND API EVOLUTION

16 Spring Security Fundamentals for REST APIs

17 JWT and OAuth2 Resource Servers

18 Authorization: Roles, Scopes, Ownership, and Method Security

19 API Versioning, Idempotency, ETags, and Backward Compatibility

20 Documenting APIs with OpenAPI and springdoc-openapi

PART V — INTEGRATION, TESTING, AND PERFORMANCE

21 Calling Other Services: RestClient, WebClient, and HTTP Interface Clients

22 Resilience: Timeouts, Retries, Circuit Breakers, and Rate Limiting

23 Asynchronous Work, Application Events, Messaging, and Webhooks

24 Testing REST Services: Unit, Slice, and Integration Tests with Testcontainers

25 Performance and Scalability: Virtual Threads, Connection Pools, and Profiling

PART VI — PRODUCTION AND CASE STUDY

26 Observability: Actuator, Micrometer, Structured Logging, and Tracing

27 Packaging and Deployment: Docker Images, Kubernetes, and Externalized Configuration

28 Case Study: Taking the PortLoop Platform to Production

APPENDICES

A Spring Boot REST Cheat Sheet

B Templates and Annotation Reference

C Production Checklist

D Glossary

PART I

Foundations

Why Spring Boot is a strong home for REST services, how to set up a modern Java project, what HTTP and REST really demand of an API, how Spring's container wires an application together, and a first PortLoop API built end to end.

1 · Why Spring Boot for REST Services

2 · Setting Up Your Project

3 · HTTP and REST Fundamentals

4 · Spring Core Essentials

5 · Your First REST Service

CHAPTER 1 · PART I — FOUNDATIONS

Why Spring Boot for REST Services: The Big Picture

Before writing a single controller, it pays to understand what Spring Boot actually does for you, what it deliberately leaves to you, and why so many teams choose it for HTTP APIs. This chapter builds that mental model and introduces PortLoop, the parcel delivery platform we will build throughout the book.

What We Mean by a "Production REST Service"

Almost any web framework can return JSON from a URL in a dozen lines of code. That is not the hard part of building APIs, and it is not what this book is about. The hard part is everything that surrounds that first response: accepting input from clients you do not control, validating it, storing it safely under concurrent load, reporting errors in a way that client developers can act on, authenticating and authorizing callers, evolving the contract without breaking existing integrations, observing the service when it misbehaves at three in the morning, and deploying new versions without downtime.

A production REST service, in the sense we use throughout this book, is an HTTP API that other software depends on. That dependency changes everything. A mobile app installed on thousands of phones cannot be redeployed on your schedule. A partner carrier that integrated against your webhook format last year will not rewrite its code because you renamed a field. A merchant whose warehouse system books shipments through your API expects the same request to behave the same way tomorrow. Every decision you make about URLs, status codes, JSON shapes, and error formats becomes a promise.

So when we evaluate a framework for this kind of work, the question is not "can it serve JSON?" but "how much of the surrounding machinery does it give me, how well is that machinery designed, and how easily can I replace the parts that do not fit?" Spring Boot answers those questions better than most, and the rest of this chapter explains why — and where its trade-offs lie.

A Short History: From Spring Framework to Spring Boot

To understand Spring Boot, you first need to understand the Spring Framework underneath it. Spring started in the early 2000s as a reaction against heavyweight enterprise Java of that era, where application servers dictated how you structured code and simple business objects were buried under container-specific interfaces. Spring's central idea was **inversion of control**: your classes are plain Java objects, and a container creates them, wires their dependencies together, and manages their lifecycle. You describe *what* your objects need, and the container figures out *how* to supply it.

Over time the Spring Framework grew a large family of modules: Spring MVC for web applications, transaction management, data access helpers, messaging, security (as a separate project), and more. Each module was well designed, but assembling them into a working application required a lot of configuration. Developers spent hours choosing compatible library versions, declaring infrastructure beans, configuring a servlet container, and packaging the result into a WAR file for deployment to an external application server.

Spring Boot, first released in 2014, attacked that assembly problem directly. It did not replace the Spring Framework; it sits on top of it. Boot contributes three big ideas:

- **Starters** — curated dependency bundles such as `spring-boot-starter-web` that pull in a coherent, tested set of libraries with compatible versions.
- **Auto-configuration** — conditional configuration classes that inspect your classpath and your properties, then create sensible infrastructure beans (a JSON mapper, an embedded web server, a data source, a transaction manager) unless you define your own.
- **Executable applications** — a self-contained JAR with an embedded server that you start with `java -jar`, instead of deploying into a separately managed application server.

Together these ideas collapse the distance between "here is an idea for an endpoint" and "it is running on my laptop" from hours to minutes, without hiding the underlying Spring Framework from you when you need it.

The Spring Boot 3 generation

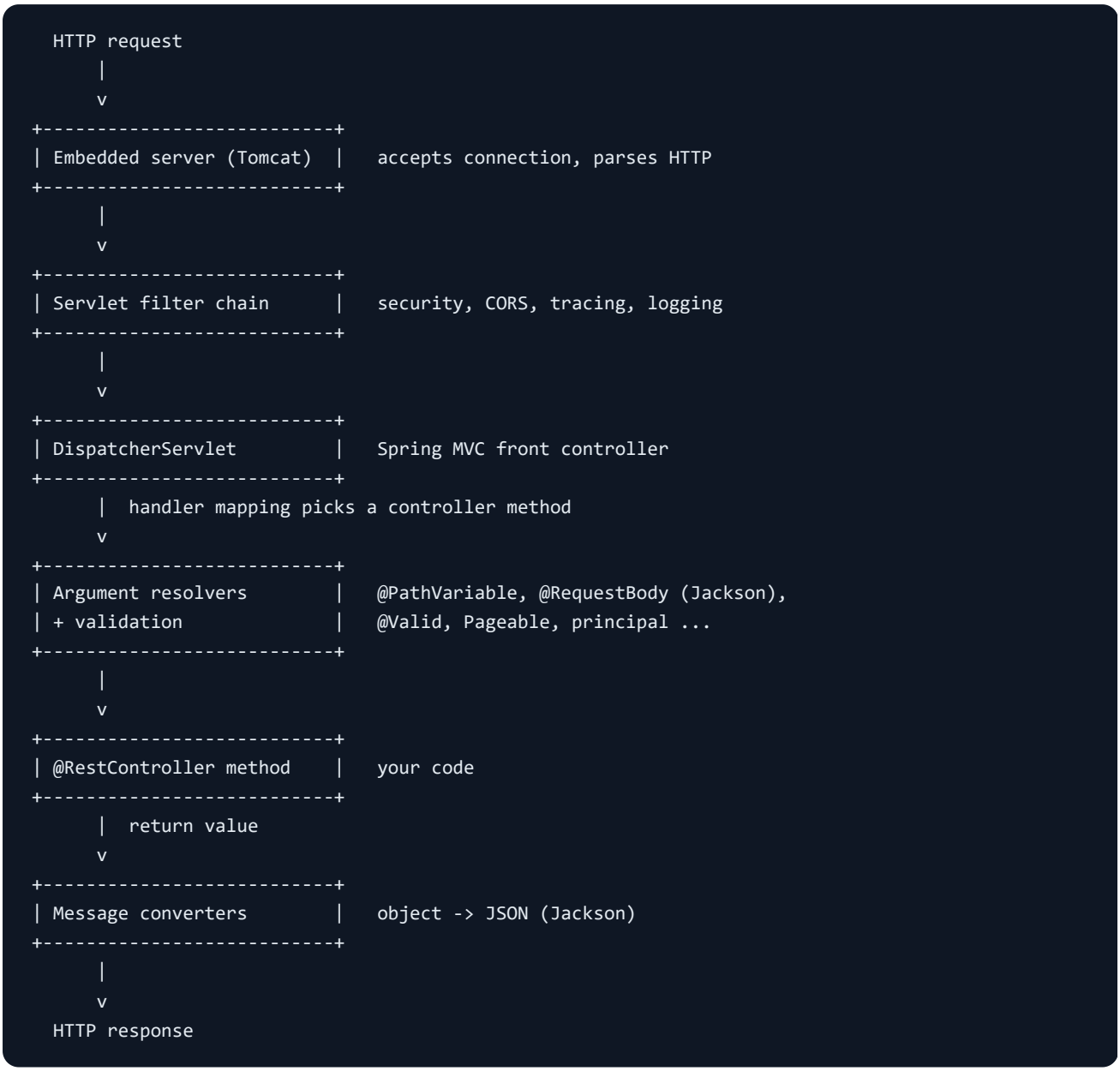
This book targets Spring Boot 3.x running on Java 21. Spring Boot 3.0 was a significant release: it requires Java 17 or later, builds on Spring Framework 6, and moved from the old `javax.*` namespaces to Jakarta EE's `jakarta.*` namespaces. If you read older tutorials, you will see imports such as `javax.persistence.Entity` or `javax.validation.constraints.NotNull`; in Spring Boot 3 these are `jakarta.persistence.Entity` and `jakarta.validation.constraints.NotNull`. Mixing the two is one of the most common sources of confusing errors when developers copy code from older articles.

Subsequent 3.x releases added features we will use heavily: Testcontainers integration and Docker Compose support in 3.1; the `RestClient` and `JdbcClient` APIs and first-class virtual thread support in 3.2; and built-in structured logging and the `@MockitoBean` test annotation in 3.4. Each of these gets its own treatment later in the book.

> **NOTE**
>
> Spring Boot 4 and Spring Framework 7 exist as the newer major line and continue the same direction (for example, first-class API versioning support). The concepts in this book carry over, but property names and some APIs can change between major versions, so check the official migration guide for your version before upgrading.

What Spring Boot Gives You for REST APIs

Let us make the value concrete. When you add `spring-boot-starter-web` to a project, auto-configuration sets up a stack of components that you would otherwise assemble by hand. The following diagram shows the path a typical request takes through that stack.



Each box in that diagram is a place where you can customize behavior, and each one is preconfigured with reasonable defaults. In practice, this means Spring Boot gives you, out of the box:

- **An embedded web server.** Tomcat by default, with Jetty and Undertow available as alternatives. No separate server installation, no WAR deployment.
- **Spring MVC's annotation-based programming model.** You write methods annotated with `@GetMapping`, `@PostMapping`, and so on; the framework handles routing, parameter binding, and content negotiation.

- **JSON serialization with Jackson.** An `ObjectMapper` is configured for you, including support for Java 8 date and time types and records, and you can tune it with properties or customizer beans.
- **Bean Validation integration.** Add `spring-boot-starter-validation` and annotate request bodies with `@Valid`; constraint violations become 400 responses automatically.
- **Structured error responses.** Spring Framework 6 supports RFC 9457 problem details through `ProblemDetail`, so errors can have a standard JSON shape.
- **Externalized configuration.** Settings come from `application.yml`, environment variables, command-line arguments, and more, with a well-defined precedence order.
- **Production features.** Spring Boot Actuator exposes health checks, metrics, and operational endpoints; Micrometer integrates with common monitoring systems.

None of this is magic. Every auto-configured component is an ordinary Spring bean created by an ordinary configuration class that happens to be annotated with conditions. When you understand that — and Chapter 4 makes sure you do — Spring Boot stops feeling like a black box and starts feeling like a very well-organized default configuration you are free to override.

The smallest useful Spring Boot REST service

Here is a complete, runnable Spring Boot application exposing one endpoint. Do not worry about every detail yet; Chapter 2 walks through project setup and Chapter 5 builds a full API. The point is how little ceremony is required.

PortLoopApplication.java

```
package com.portloop;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class PortLoopApplication {

    public static void main(String[] args) {
        SpringApplication.run(PortLoopApplication.class, args);
    }
}
```

PingController.java

```
package com.portloop.ping;

import java.time.Instant;
import java.util.Map;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
```

```
@RestController
@RequestMapping("/api/v1/ping")
public class PingController {

    @GetMapping
    public Map<String, Object> ping() {
        return Map.of(
                "service", "portloop",
                "status", "ok",
                "time", Instant.now());
    }
}
```

Run the application and call the endpoint:

```
$ curl -i http://localhost:8080/api/v1/ping
HTTP/1.1 200
Content-Type: application/json

{"service":"portloop","status":"ok","time":"2026-03-14T09:21:07.512Z"}
```

A handful of things happened without any configuration from us. Tomcat started on port 8080. The `DispatcherServlet` was registered. Component scanning found `PingController` because it lives in a sub-package of `com.portloop`. Jackson serialized the map, including the `Instant`, as ISO-8601 text. The response carried a proper `Content-Type` header. Each of those behaviors is configurable, and each is a decision someone would otherwise have to make and code by hand.

> **TIP**
>
> Keep your main application class in the root package of your project (here `com.portloop`). `@SpringBootApplication` scans its own package and all sub-packages for components. If you put controllers in a sibling package such as `com.portloopapi`, they silently will not be found, and your endpoints return 404 with no obvious error.

Why Teams Choose Spring Boot

Technical capability is necessary but not sufficient. Frameworks are chosen for years, and many of the reasons teams settle on Spring Boot are about the long term rather than the first week.

A coherent, integrated ecosystem

A REST service rarely stands alone. It needs a database, probably a cache, often a message broker, security integration with an identity provider, and clients for other HTTP services. In the Spring ecosystem these pieces are designed to work together: Spring Data for persistence, Spring Security for authentication and authorization, Spring's transaction abstraction, Spring AMQP and Spring for Apache Kafka for messaging, and Spring Boot starters for each. Because they share the same

container, configuration model, and conventions, knowledge transfers from one to the next. Once you understand how Boot auto-configures a data source, you understand the pattern it uses for a Kafka producer or a Redis connection.

Dependency management you do not have to think about

Spring Boot publishes a bill of materials — a curated list of dependency versions that have been tested together. When you use the Spring Boot parent POM or Gradle plugin, you declare dependencies without version numbers and get compatible versions of Jackson, Hibernate, Tomcat, Micrometer, and many other libraries. Upgrading Spring Boot upgrades that whole set in a coordinated way. This sounds mundane, but anyone who has debugged a `NoSuchMethodError` caused by two libraries pulling in incompatible versions of a shared dependency knows how valuable it is.

Mature, conservative evolution

Spring projects follow a predictable release cadence, publish migration guides for major versions, and deprecate APIs before removing them. You will see this throughout the book: `@MockBean` was deprecated in favor of `@MockitoBean` rather than removed overnight; `WebSecurityConfigurerAdapter` was deprecated in Spring Security 5.7 before its removal in 6.0. For a service you intend to run for years, predictable evolution matters more than novelty.

Production features built in

Health checks, metrics, graceful shutdown, externalized configuration, and structured logging are not afterthoughts in Spring Boot; they are first-class features. Kubernetes liveness and readiness probes can be served directly by Actuator. Micrometer metrics can be exported to Prometheus or other backends by adding a dependency. These are exactly the capabilities that separate a demo from a service an operations team is willing to run.

What Spring Boot Does Not Do for You

It is equally important to be clear about the limits. Spring Boot is a toolkit and a set of defaults, not an architecture. It will not:

- **Design your API.** It will happily let you expose your JPA entities directly as JSON, use `POST` for everything, and return 200 with an error message in the body. Good resource design (Chapter 10) and careful DTO boundaries (Chapter 7) are your responsibility.
- **Make your database access efficient.** Spring Data JPA makes it easy to write a repository method; it also makes it easy to trigger dozens of queries for a single request. Chapter 12 is devoted to the N+1 problem for a reason.
- **Secure your endpoints by intent.** Adding Spring Security locks everything down by default, which is a safe starting point, but deciding who may do what — a merchant can see only its own shipments, a courier can scan only parcels assigned to its depot — is domain logic you must write (Chapters 16–18).

- **Guarantee correctness under concurrency.** Transactions, optimistic locking, and idempotency keys are tools; using them correctly requires understanding your domain's invariants (Chapters 13 and 19).
- **Replace measurement.** Defaults are chosen to be reasonable for many workloads, not optimal for yours. Pool sizes, timeouts, and thread models need to be tuned against real traffic.

> **WARNING**
>
> The ease of getting started is also Spring Boot's biggest trap. Because a working endpoint takes minutes, teams sometimes ship the first thing that works: entities serialized straight to JSON, no validation, no error contract, default timeouts on outbound calls. Those shortcuts become part of your public contract the moment a client depends on them, and they are expensive to undo. Treat the first version of every endpoint as a design decision, not a prototype.

Spring Boot Among the Alternatives

Spring Boot is not the only way to build REST services in Java, and it is worth knowing where it sits relative to other options so you can explain your choice to your team. We will avoid benchmark claims here — any performance comparison depends heavily on workload, configuration, and version, and you should measure your own — and focus on design differences.

Option	Style	Typical strengths	Considerations
Spring Boot	Full ecosystem, runtime reflection plus optional AOT	Breadth of integrations, maturity, community	Larger dependency footprint; many concepts to learn
Quarkus	Build-time optimized, Jakarta and MicroProfile APIs	Fast startup focus, native images	Different programming model and extension system
Micronaut	Compile-time dependency injection	Low reflection use, native images	Smaller ecosystem than Spring
Jakarta EE server	Standard APIs on an app server	Vendor-neutral specifications	Deployment model varies by vendor
Lightweight libraries	Minimal HTTP routing (e.g. Javalin)	Small, explicit, easy to read	You assemble everything else yourself

All of these can produce excellent services; the choice usually comes down to team knowledge, required integrations, and what your deployment environment rewards. Where startup time dominates, look at Spring Boot's ahead-of-time processing and GraalVM native image support as well as the alternatives.

Servlet stack or reactive stack?

Spring offers two web stacks. **Spring MVC** runs on the Servlet API and, traditionally, uses one thread per request: while your code waits for the database, that thread waits too. **Spring WebFlux** is a reactive, non-blocking stack built on Project Reactor, where a small number of event-loop threads serve many concurrent requests as long as no code blocks them.

This book uses Spring MVC as its primary stack, for three reasons. First, the imperative programming model is easier to read, debug, and test, and most Java libraries — including JDBC drivers and JPA — are blocking. Second, Java 21 virtual threads change the calculus: with `spring.threads.virtual.enabled=true`, Spring Boot 3.2 and later can run request handling on virtual threads, which are cheap enough that blocking a thread no longer means tying up a scarce OS resource. Many services that would once have needed a reactive design to handle high concurrency can now stay imperative. Third, the majority of production Spring APIs use Spring MVC, so it is what you are most likely to maintain.

WebFlux remains the right choice in some situations — streaming responses, very high fan-out to other services, or teams already fluent in reactive programming. We will use `WebClient`, the reactive HTTP client, in Chapter 21 where it fits naturally, and discuss virtual threads in depth in Chapter 25. But you do not need to learn reactive programming to build an excellent REST service.

> **NOTE**
> Virtual threads do not make blocking free. They remove the cost of parking a platform thread, but a request still waits for the database, and a connection pool with ten connections still serves at most ten concurrent queries. Chapter 25 explains how thread pools, connection pools, and virtual threads interact, and why you must measure with your own workload.

Meet PortLoop: Our Running Example

Abstract examples make for forgettable books. Throughout this book we build the API for **PortLoop**, a fictional parcel delivery and freight platform. We chose a logistics domain deliberately: it exercises almost every hard problem in REST design in a way that feels natural rather than contrived.

The business

PortLoop connects several kinds of users. **Merchants** — online shops and wholesalers — book shipments through a web dashboard or directly via the API from their warehouse systems. **Couriers** collect parcels and deliver them, scanning each parcel with a mobile app at every step. **Customers** — the people receiving parcels — check delivery progress on a public tracking page. **Partner carriers**, such as the fictional `NORTHWIND-XPRESS` and `BLUEFIN`, handle long-haul legs and push status updates to PortLoop via webhooks. Internally, **support** staff investigate problems and **administrators** manage the platform.

The core domain

Here are the main concepts we will model. You will see these names again and again, so it is worth reading them carefully now.

Concept	Purpose	Key fields
Merchant	A business that ships parcels	id, name, contactEmail, plan
Shipment	One booked delivery	trackingNumber, status, origin, destination
Parcel	A physical box in a shipment	barcode, dimensions, weightGrams
TrackingEvent	Something that happened to a shipment	type, location, occurredAt, source
Quote	A price offer before booking	serviceLevel, priceAmount, expiresAt
Depot	A physical sorting location	code, city, countryCode
Courier	A driver attached to a depot	name, depotId, vehicleType

A **shipment** is the heart of the system. It belongs to a merchant, has an origin and destination address, a service level (`STANDARD`, `EXPRESS`, or `SAME_DAY`), and one or more parcels. Most importantly, it has a **status** that moves through a state machine:

```
CREATED -> LABEL_PRINTED -> PICKED_UP -> IN_TRANSIT
        -> OUT_FOR_DELIVERY -> DELIVERED

Side exits:
  any active state  -> EXCEPTION   (damaged, address problem, ...)
  CREATED / LABEL_PRINTED -> CANCELLED
  EXCEPTION / OUT_FOR_DELIVERY -> RETURNED
```

That state machine immediately raises interesting API questions. Should a client change status with `PATCH` on the shipment, or should status changes be modeled as events posted to a sub-resource? What should happen when a courier's scanner, which lost connectivity in a basement, uploads a `PICKED_UP` scan an hour after the shipment was already marked `IN_TRANSIT` by a carrier webhook? What status code do we return when a merchant tries to cancel a shipment that is already out for delivery? We will answer each of these in the relevant chapter.

The API surface

PortLoop's API lives under the base path `/api`, versioned from day one. The main resources look like this:

```
GET    /api/v1/merchants/me
POST   /api/v1/quotes
POST   /api/v1/shipments
GET    /api/v1/shipments?status=IN_TRANSIT&page=0&size=20
GET    /api/v1/shipments/{id}
```

```
PATCH  /api/v1/shipments/{id}
GET    /api/v1/shipments/{id}/parcels
GET    /api/v1/shipments/{id}/events
POST   /api/v1/shipments/{id}/events
GET    /api/v1/tracking/{trackingNumber}          (public)
POST   /api/v1/webhooks/carriers/{carrierCode}    (partner carriers)
```

A booking request from a merchant's warehouse system will look roughly like this by the time we finish Part II:

```
POST /api/v1/shipments HTTP/1.1
Host: api.portloop.example
Authorization: Bearer eyJhbGciOi...
Content-Type: application/json
Idempotency-Key: 5d1c6b0e-2f7a-4c1e-9b44-8a1f0c3e7d21

{
  "serviceLevel": "EXPRESS",
  "origin": {
    "line1": "12 Harbour Road",
    "city": "Leeds",
    "postalCode": "LS1 4AB",
    "countryCode": "GB"
  },
  "destination": {
    "line1": "88 Station Street",
    "city": "Manchester",
    "postalCode": "M1 2CD",
    "countryCode": "GB"
  },
  "parcels": [
    { "lengthCm": 40, "widthCm": 30, "heightCm": 20, "weightGrams": 2400 }
  ],
  "declaredValue": 149.90
}
```

And the response:

```
HTTP/1.1 201 Created
Location: /api/v1/shipments/3f6c2a9e-8b1d-4d5e-a0c7-1e2f3a4b5c6d
Content-Type: application/json

{
  "id": "3f6c2a9e-8b1d-4d5e-a0c7-1e2f3a4b5c6d",
  "trackingNumber": "PL-7K2M9Q",
  "status": "CREATED",
  "serviceLevel": "EXPRESS",
  "weightGrams": 2400,
  "createdAt": "2026-03-14T09:30:12Z"
}
```

Notice the details that are already present: a `201 Created` status with a `Location` header pointing to the new resource, an `Idempotency-Key` so that a retry after a network timeout does not book the shipment twice, a bearer token identifying the merchant, and a response body that is a deliberately designed DTO rather than a dump of the database row. Each of these is a small decision with large consequences, and each will be explained properly in its own chapter.

Why logistics is a good teacher

Consider the kinds of pressure this domain puts on an API:

- **State machines** force us to think about which transitions are legal, how to report illegal ones, and how to handle concurrent updates — a perfect vehicle for optimistic locking and the `409 Conflict` status code.
- **High-volume event streams** from scanners and carriers stress write throughput, pagination of long histories, and the difference between synchronous and asynchronous processing.
- **Idempotent writes** matter because a duplicate booking costs a merchant real money, and mobile networks drop connections constantly.
- **Partner webhooks** teach us to accept inbound calls from systems we do not control, verify signatures, and cope with duplicates and out-of-order delivery.
- **Multi-tenant security** is unavoidable: a merchant must never see another merchant's shipments, a courier sees only its assignments, and the public tracking page must reveal just enough and nothing more.

These are not exotic requirements. Swap "shipment" for "order", "payment", "ticket", or "claim", and you will recognize the same pressures in almost any business API you build.

The Architecture We Will Build

PortLoop starts life as a single Spring Boot application — a well-structured monolith — backed by PostgreSQL. That is a deliberate choice. Splitting a system into microservices before you understand its boundaries tends to produce distributed complexity without the benefits. A single deployable unit with clear internal modules gives you fast development, simple transactions, and straightforward testing, while leaving the door open to extract services later when there is a concrete reason to.

