packt

Mastering RESTful Web Services with Java

Practical guide for building secure and scalable production-ready REST APIs

Marián Varga | Pedro Henrique Pereira de Andrade
Silvio de Morais | Thiago Bomfim | Igor Avancini Fraga

Mastering RESTful Web Services with Java

Practical guide for building secure and scalable production-ready REST APIs

Marián Varga
Pedro Henrique Pereira de Andrade
Silvio de Morais
Thiago Bomfim
Igor Avancini Fraga

Mastering RESTful Web Services with Java

Portfolio Director: Ashwin Nair
Relationship Lead: Sneha Shinde
Project Manager: Ruvika Rao
Content Engineer: Roshan Ravi Kumar
Technical Editor: Sweety Pagaria
Copy Editor: Safis Editing
Proofreader: Roshan Ravi Kumar
Indexer: Pratik Shirodkar
Production Designer: Jyoti Kadam
Growth Lead: Anamika Singh

First published: July 2025

Production reference: 1090725

Published by Packt Publishing Ltd.
Grosvenor House
11 St Paul's Square
Birmingham
B3 1RB, UK.

ISBN 978-1-83546-610-0

www.packtpub.com

Contributors

About the authors

Marián Varga has dedicated his career to the integration and API aspects of software solutions across diverse industries. His extensive experience has given him a front-row seat to the evolution of various API styles, their implementation possibilities, and the challenges they present. Passionate about connecting systems through APIs and integration solutions, Marián also bridges the gap between business and technical people, enhancing the efficiency and enjoyment of software development. Marián creates content for and builds a community of people interested in integration at love2integrate.com.

Pedro Henrique Pereira de Andrade has over 14 years of hands-on experience as a software engineer, specializing in ERP systems for small to medium-sized companies across diverse industries. Over the past six years, he has transitioned to larger projects in various countries for different domains, showcasing his adaptability and expertise in using Java technologies such as the Spring Framework for web applications.

As a co-founder of BarreirasJUG, a Java User Group in Brazil, Andrade has actively contributed to the Java community, fostering collaboration and knowledge sharing.

Silvio de Morais is a software architect from Porto Alegre, Brazil, with over 30 years of experience in IT and software development. He has led projects for Fortune 500 companies, focusing on digital transformation, cloud computing, and software architecture. Silvio holds degrees from PUC Brazil and Harvard University and has conducted research at the University of Tokyo. He actively contributes to tech forums, conferences, and Java User Groups, promoting continuous improvement. He now lives in Orlando, FL, with his wife.

Thiago Bomfim has many years of experience in developing Java web applications. He has worked in start-ups and large companies, where he has had the opportunity to create REST APIs for all types of clients, including desktop, web, backend, and mobile. His career is fueled by a deep commitment to engineering excellence, following the best practices of RESTful APIs and focusing on high performance and backward compatibility. Beyond coding, Thiago is passionate about sharing knowledge. He writes technical articles on his blog, speaks at events, mentors developers, and co-hosts the Out of The Box Developer podcast.

Igor Avancini Fraga possesses over 12 years of experience in software engineering and is a full member of the **Java Community Process (JCP)** by Oracle. He has diligently worked on developing applications, services, and features using Java, C#, and X++ for various companies in Brazil and the United States, thereby delivering numerous successful projects throughout his career, encompassing both backend and frontend domains. In recent years, his focus has predominantly been on backend development tailored for the cloud, with a particular emphasis on the development of RESTful APIs and their seamless integration with other services, third-party platforms, and, more recently, AI technologies.

About the reviewers

Mohamed Abdou, a skilled software engineer at Amazon UK, is a contributor to artificial general intelligence. His career reflects technical acumen and passion for impactful software solutions. Mohamed's extensive experience covers large-scale systems, software engineering, AI/ML, and cybersecurity. He has a strong academic background and significant hands-on experience in building complex software and APIs, including open source contributions used by millions and a technical book benefiting many developers. He offers valuable practical knowledge and a unique perspective on modern development. His expertise in secure coding, agile methods, technical leadership, mentoring, and problem-solving highlights his capabilities as a seasoned professional.

Felipe Alexandre Oliveira is a seasoned backend developer with over 11 years of experience in software engineering. He has contributed to the development of high-traffic systems and applications with complex business logic, consistently delivering scalable and maintainable solutions. His technical expertise spans **object-oriented programming** (**OOP**), data structures, Java, Spring, **test-driven development** (**TDD**), AWS, and Docker.

Felipe has a strong interest in application scalability, cloud architecture, and software development best practices. He is passionate about technology and thrives on the challenge of turning real-world problems into effective, user-focused solutions.

Table of Contents

Preface

This book is a practical guide to designing, building, documenting, testing, and deploying RESTful APIs using modern Java frameworks. REST APIs have become a standard way of enabling communication between systems, especially in microservices and other distributed architectures. With Java being one of the most widely used languages for backend development, understanding how to build effective APIs with Java is a key skill for today's developers.

Throughout the chapters, you'll explore hands-on techniques using Spring Boot, OpenAPI, and other popular tools and libraries. You'll also gain exposure to containerization, observability, API security, and performance tuning – everything you need to create robust, scalable, and maintainable APIs. Whether you're working in a start-up or an enterprise, the skills covered here will help you deliver high-quality backend services that meet modern development standards.

Who this book is for

This book is ideal for backend Java developers with a few years of experience who are looking to improve the design, performance, and scalability of their REST APIs. It's also a valuable resource for tech leads and software architects involved in designing distributed systems.

What this book covers

Chapter 1, *Understanding RESTful Core Concepts*, introduces the foundational principles of REST architecture and explains the different maturity levels of RESTful APIs. It also covers the role of JSON as a standard data format, outlines practical guidelines for designing effective APIs, and explores common use cases where REST is applied in real-world systems.

Chapter 2, *Exposing a RESTful API with Spring*, focuses on designing and implementing a RESTful API using Spring Boot. You'll begin by designing a simple Product API, then move on to building it step by step with Spring Boot, one of the most popular frameworks for Java-based web applications. This chapter provides hands-on experience in turning REST concepts into a working API.

Chapter 3, *Documenting Your API Effectively*, explores the importance of clear and accurate API documentation in building reliable and maintainable services. You'll learn how to use OpenAPI to standardize your API documentation and understand the differences between specification-first and code-first approaches. The chapter also walks you through documenting the Product API introduced earlier, ensuring it is both understandable and usable by other developers and systems.

Chapter 4, *Generating Code from OpenAPI*, guides you through creating a full API based on an OpenAPI specification. You'll start by defining the Order Management API specification, then generate the necessary code to jumpstart development. The chapter also covers organizing the project's package structure, implementing the API controller, and integrating the Order Management API and the Product API introduced earlier.

Chapter 5, *Managing API Evolution*, shows how to handle changes and updates to your APIs without disrupting existing users. You'll learn about different versioning strategies, see how to implement versioning in the Product API, and explore best practices for managing API evolution to ensure your services remain reliable and backwards compatible.

Chapter 6, *Advanced API Concepts and Implementations*, dives into key techniques for building robust and user-friendly APIs. You'll explore advanced data handling methods such as pagination, filtering, and file upload/download through REST APIs. The chapter also covers HATEOAS to improve API navigation and introduces resilience patterns to make your APIs more reliable and fault-tolerant in real-world scenarios.

Chapter 7, *Securing Your RESTful APIs*, focuses on protecting your APIs from unauthorized access and security threats. You'll learn about authentication and authorization techniques, explore key security principles from OWASP, and understand how to address common vulnerabilities through the management of **Common Vulnerabilities and Exposures (CVEs)** to keep your APIs safe and trustworthy.

Chapter 8, *Testing Strategies for Robust APIs*, explores both traditional API testing methods and the exciting impact of generative AI on the testing process. You'll learn how AI tools such as ChatGPT are transforming test creation, making it faster, smarter, and more effective, while still covering essential testing types, tools, and best practices for building reliable APIs.

Chapter 9, *Monitoring and Observability*, dives into best practices for logging and tracing in RESTful APIs. You'll learn how structured logging, correlation IDs, and centralized filters help with effective troubleshooting, while distributed tracing tools such as Micrometer and OpenTelemetry provide deep visibility into service performance and request flows across distributed systems.

Chapter 10, *Scaling and Performance Optimization Techniques*, covers essential strategies to improve the speed and scalability of RESTful APIs. You'll learn about performance-focused design, leveraging Java's virtual threads, using infrastructure effectively, and validating improvements through load testing to ensure your APIs handle growing demands efficiently.

Chapter 11, *Alternative Java Frameworks to Build RESTful APIs*, provides a high-level overview of popular Java frameworks beyond Spring Boot. You'll explore the benefits of standardization and the choice between imperative and reactive programming, and learn about Java EE, Jakarta EE, and MicroProfile. The chapter also includes practical examples using Quarkus and Helidon to show how REST principles apply across different tools.

Chapter 12, *Deploying APIs*, guides you through the essential steps to move your Java RESTful APIs from development to production. You'll explore deployment preparation, containerization with Docker, and the use of PaaS platforms. This chapter focuses on practical, beginner-friendly workflows that serve as a solid foundation for more advanced DevOps practices in the future.

To get the most out of this book

This book is primarily aimed at backend developers working with, or interested in working with REST APIs. It also benefits architects and tech leads who want to dive deeper into topics like API monitoring, deployment, managing API growth in a healthy way, avoiding breaking changes, and improving resilience and security.

Download the example code files

The code bundle for the book is hosted on GitHub at `https://github.com/PacktPublishing/Mastering-RESTful-Web-Services-with-Java`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing`. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://packt.link/gbp/9781835466100`.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. For example: "To add the Spring Data dependency, include the following entry in your `pom.xml` file:".

A block of code is set as follows:

```
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
```

Any command-line input or output is written as follows:

```
curl -X 'GET' \
```

Bold: Indicates a new term, an important word, or words that you see on the screen. For instance, words in menus or dialog boxes appear in the text like this. For example: "To adhere to the **Don't Repeat Yourself (DRY)** principle, we must update the code from version 1."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book or have any general feedback, please email us at `customercare@packt.com` and mention the book's title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you reported this to us. Please visit `http://www.packt.com/submit-errata`, click **Submit Errata**, and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `http://authors.packt.com/`.

Share your thoughts

Once you've read *Mastering RESTful Web Services with Java*, we'd love to hear your thoughts! Please `click here to go straight to the Amazon review page` for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily.

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below:

`https://packt.link/free-ebook/9781835466100`

2. Submit your proof of purchase.
3. That's it! We'll send your free PDF and other benefits to your email directly.

Part 1

Steps to a Great API

This part lays the foundation for building robust **RESTful APIs** using Java. It introduces the key concepts of **REST** architecture, walks through exposing a real-world API with Spring Boot, and covers essential documentation techniques using **OpenAPI**. You'll also learn how to generate API code from specifications to streamline development and ensure consistency.

This part will cover the following chapters:

- *Chapter 1, Understanding RESTful Core Concepts*
- *Chapter 2, Exposing a RESTful API with Spring*
- *Chapter 3, Documenting Your API Effectively*
- *Chapter 4, Generating Code from OpenAPI*

1

Understanding RESTful Core Concepts

The history of web services is a fascinating journey through the evolution of the way that systems are built, distributed, and scaled.

Initially, the monolithic approach, which involved developing everything within a single application and machine, was the standard architectural method for many years. However, the need to divide software into distinct components with separate responsibilities has transformed how we design and implement systems.

The need to integrate distributed systems composed of software running on separate machines has existed since the early days of computing, and its importance is increasing, as the way we develop software is moving from monoliths toward more distributed architectures.

The global spread of the Internet, especially the World Wide Web, brought about the standardization of communication protocols, such as **Internet Protocol (IP)**, **Transmission Control Protocol (TCP)**, and **Hypertext Transfer Protocol (HTTP)**. The success of the World Wide Web and its support by different devices, operating systems, and applications brought the idea that the infrastructure of the web could be used for connecting applications in general. This is when the term **web services** was adopted for using web technologies to create APIs.

In this book, you will acquire the knowledge needed to be ready to develop and master the creation of RESTful web services, starting from the concepts to full implementation, following the best practices in the market.

This chapter will prepare you for the rest of your journey in this book, understanding what the **Representational State Transfer (REST)** architecture and RESTful API services are, how these terms differ from each other, principles, maturity levels, and guidelines, as well as the project you will be creating throughout this chapters to implement this knowledge.

The following topics will be covered in this chapter:

- Why REST?
- Principles of REST architecture
- Levels of a RESTful API
- Representing data with **JavaScript Object Notation (JSON)**
- The importance of guidelines for developing REST APIs
- Common REST API use cases
- Architecture impact on REST API design
- Alternatives to REST

Why REST?

To understand why REST is the dominating architectural style for most web services, we will need to understand what was available before the rise of REST and the challenges that made this change so important in the way that distributed services are built.

The pre-REST era

Before REST, the web services landscape was dominated by protocols such as **Simple Object Access Protocol (SOAP)** and **Extensible Markup Language–Remote Procedure Call (XML-RPC)**. These were powerful but complex standards that allowed for detailed communication between clients and servers. However, they were often seen as cumbersome due to their verbose nature and the strict requirements they imposed on developers.

SOAP, for instance, required developers to write extensive XML documents with specific calls and responses. It was notorious for its complexity and difficulty in debugging. Similarly, XML-RPC, while simpler than SOAP, still involved significant overhead for simple requests and responses (e.g., verbose and complex XML formatting, serialization and deserialization, a text-based protocol, and parsing complexity). Both SOAP and XML-RPC only used the HTTP protocol as a *transport* and duplicated, in their own ways, several features that the protocol offered and that were also supported by the existing web infrastructure.

To overcome these challenges and improve the way the systems communicated with each other, REST was created and has been widely implemented since its inception.

Understanding REST

REST was introduced in 2000 by Dr. Roy Fielding in his doctoral dissertation titled *Architectural Styles and the Design of Network-based Software Architectures*.

This architectural style was proposed as a more efficient, flexible, and scalable alternative to the existing standards of the time, such as SOAP and XML-RPC.

Dr. Fielding's dissertation aimed to simplify the way web services were created and consumed, leveraging the existing capabilities of the HTTP protocol.

The key principles of REST – **statelessness**, **cacheability**, **uniform interface**, and a **client-server architecture** – were designed to make web services more intuitive and aligned with the design of the web itself.

We will be covering each one of these principles in detail in the *Principles of REST architecture* section.

When we implement the REST architecture into web services, applying all these key principles, then we can say that we have a RESTful API. Let us understand this difference better in the next session.

Unpacking RESTful

RESTful APIs represent an approach to designing web services that adhere to the principles of REST, so they are not the same.

While REST provides the theoretical framework for building scalable and interoperable systems, RESTful APIs put these principles into practice, enabling developers to create robust and flexible APIs that are easy to understand, maintain, and extend.

The introduction of RESTful APIs marked a significant shift in web services since developers quickly adopted REST due to its simplicity and the way it facilitated the development of scalable and performant web applications. RESTful APIs became the backbone of web communication, powering everything from social media platforms to e-commerce sites.

Now that we have a clear understanding of REST and RESTful, let's dive deep into the principles of REST architecture. This will give us a clearer understanding of its key principles and how to achieve them.

Principles of REST architecture

Up to this point, we have only mentioned the key principles of REST. Let us dive deeper to understand these principles in more detail.

Uniform interface

The **uniform interface** is the cornerstone of any REST design, promoting a standardized way of interacting with a given set of resources. This principle encompasses four key constraints:

- **Identification of resources**: Each resource, whether it is a document, image, or service, is identified using a unique **uniform resource identifier** (**URI**)
- **Manipulation of resources via representations**: When a client possesses a representation of a resource, along with any attached metadata, it can modify or delete the resource on the server if it has the necessary permissions
- **Self-descriptive messages**: Each message contains enough information to describe how to process it, which may include the representation format and the desired state transitions
- **Hypermedia as the engine of application state (HATEOAS)**: Clients interact with a RESTful service entirely through hypermedia provided dynamically by application servers – a concept known as HATEOAS

Example: Imagine a library system where each book is a *resource* identified by an ISBN number (URI). When you want to borrow a book, you get a *representation* (a card with book details), which you use to check out the book. The library's checkout system tells you how to proceed (using *self-descriptive* messages), and the catalog guides you to related resources (*HATEOAS*), such as the author's other books.

Client-server separation

This principle enforces the separation of concerns by dividing the user interface concerns from the data storage concerns. This separation allows the client and server components to evolve independently, leading to a more flexible and scalable application architecture.

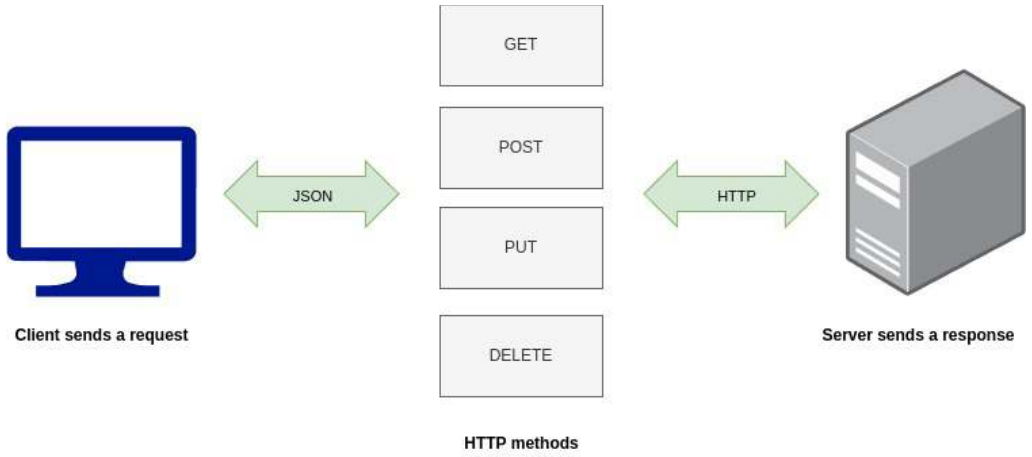


Figure 1.1 – Client-server diagram

Example: A user submits a form on a website with their data to finalize a checkout (client sending the request); this will be JSON using the `POST` HTTP verb and will be received by the server. The server will receive the data, create this new resource with the customer data, store the customer data, place a new order, and return the appropriate response to the user.

Statelessness

In a RESTful service, each request from a client to a server must contain all the information the server needs to fulfill the request. The server does not store any session state about the client, which means that each request is independent and isolated.

Example: Each time you order a coffee at a café, you provide your full order details. The barista does not need to remember your previous orders; they just make the coffee based on the current order alone.

Cacheability

Responses must, implicitly or explicitly, define themselves as cacheable or not. This helps improve the network's efficiency by reducing client-server interactions for frequently requested resources.

Note that caching brings the risk that the client may see an outdated version of the resource, especially if the resource changes frequently. However, we often accept the risk in exchange for better performance.

Example: A resource requested thousands of times per day is eligible for caching since this will drastically reduce the usage of **database (DB)** resources and improve the response time.

Layered system

A RESTful architecture may consist of layered hierarchies, which can include load balancers, caches, or authentication gateways. This layered system ensures that a client cannot ordinarily tell whether it is connected directly to the end server or to an intermediary along the way. The layered system is one of the key constraints of the RESTful architecture and it offers several benefits:

- **Modularity**: The layered system allows you to organize your application into logical layers, each with a specific role and responsibility. This makes the system easier to understand, develop, and maintain.
- **Separation of concerns**: Each layer can focus on its own tasks. For example, an authentication layer can handle all aspects of authentication and nothing else. This separation of concerns leads to cleaner, more maintainable code.
- **Interchangeability**: If a layer is designed and implemented in a modular way, it can be replaced or upgraded without affecting other layers. This is particularly useful when you want to update or improve a specific part of the system.
- **Scalability**: You can scale different layers independently based on their individual load and performance requirements. For example, if your application layer is experiencing heavy load, you can add more servers to that layer without having to scale your database layer.
- **Security**: By segregating the system into layers, you can apply appropriate security controls to each layer. For example, you can put a firewall between layers to control traffic and protect sensitive layers from potential attacks.

Example: When you send a letter, it goes through various postal offices (layers) before reaching the destination. Similarly, a RESTful request might pass through security checks and load balancers without the client's knowledge.

Code on demand

Servers can extend client functionality by transferring executable code. This is an optional feature that is used sparingly in the context of APIs because it is difficult to implement reliably and securely.

Example: A web application could ask the browser to download and execute a calendar widget. The widget is a piece of executable code that extends the functionality of the browser.

Now that you understand the principles of REST architecture, let's learn how to achieve glory in a RESTful API through the levels explained by the Richardson Maturity Model.

Levels of a RESTful API

One way to understand the concept of RESTfulness is through the **Richardson Maturity Model**, which outlines various levels of adherence to REST principles in API design.

Named after Leonard Richardson, who introduced it in 2008, the model consists of four levels, each representing progression toward more RESTful design:

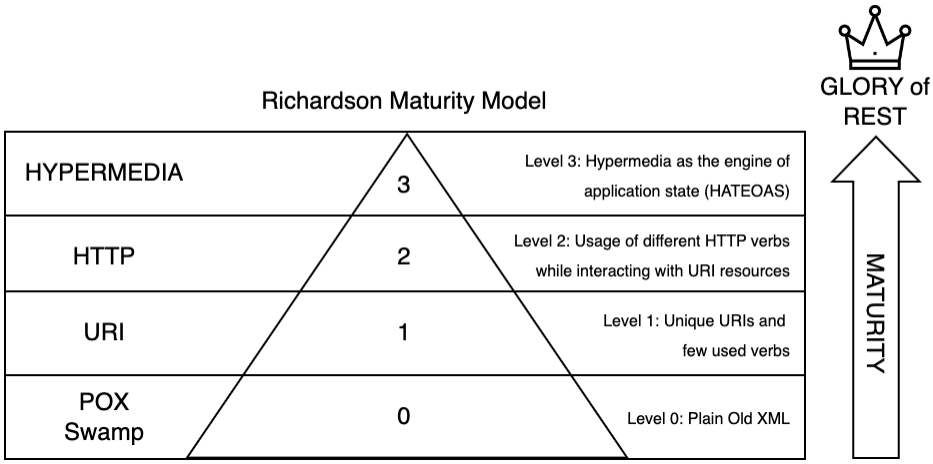


Figure 1.2 – Richardson Maturity Model diagram

Level 0 – the swamp of Plain Old XML (POX)

At this level, APIs are characterized by a single URI for all operations, typically using HTTP `POST` requests. The payloads are often XML-based, and there is no distinction between different resource types or HTTP methods. This level lacks the fundamental principles of REST, such as resource identification and separation of concerns.

Level 1 – resources

This level introduces the concept of resources, where each entity in the system is represented as a unique URI. However, HTTP methods are not fully utilized, and operations are typically performed using a combination of HTTP `POST` and `GET` requests.

While resources are identified, the API lacks the uniform interface and predictability associated with RESTful design.

Level 2 – HTTP verbs

At this level, APIs start leveraging HTTP methods – `GET`, `POST`, `PUT`, `DELETE` – to perform **CRUD** (**Create, Read, Update, Delete**) operations on resources.

Each HTTP method corresponds to a specific action, providing a more intuitive and standardized interface for interacting with the API. However, hypermedia links are still missing, limiting the discoverability and flexibility of the API.

Level 3 – hypermedia controls (HATEOAS)

The highest level of RESTfulness, *Level 3*, introduces HATEOAS. In addition to leveraging HTTP methods, RESTful APIs at this level include hypermedia links in responses, allowing clients to navigate the API dynamically. By providing links to related resources and actions, HATEOAS reduces coupling between client and server, enhancing flexibility and scalability.

Each level of the Richardson Maturity Model represents progression toward more RESTful design, with *Level 3* being the ideal endpoint where APIs fully embrace the principles of REST, including resource identification, uniform interface, and hypermedia-driven navigation.

Most real-life APIs do not reach *Level 3*, but when they follow as many REST principles from the lower levels as possible, they benefit significantly from standardization and leverage the capabilities of the HTTP protocol.

Many APIs, instead of implementing hypermedia controls, publish machine-readable (but static) specifications of their supported operations using the OpenAPI standard.

Now, let's examine one of the most notable characteristics of REST services: how requests and responses are represented (typically in JSON), and why JSON is preferred over XML.

Representing data with JSON

JSON is a simple data format which is designed for easy reading and writing for humans, while also being straightforward for machines to process and create. One small feature that a human user may miss is that JSON does not support comments.

It is based on a subset of the JavaScript programming language and is completely language-independent, with parsers available for every programming language.

A JSON document is built from two structures:

- **A collection of name/value pairs**: In various languages, this is implemented as an object, record, struct, dictionary, hash table, keyed list, or associative array
- **An ordered list of values**: In most languages, this is implemented as an array, vector, list, or sequence

Here is a simple example of a JSON document for a book object:

```
{
  "book": {
    "title": "Mastering RESTful JSON Essentials",
    "author": "Multiple Authors",
    "published": true,
    "edition": 1,
    "tags": ["programming", "web", "json"]
  }
}
```

The value of the `"book"` field is the object structure containing name/value pairs. The value of the `"tags"` field is a list of ordered unnamed values.

So, why is JSON preferred for RESTful APIs?

JSON and XML are both formats used for data interchange, but they have distinct characteristics that can make one more suitable than the other for certain applications, particularly RESTful APIs. Here is a brief overview of why JSON is often preferred over XML for RESTful services:

- **Performance**: JSON's lightweight nature means it can be parsed more quickly than XML, which is crucial for the performance of RESTful APIs
- **Simplicity**: JSON's simpler syntax makes it easier to read and write and to parse and generate programmatically
- **Web-friendly**: Given its compatibility with JavaScript and the fact that modern web development heavily relies on JavaScript, JSON aligns well with web technologies

While both JSON and XML have their places, JSON's efficiency, simplicity, and web-friendliness make it the preferred choice for many developers when it comes to RESTful API development. There are also binary structured data interchange formats, such as ProtoBuf, that provide better performance than JSON as they remove the need for text parsing, but the binary formats are not as widespread and ubiquitously supported.

As shown earlier, in RESTful services, JSON plays a crucial role as the medium of data exchange. REST does not prescribe any particular language to be used to represent the data of the resources. However, thanks to its ease of use and ubiquitous support, JSON is a natural choice for most RESTful APIs. When clients make requests to servers, they can send JSON in the body of the request. The server then processes this JSON, performs the required operations, and can also send back JSON in the response body. This consistent format allows for clear and structured data interchange.

JSON's functionality in REST services includes the following:

- **Serialization**: Converting an object into a JSON string to send it over a network. For example, see the following:

```
String serializedJson = objectMapper.writeValueAsString(
    bookDetails);
```

- **Deserialization**: Converting a JSON string received over a network into an object. For example, see the following:

```
BookDetails bookDetails = objectMapper.readValue(
  json, BookDetails.class);
```

- **Data interchange**: Acting as the medium for exchanging data between client and server.

These examples show the usage of `ObjectMapper` from the Jackson library to serialize and deserialize JSON, with the JSON book example earlier in this section. This library is already included in the Spring web framework but can also be imported separately if needed.

We have seen the importance of JSON and the change this has introduced in comparison to the previously used XML. Now, you will learn one of the crucial steps to master the development of great REST APIs, and that is to adhere to well-proven guidelines, as explained in the next section.

The importance of guidelines in REST APIs

Guidelines serve as a roadmap for developers, outlining best practices and standards that should be followed during the API development process. They are a set of rules that aim to prevent common pitfalls and promote excellence in API design. Here are some of the most important aspects of following well-established guidelines:

- **Consistency and predictability**: Developers ensure a level of consistency that makes APIs predictable and easier to understand. This uniformity is essential for both the developers who create APIs and the users who consume them.

- **Interoperability**: Guidelines often emphasize the use of standard protocols and data formats, which fosters interoperability across different platforms and technologies. This is particularly important in a diverse ecosystem where APIs serve as the bridge between various software components.
- **Scalability**: Well-designed APIs that adhere to guidelines are more likely to be scalable, handling increased loads and accommodating growth without requiring significant re-design or refactoring.
- **Security**: This is fundamental in API development, and guidelines provide strategies to secure APIs against common threats, ensuring the protection of sensitive data and services.

Market relevance of following guidelines

Adherence to guidelines has a direct impact on the market. Well-designed APIs that follow guidelines can lead to the following outcomes:

- **Increased adoption**: APIs that are easy to understand and integrate can quickly gain popularity among developers, leading to widespread adoption and a larger user base
- **Enhanced interoperability**: In markets such as healthcare, where data sharing is vital, guidelines ensure that different systems can communicate effectively, thereby improving patient care and operational efficiency
- **Regulatory compliance**: Especially in regulated industries, following guidelines helps organizations comply with legal standards, avoiding penalties and fostering trust among stakeholders

Examples of guideline-driven success

Here are some of the most successful guidelines that you can use as a reference:

- The Microsoft Azure REST API Guidelines at `https://github.com/microsoft/api-guidelines/blob/vNext/azure/Guidelines.md`
- The HL7 FHIR guidelines for healthcare systems at `https://fhir.org/`
- Community-driven API enhancement proposals at `https://aep.dev/`

Many more design guidelines are available on the API Stylebook Design Guidelines page from companies such as PayPal, Adidas, Heroku, and others at `https://apistylebook.com/design/guidelines/`.

In this book, we will follow the Microsoft Azure REST API Guidelines.

Next, we will look at some of the most common use cases where REST APIs are used and implemented.

Common REST API use cases

Here, we will explore some common use cases for REST APIs with examples that illustrate their practical applications:

- **Integration with third-party services**: Syncing data between different platforms, such as **Customer Relationship Management** (**CRM**) and **Enterprise Resource Planning** (**ERP**) systems
- **Social media services**: Enabling social interactions and content sharing on various platforms
- **E-commerce transactions**: Managing product listings, orders, and payments
- **Internet of Things (IoT)**: Connecting and controlling smart devices remotely
- **Health and fitness tracking**: Aggregating data from various health and fitness devices

In the next section, we will discuss the impact of the architectural design of REST APIs in a complete system and its clients.

Architecture impact on REST API design

The need to design and implement APIs can occur in many different contexts. The goal we want to achieve with an API and the role it plays within the architecture of the application or system being developed influences the relative importance of the various API aspects described in the respective chapters of this book. The unique mix of architectural aspects also impacts the choice of approaches and techniques to implement APIs.

The following sections present the most common patterns.

Cross-organizational public APIs

APIs consumed beyond organizational boundaries are the closest to the original application of web technologies.

Controlling all API clients is unfeasible. Public APIs with numerous independent consumers frequently prioritize backward compatibility over other design goals. The introduction of incompatible changes incurs a substantial cost associated with supporting multiple API versions at the same time.

Lastly, public APIs require higher security standards compared to internal APIs.

Frontend-to-backend APIs developed by a single team

Most contemporary applications comprise a frontend component operating on an end-user device (web browser or mobile application) and a backend component, typically deployed on a cloud or other server infrastructure.

In numerous instances, both the backend and its corresponding frontend are developed within a single agile development team. In this scenario, API changes can be mirrored on both ends swiftly, and the importance of backward compatibility may be lower.

Nonetheless, even in this situation, there may be clients beyond the team's control utilizing an older API version, for example, mobile applications lacking recent updates.

Frontend-to-backend APIs are public APIs as well because the clients are beyond our control, necessitating their treatment as such from a security standpoint.

APIs interconnecting microservices

Microservice development teams should maintain autonomy to progress rapidly. They need to pay attention to the documentation and backward compatibility of their APIs to avoid disrupting other services connected to them. Unlike public APIs, within an organization, it is feasible to track the usage of internal microservice APIs and their versions.

Solutions managing the formal specifications of various APIs utilized within an organization, known as **schema registries**, can assist in balancing the pace of development and the stability of the overall business solution.

With multiple services implementing multiple APIs within a single organization, it often proves beneficial to delegate some of the API responsibilities to infrastructure components, such as a service mesh.

In the next section, you will learn about API styles that may be preferable to REST in some scenarios. Remember that it does not matter how well you implement a RESTful API if you use it in the wrong place.

Alternatives to REST

This section presents some of the existing alternatives in the market to REST APIs and their comparison. It is good to know that REST is not always the best option for every case, and it is wise to be able to differentiate between the existing options available. Let us mention some examples here:

Remote Procedure Call (RPC)

RPC is an API style that obfuscates the existence of an API, rendering it akin to a standard procedure invocation (or a method in object-oriented languages) in a specific programming language. This style encompasses SOAP web services and XML-RPC.

Other notable representatives include CORBA (utilized by Enterprise Java Beans) and gRPC (a new one with support for numerous programming languages). gRPC delivers performance benefits due to its use of efficient binary message encoding.

RPC APIs are optimally suited for integrations where the number of clients is limited, and the API provider can exert control over them.

Their disadvantages are as follows:

- High coupling between the client and the server
- Limited or no compatibility between different API versions
- Clients are forced to use a less common and more complex technology
- Familiarity with the well-known HTTP protocol cannot be leveraged to understand the semantics of the operations

GraphQL

Analogous to SQL, **GraphQL** defines a data query and mutational language, enabling the client to specify which properties and nodes of a graph-structured data source should be returned by the server.

Like SOAP, GraphQL exclusively employs the HTTP `POST` method to tunnel all requests. GraphQL affords the client the liberty to execute operations using unexpected combinations of data elements. This can result in an excessive amount of logic on the client side, leading to duplication as there are typically multiple clients for a single server.

If there is a requirement to provide data in a different structure and detail level for different clients, with REST APIs, the problem can be addressed by doing the following:

- Incorporating parameters in the request (for instance, query parameters)
- Creating multiple APIs on the server side, specialized for the client type, by using an architecture known as **Backends for Frontends** (**BFF**)

Messaging (event) APIs

Asynchronous communication via messaging platforms, such as Kafka or **Java Message System (JMS)**, as opposed to the synchronous request-response style of web APIs, is commonly employed for connecting applications and services within an organization and used in conjunction with REST APIs.

While providing benefits such as loose coupling, low latency, and more, asynchronous communication is also more complex and indirect.

Event interfaces, such as the web APIs, need to be well documented and they face challenges of API evolution and compatibility. AsyncAPI, the specification standard utilized for asynchronous APIs, is inspired by, and shares considerable overlap with, the more mature OpenAPI, which will be discussed later in this book.

In the next section, we introduce the project we will be creating during this book to apply these REST concepts in depth.

About the project

In this book, we will develop a project to apply the concepts mentioned in this chapter, having hands-on practice and experience. We will build two APIs that will communicate with each other.

The first API will be a Product API, which will be introduced in *Chapter 2*. This API, as its name suggests, is focused on product operations, such as reading product data and adding new products.

The second API will be the Order Management API. It will be introduced in *Chapter 4* and will hold the orders containing the products stored in the first API, so they will communicate with each other.

Throughout the chapters, you will be able to implement these APIs, apply most of the best practices mentioned in this chapter, evolve these, document, test, and version to guarantee backward compatibility, implement tracing tools, and deploy to the cloud.

Summary

In this chapter, you learned the difference between REST and RESTful, and why the REST approach is the preferred method for building API web services compared to previous methods used before.

We have investigated the principles of REST architecture, the levels of using the Richardson Maturity Model, and the existing alternatives to REST.

Then, we looked at the role of JSON in REST services and the importance of following well-defined guidelines proven by numerous projects while developing REST web services.

Lastly, we investigated the architectural impact of REST API design and the project that will be built throughout this book in the following chapters.

This chapter presents the basics for you to get started with the upcoming content that will be presented and built throughout this book.

In the next chapter, we will explore the design and practical implementation of RESTful APIs using Spring Boot.

2

Exposing a RESTful API with Spring

To implement REST APIs, in most chapters of this book, we are going to use **Spring Boot**. As it is a well-known and popular framework, it is very likely that you are familiar with it or have used it before.

However, the general API design principles are easily transferable to other Java technologies. You will learn more about how we can apply these principles to other technologies in *Chapter 11*.

In this chapter, we will discuss the process of designing a RESTful API. We will also concentrate on the practical implementation of the API using the Spring Framework, a popular choice for building RESTful APIs. By the end of this chapter, you will have the knowledge to design and create a RESTful API following the best practices.

In this chapter, we will be covering these topics:

- Designing the product API
- API implementation using Spring Boot

Technical requirements

In this chapter, we will implement an example product API. To be able to follow along and use the code examples as they are printed in the book, you should have the following:

- Intermediate knowledge of the Java language and platform
- At least a basic knowledge of Spring Boot or a similar framework

- Java 21 and Maven 3.9.0 installed
- Basic knowledge of a tool for calling REST APIs, such as `curl`, is recommended

In this chapter, we are going to apply REST principles to create our API; you can access the code for this chapter on GitHub at `https://github.com/PacktPublishing/Mastering-RESTful-Web-Services-with-Java/tree/main/chapter2`.

Designing the product API

The product API we will develop is an API for managing products. Our API will offer various operations, which will be detailed during the design phase.

Spending an adequate amount of time on properly designing the API before starting to write the code can save a lot of time later, preventing costly and risky refactoring.

The design stage should include the following:

- **Defining the requirements**: Understanding the use cases and who will use the API is essential to creating an API that contains everything that is needed and nothing more. A good understanding of the requirements lets you avoid breaking changes for as long as possible. In *Chapter 5*, we will talk about how you can evolve your API and ensure backward compatibility.
- **Identifying the resources**: Resources are usually domain entities such as users or products. Relationships between multiple entities are often represented by a hierarchical URI structure.
- **Defining the resource structure**: After identifying the resources, it is necessary to define the resource fields and relationships.
- **Designing the endpoints**: With the resources and the domain defined, the next step is to identify the endpoints that should be exposed, how they should be exposed, and what HTTP methods should be used for what purpose.
- **Error handling**: Having clear error responses with standard error codes helps the client to react to the error correctly.
- **Security**: It is essential to prevent malicious actors from accessing resources that they are not authorized to. You will learn more about API security in *Chapter 7*.

An endpoint is a specific URI that enables clients to interact with a server via an API to perform a specific function. It represents an operation that can be executed using a unique path and the corresponding HTTP method.

Based on these design steps, you can proceed directly to implementing the API in code using Java. This is called the **code-first approach**, usually representing the fastest path to a working API.

A different approach, called **specification-first**, will be explained in *Chapter 3*.

Defining the requirements

The API requirements can be divided into functional and non-functional:

- **Functional requirements**: These describe the specific functions and features that the software must perform. Examples include data validation, data processing, and system interactions.
- **Non-functional requirements**: These are also known as quality attributes or software quality requirements that specify the qualities or characteristics that the software must possess. Examples include performance (response time and throughput), reliability, security, scalability, usability, and maintainability.

In this chapter, we will only work on the functional requirements. We will get to some of the non-functional requirements in *Chapters 7 and 10*.

As mentioned earlier, our example REST API will be a product API and will have the following requirements:

- **Product creation**: The API should allow users to create new products by providing necessary information such as the SKU (Stock Keeping Unit, in the context of this book it is the unique product ID), name, description, and price
- **Product retrieval**: Users should be able to retrieve information about all products available in the system
- **Product details**: The API should provide endpoints to retrieve detailed information about a specific product, identified by its SKU
- **Product update**: Users should be able to update existing product information, such as name, description, or price

- **Product description update**: The API should provide an endpoint to update the description only
- **Product deletion**: The API should allow users to delete a product from the system, identified by its SKU
- **Unique SKU constraint**: The API should enforce a constraint to ensure that each product has a unique SKU, preventing duplicate products with the same identifier

Now that we have identified the requirements, the next step is to identify our resources.

Identifying the resources

A REST API can have one or more resources, and it can also have a hierarchical URI structure. A hierarchical URI structure is a way of organizing resources in a URL hierarchy that reflects the relationships between those resources. In a hierarchical URI structure, resources are arranged in a tree-like format, where each segment of the URL represents a level of the hierarchy.

URI or URL?

In the context of REST APIs, we often use the acronyms **URI** (which stands for **Uniform Resource Identifier**) and **URL** (which stands for **Uniform Resource Locator**) interchangeably. To be precise, URLs are a subset of URIs. There is a second subset of URIs called **URNs** (which stands for **Uniform Resource Names**). The advantage of a URL over a URN is that besides being a unique identifier of the resource, it also contains information that can be used to locate and access the resource: protocol, hostname, and path. The path part of a URL supports a hierarchical structure such as the sequence of folders and a filename that you know from filesystem paths.

Let's take the following as an example:

```
https://example.com/blog/posts/12
```

In this URI, we have the following:

- `https://example.com` is the base URL
- `/blog` represents the top-level resource, indicating that we are accessing the blog section of a website
- `/posts` is a sub-resource under the `/blog` resource, representing a collection of blog posts
- `/12` is a specific blog post identified by its unique identifier

In our case, we have only one resource, which is the product, so we have only `/products`.

Resources should be nouns; they should not be verbs. For example, `/products/create` should not be used; instead, use the corresponding HTTP method for this action. This is essential for adhering to level 2 of the Richardson Maturity Model.

In English, many verbs can be used as nouns too, for example, `/products/quote`. When we make a `POST` request to this resource, it means *create a quote*. The verb part of the sentence is represented by the correct HTTP method.

Resources map to business entities and the API usually works with multiple entities of the same type. When creating or listing resources (entities) of a particular type, we are working with a collection of objects. For collection resources, we use the plural form of the noun – for example, `products` instead of `product` because we want to support operations on the collection of products.

RFC 3986 defines a URI as a compact sequence of characters that identifies an abstract or physical resource. According to RFC 1738, a URL is a specific type of URI that represents the location of a resource accessible via the internet. Thus, a URI can be further classified as a locator, a name, or both, with the term URL referring to the subset of URIs. This means that all URLs are URIs, but not all URIs are URLs. RFC 2141 further defines URNs as another type of URI that names a resource in a persistent, location-independent manner.

Now that we have identified the resources, the next step is to define the resource structure.

Defining the resource structure

After identifying the resources, we should identify the attributes of the resources and any relationships that are important in the API. The attributes represent the data fields associated with the resource. Consider the data types, constraints, and any required or optional attributes for each resource. A relationship indicates how the resources are related to each other and whether they have any hierarchical or nested relationships. For example, a user may have multiple orders associated with them, or a product may belong to a specific category.

In our case, we have only one resource, which is the product, which should have these attributes:

- `name`: The name of the product
- `sku`: The unique key of the product
- `description`: A description of the product
- `price`: The price of the product

We can also define the rules for these attributes in this step:

- `name`: Required field, with a string with a length between 3 and 255 characters
- `sku`: Required field, with the pattern AA99999
- `description`: Required field, with a string with a length between 10 and 255 characters
- `price`: Required field, with a value greater than 0

After this step, we can design the endpoints.

Designing the endpoints

To design the endpoints, it is important to know the HTTP methods and the HTTP status codes. Before talking about this, though, we should go one step back and understand how it started.

The HTTP methods and principles we are going to share here follow the guidelines from Microsoft mentioned in the previous chapter. Any REST guidelines should adhere to the internet standard known as RFC 9110, the updated version of RFC 2616. **RFC** stands for **Request for Comments**. RFC specifies HTTP, TCP, IP, SMTP, and many other vital internet protocols. Fielding's dissertation, titled *Architectural Styles and the Design of Network-based Software Architectures*, was an important source for RFC 2616, defining the HTTP/1.1 protocol.

HTTP methods (verbs)

HTTP defines several methods (also known as verbs) that indicate the action to be performed on a resource. The most used HTTP methods in RESTful APIs are the following:

- `GET`: The `GET` method requests are used to retrieve resource representations, and they should not alter the server state. They are safe and idempotent, meaning they do not have any side effects on the server and can be repeated without changing the server state. (Think about the `get` method of a `Map` in Java.)
- `POST`: The `POST` method requests are used to create new resources or generally submit data to be processed by the server. They are non-idempotent, meaning that each request is unique and may have side effects on the server. (Think about the ++ operator in Java; the result will be different if it is evaluated multiple times.)

- `PUT`: These requests are typically used to create or update the entire representation of a resource. They are idempotent, meaning that sending the same request multiple times should have the same effect as sending it once. (Think about the `put` method of a `Map` in Java.)
- `PATCH`: This is like `PUT` but used to apply partial modifications to a resource. It is often used when you want to update only certain fields of a resource. (Think about a setter method of a Java Bean/POJO.)
- `DELETE`: This method is used to remove a resource from the server. The method is idempotent, meaning that sending the same request multiple times should result in the same state on the server.

HTTP status codes

HTTP status codes are three-digit numbers that a server sends back to the client after receiving a request. They indicate the outcome of the request and provide information about the status of the server or the requested resource. All HTTP status codes are grouped into the following categories:

- `1xx Informational`: This indicates that the request was received and is being processed.
- `2xx Success`: This indicates that the request was received, understood, and processed successfully by the server.
- `3xx Redirection`: This indicates that further action is needed to complete the request. The client may need to redirect to a different URI.
- `4xx Client Error`: This indicates that there was an error in the client's request.
- `5xx Server Error`: This indicates that there was an error on the server side while processing the request.

Now that we understand HTTP methods and statuses, we can define the API endpoints.

Defining the endpoints for our API

Let's define the endpoints for the resource we identified in the previous step. We will use the HTTP methods with the `products` resources to ensure we can perform all the necessary operations as outlined in the requirements phase.

GET /products

This endpoint will be responsible for returning a list of products. We should return a `200 (ok)` HTTP status with the products.

PUT /products/{id}

This endpoint will be responsible for creating or updating the product. `PUT` is an idempotent method, so we can call it many times and the result will be the same. In case of success, we should return `201 (created)` if the product doesn't exist and `200` if the product already exists.

We could also return `202 (accepted)` if the product will be processed in the future as an asynchronous task, which is not the case here.

If we decided to use the `POST` method, we would need to choose between throwing an exception if the product already exists – this exception could be `409 (conflict)` – or creating another product. In our case, because the SKU is not generated by the application, it is an attribute passed to the API, and we cannot have two products with the same SKU, so we will need to return `409` and have another endpoint that would be responsible for updating the resource as it is a requirement. Instead of doing this, we can use the `PUT` method, which can create or update the resource.

According to RFC 9110, Section 9.3.4, "The fundamental difference between the POST and PUT methods is highlighted by the different intent for the enclosed representation. The target resource in a POST request is intended to handle the enclosed representation according to the resource's own semantics, whereas the enclosed representation in a PUT request is defined as replacing the state of the target resource. Hence, the intent of PUT is idempotent and visible to intermediaries, even though the exact effect is only known by the origin server".

Proper interpretation of a PUT request presumes that the user agent knows which target resource is desired. To get a better insight, please refer to `https://www.rfceditor.org/rfc/rfc9110.html#section-9.3.4`.

DELETE /products/{id}

This endpoint should remove the product. We can return `204 (no content)` in case the product is removed. Even if the product does not exist, we can return `204`, as the method is idempotent, meaning that sending the same request multiple times should result in the same state on the server.

PATCH /products/{id}

This endpoint should be used to update the description of the product. As it is a partial update of a resource, it is recommended to use `PATCH`. The `PATCH` method was introduced in RFC 5789 as a partial update that doesn't need to be idempotent. If the product does not exist, we can return a `404 (Not found)` error. In case of success, we can return the updated product data and a `200` HTTP status.

GET /products/{id}

This endpoint is responsible for returning the details of a single product by the ID. If the product does not exist, we can return a `404 (Not found)` error.

In case of success, we should return the product representation and a `200` HTTP status.

Now that we have defined our endpoints, let's see the possible errors that we may encounter and how we can prepare for them effectively.

Error handling

Preparing for common issues by defining possible errors can help us implement a more reliable API. We have already discussed some errors in the previous step; however, let's get deep into it and see the most common HTTP status codes in the `4xx` range.

The most common HTTP status codes in the `4xx` range, which indicates client errors, are the following:

- `400 Bad Request`: This status code indicates that the server could not understand the client's request due to invalid syntax or a malformed request message.
- `401 Unauthorized`: This status code indicates that the client needs to authenticate itself to access the requested resource. It typically occurs when the client fails to provide proper authentication credentials or access tokens.
- `403 Forbidden`: This status code indicates that the client is authenticated, but it does not have permission to access the requested resource. It may be due to insufficient permissions or access control restrictions imposed by the server.
- `404 Not Found`: This status code indicates that the server could not find the requested resource. It is commonly used to indicate that the URI provided by the client does not correspond to any known resource on the server.
- `405 Method Not Allowed`: This status code indicates that the HTTP method used by the client is not supported for the requested resource. For example, trying to use a `POST` request on a resource that only supports `GET` requests would result in a `405 Method Not Allowed` response.

- `409 Conflict`: This status code indicates that the request could not be completed because of a conflict with the current state of the resource. It typically occurs when the client attempts to create or update a resource but the server detects a conflict with the resource's current state; for example, when two requests attempt to update the same product's details concurrently, a `409` status can be returned if one request's changes conflict with the other due to a version mismatch.
- `422 Unprocessable Entity`: This status code indicates that the server understands the request but cannot process it due to semantic errors or validation failures in the request payload. It is often used to indicate validation errors in the request data, for example, trying to buy a product that is out of stock.
- `429 Too Many Requests`: This status code indicates that the client has exceeded the rate limit or quota imposed by the server for the number of requests allowed within a certain time period. It is often used to prevent abuse or excessive usage of server resources by limiting the rate of incoming requests from a single client or IP address. This error is typically handled by an API gateway using a rate-limiting strategy, as we'll cover in *Chapter 6*.

In our API implementation, we should be able to handle some errors:

- When a product does not exist, we should return `404 Not Found`
- When the payload does not meet the requirements, for example, if a negative number is provided for the product price, we should return `400 Bad Request` with the details

Some error conditions (for example, an unsupported method leading to `405 Method Not Allowed`) are handled automatically by the framework.

The security-related `401` and `403` HTTP status codes will be covered in *Chapter 7*.

In the upcoming chapters, we will take on security and documentation as separate topics and address them head-on. However, for the development iteration represented by this chapter, we have completed our design phase. Moving forward, we will focus on implementing our product API and ensuring that we can meet the requirements that we defined.

API implementation using Spring Boot

Spring Boot is the most popular Java framework for microservice applications. It provides embedded servlet containers such as **Tomcat**, **Jetty**, and **Undertow**. Embedding a container allows you to package your application as an executable JAR file that can be run directly without needing to deploy your application to a separate application server. We are going to use Tomcat as it is the default.

Our application will provide some endpoints and will use an SQL database. We will use **H2**, which is a lightweight and open source database. It can function as both an in-memory and filesystem database, making it easily usable in development environments without the need for additional infrastructure.

We are going to use a Spring Data dependency to integrate with the persistence layer, but since this is not the focus of the book, we won't delve deep into it. We recommend the book *Persistence Best Practices for Java Applications, by Otávio Santana and Karina Varela, Packt Publishing*, which talks more about the persistence layer. To add the Spring Data dependency, include the following entry in your `pom.xml` file:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

As our API is a web application, we are also going to use Spring Web to create our REST endpoints and Bean Validation to validate the user input, which we are going to talk about later in this chapter. We will use the **Clean Architecture** (`https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html`) design for our API.

Clean Architecture was created by Robert C. Martin, who was influenced by other well-known architectures, such as the Onion Architecture by Jeffrey Palermo and Hexagonal Architecture (a.k.a. ports and adapters) by Alistair Cockburn.

Clean Architecture shares a similar goal with these other architectures: the separation of concerns by dividing the software into layers. The key distinction of Clean Architecture is its clear definition of layers. While other architectures provide less precise layer definitions, Clean Architecture distinctly defines four layers, each with specific roles and responsibilities.

We are going to use Clean Architecture because its multiple layers allow for a clear separation of concerns, facilitating better organization and understandability. *Figure 2.1* shows the layers defined by Clean Architecture:

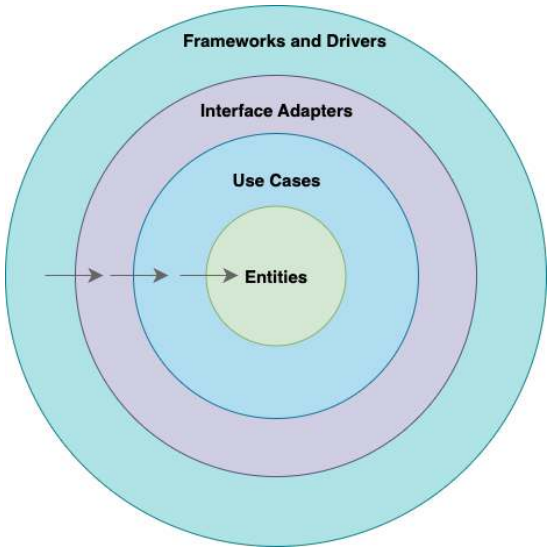


Figure 2.1 – Clean Architecture layers

Let's break down each of these layers:

- **Entities**: This layer contains the business entities or domain objects of the application.
- **Use cases**: This layer contains the application-specific business logic or use cases. This layer represents the application's behavior in terms of the actions or operations that users can perform.
- **Interface adapters**: This layer is responsible for adapting the application's internal representation to external interfaces such as **user interfaces (UIs)**, REST APIs, databases, or third-party services.
- **Frameworks and drivers**: This layer includes libraries, frameworks, and infrastructure code that handle external concerns such as UI rendering, database access, web servers, and external APIs.

In our application, we are going to use only three layers: entities, use cases, and interface adapters. This is because for our application, we decided we do not want to decouple from the Spring Framework completely. In our case, we prefer having direct access to the many useful features the Spring Framework provides. We accept coupling with the framework while reducing the complexity of creating a layer to separate it.

The layers are defined to achieve loose coupling, ease of maintenance, and reusability. Robert C. Martin says that we can have more layers, and these layers are only schematic. The main principle of layered architectures is to ensure that dependencies cross the boundaries in only one direction; an inner layer should not depend on an outer layer.

Now that we have defined the architecture and also the technologies, let's build our product API.

Creating the endpoints of our product API

As mentioned previously, to create our product API, we are going to use Spring Boot. We recommend using the latest version of Spring and Java (JDK); we will be using the following versions:

- Java 21
- Spring Boot 3.2.5

Before starting to create our API, we will define an interface with all the expected endpoints:

```
public interface ProductsApi {
    ResponseEntity<ProductOutput> createOrUpdateProduct(String productId,
        @Valid ProductInput productInput);
    ResponseEntity<Void> deleteProduct(String productId);
    ResponseEntity<ProductOutput> editProductDescription(String productId,
        @Valid ProductDescriptionInput input);
    ResponseEntity<ProductOutput> getProductById(String productId);
    ResponseEntity<List<ProductOutput>> getProducts();
}
```

This "`ProductsAPI`" interface contains all the expected endpoints defined in the design step.

Because we are following clean architecture guidelines, we defined three **data transfer objects (DTOs)** for input and output: `ProductOutput`, `ProductInput`, and `ProductDescriptionInput`. Utilizing these DTOs enables us to modify them without affecting our domain classes.

To create our product API, we will have to define a Spring controller. This class will be managed by Spring to handle requests to our endpoints. Our API class will implement the `ProductsApi` interface, and we will need to add two annotations, `@RestController` and `@RequestMapping("/api/products")`:

```
@RestController
@RequestMapping("/api/products")
public class ProductsApiController implements ProductsApi {
    private final ProductsQueryUseCase productsQueryUseCase;
```

```
    private final ProductsCommandUseCase productsCommandUseCase;
    private final ProductMapper productMapper;
    public ProductsApiController(
        ProductsQueryUseCase productsQueryUseCase,
        ProductsCommandUseCase productsCommandUseCase,
        ProductMapper productMapper) {
    this.productsQueryUseCase = productsQueryUseCase;
    this.productsCommandUseCase = productsCommandUseCase;
    this.productMapper = productMapper;
    }
  ...
}
```

`@RestController` is used to create a bean managed by Spring. This controller will handle the endpoint requests mapped by the `@RequestMapping("/api/products")` annotation, meaning that all requests whose URI matches the `/api/products` pattern will be handled by our controller.

Using `@RestController` is recommended for creating RESTful APIs with the Spring Framework. It is a shortcut for the combination of the `@Controller` and `@ResponseBody` annotations.

`ProductsQueryUseCase` and `ProductsCommandUseCase` are the implementations of our use cases. We are segregating the query and command responsibilities using the **Command Query Responsibility Segregation** (**CQRS**) pattern. This pattern is commonly used in software architecture to separate the responsibilities of reading data (queries) from writing data (commands).

CQRS is useful in systems where read and write operations have different characteristics or performance requirements. It promotes a clear separation of concerns, leading to more maintainable and scalable architecture. By segregating these responsibilities, we can later decide to break the API into two microservices and scale them separately.

The `ProductMapper` class uses the `MapStruct` library to transform one object into another object; in our case, we are mapping the data from our domain to our DTO. Let's see how we can implement it:

```
@Mapper(componentModel = "spring")
public interface ProductMapper {
    ProductOutput toProductOutput(Product product);
}
```

The implementation of `ProductMapper` is generated during the compilation phase.

Implementing the product API endpoints

In this section, we will implement the `ProductsApiController` class. This will involve adding all the endpoints we defined in the previous sections, enabling full functionality for creating, retrieving, updating, and deleting products in our API.

Create or update product endpoint

To implement this endpoint, we'll need to use Spring annotations. For the create operation, it's advisable to employ either `@PutMapping` or `@PostMapping`:

```
    @PutMapping(value = "/{productId}")
    @Override
    public ResponseEntity<ProductOutput> createOrUpdateProduct(
        @PathVariable("productId") @ValidSku String productId,
        @Valid @RequestBody ProductInput productInput) {
        final var product = productsCommandUseCase.createProduct(
            productInput.toProduct(productId));
        HttpStatus status = product.isNewProduct() ? HttpStatus.CREATED :
            HttpStatus.OK;
        return ResponseEntity.status(status)
              .body(productMapper.toProductOutput(product.product()));
     }
```

In our case, we decided to use the `PUT` annotation; this decision was made because our method is idempotent, so we can use this method to update or create the product. One of the requirements is to not allow two products to have the same SKU; in our case, we can update or create the product if it does not exist.

We then have two annotations for the input, `@Valid` and `@RequestBody`. The `@Valid` annotation will be discussed in the *Bean Validation* section. `@RequestBody` means that we should send the input by passing it into the body of the request.

Following best practices, this method can return `200 (ok)` or `201 (created)`, depending on whether the product already exists or not.

All API call examples will be demonstrated using a `curl` command, but you can easily perform the same requests using various tools, such as **Postman** or **UseBruno**, directly from your IDE, or with any other API testing tool of your choice.

To call this endpoint, we can use a request as follows:

```
 curl -X 'PUT' \
  'http://localhost:8080/api/products/AK21101' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "name": "Keyboard",
  "description": "Ergonomic Keyboard",
  "price": 60
}'
```

We will receive the following output:

```
HTTP/1.1 201
Content-Type: application/json
{
  "name": "Keyboard",
  "sku": "AK21101",
  "description": "Ergonomic Keyboard",
  "price": 60.00
}
```

Get all products endpoint

To return all the products, we need to use the HTTP `GET` method:

```
@GetMapping
@Override
public ResponseEntity<List<ProductOutput>> getProducts() {
    final var products = productsQueryUseCase.getAllProducts()
          .stream()
          .map(productMapper::toProductOutput)
          .toList();
    return ResponseEntity.status(HttpStatus.OK)
                .body(products);
}
```

To call this endpoint, we can use a request as follows:

```
curl -X 'GET' \
  'http://localhost:8080/api/products' \
  -H 'accept: application/json'
```

We will receive the following output:

```
HTTP/1.1 200
Content-Type: application/json
[
  {
    "name": "Keyboard",
    "sku": "AK21101",
    "description": "Ergonomic Keyboard",
    "price": 60.00
  }
]
```

Get product by ID endpoint

To return the product by ID, we need to use the HTTP `GET` method and pass the ID as a path parameter:

```
@GetMapping(value = "/{productId}")
@Override
public ResponseEntity<ProductOutput> getProductById(@
PathVariable("productId") String productId) {
    final var product = productsQueryUseCase.getProductById(productId);
    return ResponseEntity.status(HttpStatus.OK)
            .body(productMapper.toProductOutput(product));
}
```

This endpoint uses the `@PathVariable("productId")` annotation and expects to have this value in the URI. We also need to add this path variable ID to the `@GetMapping` mapping annotation.

In case of success, we should return the HTTP status `200` code and return the product content. Otherwise, we can return `404`, meaning the product does not exist. We are going to see how to handle errors in the next section.

To call this endpoint, we can use the following request:

```
curl -X 'GET' \
 'http://localhost:8080/api/products/AK21101' \
 -H 'accept: application/json'
```

We will receive the following output:

```
HTTP/1.1 200
Content-Type: application/json
{
  "name": "Keyboard",
  "sku": "AK21101",
  "description": "Ergonomic Keyboard",
  "price": 60.00
}
```

Delete product by ID endpoint

Deleting a product by ID is very similar to getting a product by ID, with the big difference being the HTTP method; in this case, we should use the `DELETE` method:

```
@DeleteMapping(value = "/{productId}")
@Override
public ResponseEntity<Void> deleteProduct(@PathVariable("productId")
String productId) {
    productsCommandUseCase.deleteProduct(productId);
    return ResponseEntity.noContent().build();
}
```

This method is very similar to the previous one. The big difference is the `@DeleteMapping` annotation and `return`. In this case, we return a `204 (no content)` HTTP status. This means that we did not provide any content and the request was done successfully.

The `DELETE` method is idempotent; because of this, we can return `204` whether the product exists or not. As the expected behavior will be the same, the product will not exist in the database.

To call this endpoint, we can use the following request:

```
curl -X 'DELETE' \
 'http://localhost:8080/api/products/AK21101' \
 -H 'accept: */*'
```

We will receive the following output:

```
HTTP/1.1 204
```

Update product description by ID endpoint

The endpoint for updating the product description by ID is much like the one for retrieving a product. In this case, we will use the `PATCH` method, as it is a partial update, so the client will only pass a body with the content that is changed:

```
@PatchMapping(value = "/{productId}")
@Override
public ResponseEntity<ProductOutput> editProductDescription(@
PathVariable("productId") String productId,
@RequestBody @Valid ProductDescriptionInput input) {
    final var product = productsCommandUseCase.updateProductDescription(
        productId, input.description());
    return ResponseEntity.status(HttpStatus.OK)
            .body(productMapper.toProductOutput(product));
}
```

This method receives the product ID in `@PathVariable` and uses the `@PatchMapping` annotation to map it to the `PATCH` method. This method also has an input argument of type `ProductDescriptionInput` with the `@RequestBody` and `@Valid` annotations. The input argument represents the request body containing the description of the product that the client would like to change.

To call this endpoint, we can use the following request:

```
 curl -X 'PATCH' \
  'http://localhost:8080/api/products/AK21101' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "description": "Ergonomic Keyboard 2.0"
}'
```

We will receive the following output:

```
HTTP/1.1 200
Content-Type: application/json
{
  "name": "Keyboard",
  "sku": "AK21101",
  "description": "Ergonomic Keyboard 2.0",
  "price": 60.00
}
```

With our endpoints in place, our API class is now complete.

You can access the full code with the following link: `https://github.com/PacktPublishing/Mastering-RESTful-Web-Services-with-Java/blob/main/chapter2/product-api/src/main/java/com/packt/productapi/adapter/inbound/rest/ProductsApiController.java`.

The code of our API is done; now we can call the endpoints and test them. However, what happens if the product does not exist, or we send invalid data? We need to cover the negative cases.

Exception handling

Exception handling is the part of the implementation where we respond to errors or exceptional situations that occur during a program's execution.

In a RESTful API, many types of errors can occur, usually belonging to the `4xx` or `5xx` group of HTTP status codes. The `5xx` statuses mean that there is an error in the server application. They can be caused by external factors on the server side (e.g., unavailability of a database server) or a bug in the application (e.g., `NullPointerException`). On the other hand, the `4xx` statuses indicate errors caused by the client. For example, `405 (Method Not Allowed)` is automatically generated by the Spring Framework when the client tries to use an HTTP method not expected by the API.

Spring and Java provide many ways to handle exceptions. We can create a `try-catch` block in each endpoint and handle them one by one; we can also have a custom `ExceptionHandler` for each controller or have a global exception handler.

We will be implementing global exception handling in our API. This approach helps to prevent repeated code and ensures uniform behavior across all endpoints, especially if the API exposes multiple resources.

Spring Frameworks supports the *Problem Details for HTTP APIs* specification, RFC 9457. This RFC is an updated standard that builds upon RFC 7807. The main goal is to provide detailed information about errors in HTTP APIs by defining five fields:

- `Type`: This should contain a URI that identifies the problem type
- `Status`: This indicates the HTTP status code
- `Title`: The title is a concise summary of the problem
- `Detail`: This field should contain more detailed information about the problem
- `Instance`: This should contain a URI that identifies the endpoint call

In Spring, we can follow the RFC guidelines by extending the `ResponseEntityExceptionHandler` class or enabling the `spring.mvc.problemdetails.enabled` property. We will extend the `ResponseEntityExceptionHandler` class, which provides the `ProblemDetail` class with the five elements defined by RFC 9457. We can also add some extensions, which we will do to provide better response errors in `400 (Bad Request)`:

```
@ControllerAdvice
public class GlobalExceptionHandler extends ResponseEntityExceptionHandler
{
    @ExceptionHandler(EntityNotFoundException.class)
    public ResponseEntity<Object> handleNotFound(
        EntityNotFoundException exception, WebRequest request) {
        ProblemDetail problemDetail = ProblemDetail.forStatusAndDetail(
            HttpStatus.NOT_FOUND, exception.getMessage());
        return this.handleExceptionInternal(exception, problemDetail,
            new HttpHeaders(), HttpStatus.NOT_FOUND, request);
    }
    @ExceptionHandler(ConstraintViolationException.class)
    public ResponseEntity<Object> handleBadRequestDueToBeanValidation(
        ConstraintViolationException ex, WebRequest request) {
        final Set<String> errors = ex.getConstraintViolations().stream()
                .map(e -> ((PathImpl) e.getPropertyPath())
                          .getLeafNode() + ": " + e.getMessage())
                .collect(Collectors.toSet());
        ProblemDetail problemDetail = ProblemDetail.forStatusAndDetail(
            HttpStatus.BAD_REQUEST, "Invalid request content.");
        problemDetail.setProperty("errors", errors);
        return this.handleExceptionInternal(ex, problemDetail,
```

```
            new HttpHeaders(), HttpStatus.BAD_REQUEST, request);
    }
    @Override
    protected ResponseEntity<Object> handleMethodArgumentNotValid(
        MethodArgumentNotValidException ex, HttpHeaders headers,
        HttpStatusCode status, WebRequest request) {
        final List<String> errors = new ArrayList<>();
        for (final FieldError error :
            ex.getBindingResult().getFieldErrors()) {
                errors.add(
                    error.getField() + ": " + error.getDefaultMessage());
        }
        for (final ObjectError error :
            ex.getBindingResult().getGlobalErrors()) {
                errors.add(
                error.getObjectName() + ": " + error.getDefaultMessage());
        }
        ProblemDetail problemDetail = ex.getBody();
        problemDetail.setProperty("errors", errors);
        return handleExceptionInternal(ex, problemDetail, headers, status,
            request);
    }
}
```

Our `GlobalExceptionHandler` extends the `ResponseEntityExceptionHandler` Spring class, which provides the `handleMethodArgumentNotValid` method; we are overriding this method and updating the `ProblemDetail` class to give a clear message in the case of `400 (Bad Request)` informing about the wrong input data. This kind of error can be detected by Bean Validation when the user does not send the correct input.

Our class also has two custom `ExceptionHandler`: the first one to handle our custom exception, `EntityNotFoundException`, and the second one to handle `ConstraintViolationException`, which can be caused due to validation from `PathVariable`.

Thanks to our `GlobalExceptionHandler`, every the `EntityNotFoundException` exception is thrown, the client will receive a **404** error.

By creating this class, we have implemented global exception handling and defined a consistent and standardized format for errors, making it easier for clients consuming the API to understand and handle errors consistently across different endpoints. Besides, it also provides more clarity, helping clients to understand the nature of the error and how to resolve it. This improves the developer experience and reduces the time required to diagnose and fix issues.

Now that we have our exception handler, we need to ensure that the user does not send invalid input. Let's see how we can achieve this using Bean Validation.

Bean Validation

Bean Validation is a Java specification based on **Java Specification Requests (JSRs)** 380, 349, and 303. These requests provide a standardized approach to validating Java beans. Bean Validation is very useful for validating user requests, by providing some predefined rules and a way to create your custom Bean Validation, making it easier to enforce data integrity and validate input data within Java applications.

Bean Validation can be applied through various annotations, as follows:

- `@NotNull`: Specifies that the annotated element must not be null
- `@Size`: Specifies the size constraints for a string, collection, map, or array
- `@Min`: Specifies the minimum value for a numeric element
- `@Max`: Specifies the maximum value for a numeric element
- `@DecimalMin`: Specifies the minimum value for a numeric element with decimal precision
- `@DecimalMax`: Specifies the maximum value for a numeric element with decimal precision
- `@Digits`: Specifies the exact or maximum number of digits for a numeric element
- `@Pattern`: Specifies a regular expression pattern that the annotated element must match
- `@Email`: Specifies that the annotated element must be a well-formed email address
- `@NotBlank`: Specifies that the annotated string must not be null or empty and must contain at least one non-whitespace character
- `@NotEmpty`: Specifies that the annotated string must not be null or empty
- `@Positive`: Specifies that the annotated numeric element must be positive (greater than 0)
- `@Negative`: Specifies that the annotated numeric element must be negative (less than 0)
- `@PositiveOrZero`: Specifies that the annotated numeric element must be positive or 0
- `@NegativeOrZero`: Specifies that the annotated numeric element must be negative or 0
- `@Past`: Specifies that the annotated date or time must be in the past

- `@PastOrPresent`: Specifies that the annotated date or time must be in the past or present
- `@Future`: Specifies that the annotated date or time must be in the future
- `@FutureOrPresent`: Specifies that the annotated date or time must be in the future or present

Besides all this, the Bean Validation specification provides the `@Valid` annotation, which is used to trigger the validation of nested objects or elements within a collection. When applied to a field or parameter representing a complex object or a collection of objects, `@Valid` tells the Bean Validation framework to validate the nested objects according to their validation constraints.

Let's implement a custom Bean Validation for our SKU field and apply the annotations we've discussed in our request.

Our API has an SKU field that should conform to a pattern (AA99999) specified in the requirements phase. We can prevent clients from using invalid SKUs by creating a custom Bean Validation rule.

We can use the following code:

```
@Constraint(validatedBy = {})
@NotNull
@Pattern(regexp = "[A-Za-z]{2}[0-9]{5}",
    message = "SKU must follow the pattern AA99999")
@Target({PARAMETER, FIELD})
@Retention(RetentionPolicy.RUNTIME)
public @interface ValidSku {
    String message() default "Invalid SKU";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}
```

This code creates a new annotation called `@ValidSku`, and it checks whether the field adheres to a specific pattern: two alphabetic characters followed by five numeric digits (e.g., `AA12345`). If the annotated field does not match the defined pattern, a validation error message, `SKU must follow the pattern AA99999`, is generated.

The `ValidSku` annotation is designed to be applied to fields and parameters, as indicated by the `@Target` annotation. It also indicates that the validation constraint should be retained at runtime for reflection purposes. This annotation streamlines the process of validating SKUs in Java applications, enhancing data integrity and ensuring compliance with specific SKU formatting requirements.

`ValidSku` also validates whether the field is not null using the `@NotNull` annotation.

After creating our annotation, we can update our API by adding `@ValidSku` to our `PathVariable`. We should also add the `@Validated` annotation from Spring to our `ProductsApiController`. It is essential to validate the method annotation and not just the fields:

```
public interface ProductsApi {
    ResponseEntity<ProductOutput> createOrUpdateProduct(
        @ValidSku String productId, @Valid ProductInput productInput);
    ResponseEntity<Void> deleteProduct(@ValidSku String productId);
    ResponseEntity<ProductOutput> editProductDescription(
        @ValidSku String productId, @Valid ProductDescriptionInput input);
    ResponseEntity<ProductOutput> getProductById(
        @ValidSku String productId);
    ResponseEntity<List<ProductOutput>> getProducts();
}
```

Now that we have created the SKU validation and our `ExceptionHandler`, we can test it by passing a wrong SKU to check the result:

```
 curl -X 'PUT' \
  'http://localhost:8080/api/products/AAAA' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "name": "Keyboard",
  "description": "Ergonomic Keyboard",
  "price": 60
}'
```

We will receive the following output:

```
HTTP/1.1 400
Content-Type: application/json
{
  "type": "about:blank",
  "title": "Bad Request",
  "status": 400,
  "detail": "Invalid request content.",
  "instance": "/api/products/123",
```

```
  "errors": [
    "productId: SKU must follow the pattern AA99999"
  ]
}
```

Our output follows the standard by RFC 9457, and we also have an extra field to detail the errors from the Bean Validation.

Now, we can enhance our `ProductInput` and `ProductDescriptionInput` classes with Bean Validation annotations to ensure the input is valid. For the `ProductInput` class, we can use the following annotations:

```
public record ProductInput(
    @NotBlank
    @Size(min = 3, max = 255)
    @JsonProperty("name")
    String name,
    @NotBlank
    @Size(min = 10, max = 255)
    @JsonProperty("description")
    String description,
    @NotNull
    @Positive
    @JsonProperty("price")
    BigDecimal price) {
}
```

By adding these annotations, we guarantee that the input is in accord with what is expected from our API. With these annotations, we provide a safer API, as we control the user input and ensure the data integrity.

Our code guarantees these behaviors:

- The name should not be blank, and its length should be between 3 and 255
- The description should not be blank, and its length should be between 10 and 255
- The price should not be null and should be a positive number

In case the user of this API does not obey these rules, the client will receive a `400` error with the message detail that we defined by the Bean Validation.

Bean Validation lets our API provide standardized feedback to clients when validation errors occur, which is crucial for maintaining data integrity. It ensures that invalid or malformed data is prevented from being processed, significantly reducing the risk of data corruption or inconsistent state within the application.

Summary

In this chapter, we covered many topics essential to creating a RESTful API. You learned how to design and implement a RESTful API following the best practices using the HTTP methods and HTTP status codes, and how to apply Bean Validation to guarantee data integrity. Not only that, you've also learned how to handle exceptions using the Spring Framework.

In the next chapter, we will see how to create well-defined documentation for our API that clients can use.

Further reading

- *Jakarta Bean Validation specification* (version 3.0): `https://jakarta.ee/specifications/bean-validation/3.0/jakarta-bean-validation-spec-3.0.html`
- *RFC 9110*: `https://www.rfc-editor.org/rfc/rfc9110.html`
- *RFC 9457*: `https://www.rfc-editor.org/rfc/rfc9457.html`

3

Documenting Your API Effectively

Comprehensive **documentation** is essential for ensuring clients understand how to use your API effectively. In this chapter, we will explore the process of documenting a REST API using **Swagger** annotations and best practices. We'll delve into several key topics to provide you with a comprehensive understanding.

Firstly, we'll discuss the importance of API specifications by examining the **OpenAPI Specification** and **JSON Schema**, essential standards that define the structure and format of API documentation.

We'll also explore the debate between **code-first** and **specification-first** approaches in API development, discussing their respective advantages and considerations. Following this, we'll document the Product API, illustrating how to use Swagger annotations effectively to describe endpoints, parameters, responses, and other critical details.

Finally, we'll demonstrate the practical utility of the Swagger **user interface (UI)**, a powerful tool for visualizing and interacting with API documentation, enhancing developer experience and facilitating seamless API consumption. Through these discussions and examples, you will gain insights into creating well-documented, standardized REST APIs that promote interoperability and ease of use.

In this chapter, we'll be covering the following topics:

- Importance of API specifications
- Introducing OpenAPI and JSON Schema
- Choosing between specification-first and code-first

- Documenting the Product API
- Using the Swagger UI

Technical requirements

In this chapter, we will implement the documentation for our Product API. To be able to follow along and use the code examples as they are printed in this book, you should have the Product API code that was created in the previous chapter. You can access the code for this chapter on GitHub at `https://github.com/PacktPublishing/Mastering-RESTful-Web-Services-with-Java/tree/main/chapter3`. The code added in this chapter will not change the actual functionality of the API; it will only provide metadata to generate the documentation for the API.

Importance of API specifications

Let us begin by discussing what API specifications are and why they are important.

From Java, we know the principles of object-oriented programming. One of those principles is **encapsulation**, which involves making as many members of a class private as possible. Only those members (typically methods) that are explicitly intended for external access should be made public. The collection of these public members forms the class's API. It is good practice to accompany the public members by Javadoc so that their usage is clear without seeing the code implementing the class. By limiting the size of the public API, we retain the liberty to change or remove the internal (private) members without breaking code outside the class.

We should also be cautious when defining the signatures of public methods. Passing unnecessary data via arguments or return values can cause unnecessary coupling, increase the complexity of the code processing the data (e.g., validations or ensuring immutability), and limit the possibilities of performance optimization.

A poorly designed API, whose structure is driven more by the ease of technical implementation rather than business requirements, can expose elements that should be hidden. Once such an API is in use, it becomes difficult to change because there may be clients outside our control who depend on it.

If this is true for APIs within a single Java program, it is even more crucial for REST APIs that cross the boundaries of individual programs to have clearly separated interfaces and thorough documentation.

Although it may be necessary to provide API users with some additional information using prose, we can greatly benefit from describing as much as possible of our API using a formal and machine-readable specification language. For REST APIs, the most widely used and advanced standard is **OpenAPI**.

Introducing OpenAPI and JSON Schema

The **OpenAPI Specification**, backed by the OpenAPI Initiative (`https://www.openapis.org/`), defines a formal language that both humans and computers can read to learn about the capabilities of a service without the following:

- Access to the service implementation code
- Separate documentation in another format (language)
- Analyzing the network traffic

The OpenAPI Specification was originally based on the Swagger Specification. OpenAPI can describe any API using the HTTP protocol, including RESTful APIs at any maturity level.

OpenAPI Specification documents can be written in JSON and YAML formats. The YAML syntax (`https://yaml.org/`) replaces a lot of the JSON punctuation with indentation; therefore, YAML tends to be more concise and easier to read for many humans. In this book, we will use the YAML format for all OpenAPI Specifications written manually (not generated by tools).

The two most important (and usually longest) parts of a typical OpenAPI Specification are as follows:

- **Paths**: These describe the actions that the API supports, specified by resources, HTTP methods, URI parameters, etc.
- **Schemas**: These describe the complex data structures of the request and response bodies (payloads)

Our Product API uses two paths:

- `/api/products/{productId}`: This supports the `GET`, `PUT`, `DELETE`, and `PATCH` methods
- `/api/products`: This supports the `GET` method only

The following schemas are used:

- `ProductInput`: This is used for full product data in requests
- `ProductDescriptionInput`: This is used to update the product description only

- `ProductOutput`: This is used for product data in responses
- `ProblemDetail`: This is used for error responses

The format we use to transfer complex data through our API is JSON, which is why the schemas part of the OpenAPI Specification uses a dialect of a specification language called JSON Schema.

JSON Schema is a separate standard that can be used to validate JSON documents outside the scope of OpenAPI Specifications. The schema part of every version of the OpenAPI Specification is based on a version of the JSON Schema standard, plus some OpenAPI-specific modifications. This is why it is called a **dialect** of the JSON Schema. It is important to use the features of the schemas section supported by the version of the OpenAPI Specification that we want to use.

Let's see what the OpenAPI Specification of our Product API would look like. We will use YAML syntax. In the following subsections, some attributes of the API specification are omitted for brevity. You can see the full specification in the accompanying GitHub repository at `https://github.com/PacktPublishing/Mastering-RESTful-Web-Services-with-Java/blob/main/chapter3/product-api/src/main/resources/Product_Catalogue_API.yml`.

Common API metadata

Besides the two main parts of the specification mentioned earlier, there are header sections providing metadata about the API and its specification. They define the version of the OpenAPI standard, the version of our API, the base URL for the API, and some human-readable names and descriptions:

```
openapi: 3.0.1
info:
  title: Product Catalogue API
  description: API for managing product catalog
  version: 0.0.1
servers:
  - url: http://localhost:8080
    description: Generated server url
tags:
  - name: products
    description: the products API
```

Product API paths

The following parts of the Product API specification are the paths, starting with the `GET` method for `/api/products/{productId}`. The `$ref` notation is used to refer to the schemas. There are separate response specifications for the `200`, `400`, and `404` HTTP response codes. We can also see the specification of the required `productId` URI parameter:

```
paths:
  /api/products/{productId}:
    get:
      parameters:
        - name: productId
          in: path
          required: true
          schema:
            type: string
          example: AK21109
      responses:
        '200':
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/ProductOutput'
        '400':
          content:
            '*/*':
              schema:
                $ref: '#/components/schemas/ProblemDetail'
        '404':c
          content:
            '*/*':
              schema:
                $ref: '#/components/schemas/ProblemDetail'
```

The same path continues with the specification of the `PUT` method that needs a request body:

```
    put:
      parameters:
        - name: productId
          in: path
          required: true
          schema:
            type: string
      requestBody:
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/ProductInput'
        required: true
      responses:
        '200':
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/ProductOutput'
        '201':
```

(The response bodies for the other status codes are omitted.)

Let's look at one more operation: the one returning the list of products (represented by a JSON array):

```
  /api/products:
    get:
      tags:
        - products
      summary: Retrieve all products
      operationId: getProducts
      responses:
        '200':
          description: A list of products
          content:
            application/json:
```

```
              schema:
                type: array
                items:
                  $ref:
    '#/components/schemas/ProductOutput'
```

Product API schemas

Putting the named schemas in a separate part of the specification lets us reuse them for multiple operations, for example, the `ProductOutput` schema.

Using the `example` attribute, we can provide additional information about the data elements that can be used by both humans and automated tools (test client or mock server generators). Without the `example` attribute, anyone who wants to test our API would have to guess meaningful values solely based on their types. Generating random strings would lead to unrealistic examples:

```
components:
  schemas:
    ProductOutput:
      type: object
      properties:
        name:
          type: string
          example: Keyboard
        productId:
          type: string
          description: ID of the product to delete
          example: AK21109
        description:
          type: string
          example: Ergonomic Keyboard
        price:
          type: number
          example: 60
```

The `ProblemDetail` schema used for error responses is so generic that it is likely that we will want to reuse it across multiple APIs (microservices). OpenAPI Specifications can be divided into multiple files. We can also host the reusable schemas on a web server and refer to them using HTTP URLs.

However, we should bear in mind that it brings coupling among the different APIs, so this kind of reuse should only be used with schemas that rarely change (the `ProblemDetail` schema fulfills this criterion).

We have introduced the basic structure and the most important attributes of OpenAPI Specifications. We will get to more advanced OpenAPI features in the following chapter.

In the following section, you will learn about an important decision you have to make: This is whether you want to start with an explicit abstract specification document, such as the one shown earlier, or derive it from Java code.

Choosing between specification-first and code-first

We made it clear that for the success of our API, we need both a concrete implementation and an abstract specification of the interface it implements. There are two basic approaches when creating APIs: **specification-first** (also known as *design-first* or *API-first*) and **code-first**.

Specification-first

Starting with the specification forces the API developers to decide upfront what needs to be in the public part of the API. Intentional hiding of the details the consumers do not need to know makes the API as small as possible and hence easier to maintain and evolve.

Here are the pros of specification-first:

- Standalone specifications tend to be smaller as they only define the elements of the API needed to satisfy the business requirements; the APIs defined by separate specification documents are easier to maintain, and the specification changes are more controllable.
- Specification-first ensures the API specification is not biased to any implementation language, supporting API providers and consumers written in multiple languages.
- You get the ability to develop the API provider and consumers in parallel. The interface between the connected parties can be negotiated without going into the details of the implementation.
- Tests can be created before the API is implemented.
- Specifications written manually (not generated) are easier for humans to read.

Here are the cons of specification-first:

- The development team needs to master the API specification language in addition to the implementation programming language.

- To maintain consistency between the specification and the code, we need tools to generate code stubs from the specification. The code generators available may not support all the features of both the specification and the implementation language.

Code-first

Code-first prioritizes the short-term speed of implementation over clean interface design. The API specification is actually reverse-engineered from the implementation code. Annotations on code elements and additional metadata are added manually to aid the tool used to generate the specification.

Here are the pros of code-first:

- Skipping the specification step leads to a working API faster; this makes sense for rapid prototyping or low-impact APIs whose clients are under our control
- Both the implementation and the interface are written in one language (usually with the help of some annotations or metadata)
- Having the implementation code available may help to include performance or other non-functional considerations in the API specification

Here are the cons of code-first:

- Need to explicitly prevent unwanted implementation details from leaking in the generated specification.
- Need to master the specification generation tools with their annotations specific to each implementation language instead of using a standard implementation-agnostic specification language.
- Hard to generate polished and nice-to-read specifications, especially for more complex APIs.
- The structure of the APIs developed without the conscious API design step is likely to reflect the (first) technical implementation rather than the business domain. This is why they tend to be hard to maintain and evolve.

In this chapter, we will demonstrate the code-first approach for the Product API because we already have its implementation code from the previous chapter.

The specification-first approach will be shown in the following chapter.

Documenting the Product API

To document our API, we will need to add some dependencies from Swagger. In the case of the Spring framework, we have the `springdoc-openapi-starter-webmvc-ui` dependency, which covers all we need to document the API of the Spring application:

```
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
    <version>2.5.0</version>
</dependency>
```

Adding this dependency in `pom.xml` allows us to use the Swagger annotations and the Swagger UI. After accessing `http://localhost:8080/swagger-ui/index.html`, we can see the Swagger UI, represented in *Figure 3.1*.

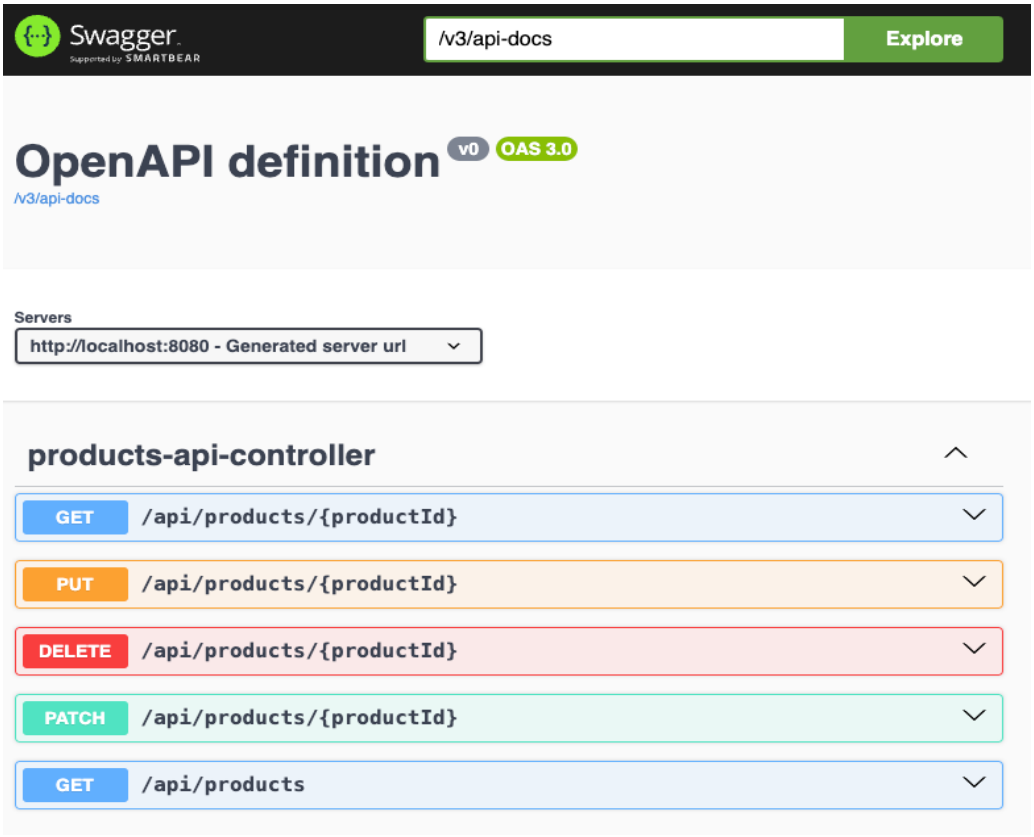


Figure 3.1 – Swagger UI

The Swagger UI is generated based on `RestController` and our endpoints. To make our API more user-friendly and comprehensible for clients, we should enhance it using Swagger annotations. These annotations are available in the `io.swagger.v3.oas.annotations` package. Let us explore some of them and see how they can be applied to improve our API documentation.

Swagger annotations

Swagger annotations are a set of annotations provided by the Swagger library used to generate interactive API documentation for RESTful web services. These annotations, when added to your code, help define and describe the structure and behavior of your API endpoints, request and response models, and overall API metadata.

@Tag

The `@Tag` annotation is used to group a set of operations; we can use it in our API class to inform that every method in this class should belong to this tag:

```
@Tag(name = "products", description = "the products API")
public interface ProductsApi {
....
}
```

@Operation

The `@Operation` annotation is used in method declarations to name API endpoints. This annotation can be combined with other annotations to provide a comprehensive description of the endpoint, including possible outputs and HTTP status codes:

```
@Operation(
    operationId = "deleteProduct",
    summary = "Logical remove a product by ID",
    responses = {
        @ApiResponse(responseCode = "204",
            description = "Product removed successfully"),
    }
)
ResponseEntity<Void> deleteProduct(
    @Parameter(name = "productId",
        description = "ID of the product to delete", required = true,
        in = ParameterIn.PATH, example = "AK21109") @ValidSku
        String productId
);
```

In the preceding code, we are using the `@Operation` annotation to define the description for our `delete` endpoint. The `@ApiResponse` annotation is used to add the possible response status, in this case, `204`, which represents that the product was removed successfully.

@Parameter

In our previous example, we used the `@Parameter` annotation. This annotation is specifically designed to be applied to method parameters, allowing for the definition of clear descriptions, examples, and additional information about each parameter. Such details are very useful for API consumers, as they provide clear examples and essential information, enhancing the usability of the API.

@ApiResponse

In our previous example, we applied the `@ApiResponse` annotation. This annotation serves to specify the potential responses from our API endpoint, detailing the HTTP status and response type, and providing examples where applicable. In our previous case, there was no return body, as it is `204`; let's see an example for a `GET` operation:

```
@Operation(
    operationId = "getProductById",
    summary = "Retrieve a product by ID",
    responses = {
        @ApiResponse(responseCode = "200", description = "Product found",
            content = {
                @Content(mediaType = "application/json",
                schema = @Schema(implementation = ProductOutput.class))}),
        @ApiResponse(responseCode = "404",
            description = "Product not found",content = @Content(
                schema = @Schema(implementation = ProblemDetail.class),
                examples = {
                    @ExampleObject(name = "Validation Error",
                        summary = "Example of validation error",
                        value = """
                        {"type":"about:blank",
                         "title": "Not Found",
                         "status": 404,
                         "detail": "Product not found with id AK21102",
```

```
                        "instance": "/api/products/AK21102"
                        }
""")})})})
ResponseEntity<ProductOutput> getProductById(
    @Parameter(name = "productId",
        description = "ID of the product to delete", required = true,
        in = ParameterIn.PATH, example = "AK21109")
    @ValidSku String productId
);
```

In our example, we have two instances of `@ApiResponse`, the first for successful responses with a status code of `200`. In this scenario, we use the `@Schema` annotation to reference the implementation class, specifically `ProductOutput.class`.

The second instance pertains to a `404` error, utilizing a schema outside our domain, namely `ProblemDetail.class` from Spring. In this case, we use another annotation from Swagger, the `@ExampleObject` annotation, to create a structured JSON representation of the expected error response, using Java's text block feature to enhance readability.

@Schema

In our previous example, we used the `@Schema` annotation to identify the content for the API response. This annotation is also used to describe the fields of the request and response models. Let's look at an example using `ProductInput`:

```
public record ProductInput(
        @NotBlank
        @Size(min = 3, max = 255)
        @Schema(name = "name",
            requiredMode = Schema.RequiredMode.REQUIRED,
            example = "Keyboard")
        @JsonProperty("name")
        String name,
        @NotBlank
        @Size(min = 10, max = 255)
        @Schema(name = "description",
            requiredMode = Schema.RequiredMode.REQUIRED,
            example = "Ergonomic Keyboard")
```

```
        @JsonProperty("description")
        String description,
        @NotNull
        @Positive
        @Schema(name = "price",
            requiredMode = Schema.RequiredMode.REQUIRED,
            example = "60.0")
        @JsonProperty("price")
        BigDecimal price) {
}
```

In this example, we are using `@Schema` to define the name of each field, indicate whether the field is required, and provide an example value. The example value is especially useful for new API users, as it offers clear information about the expected input and output.

By incorporating bean validation, we enhance our API with additional information about the fields. The generated code for `ProductInput` will look something like this:

```
    "ProductInput": {
        "required": [
            "description",
            "name",
            "price"
        ],
        "type": "object",
        "properties": {
            "name": {
            "maxLength": 255,
            "minLength": 3,
            "type": "string",
            "example": "Keyboard"
            },
            "description": {
                "maxLength": 255,
                "minLength": 10,
                "type": "string",
                "example": "Ergonomic Keyboard"
            },
            "price": {
                "type": "number",
```

```
                "example": 60
            }
        }
    }
```

By applying these annotations throughout our entire API, we achieve clear and well-defined documentation. Now, we can revisit the Swagger UI to review and explore the updated documentation.

Note

Many mock servers use the Swagger implementation to generate mocks; having a clear definition and examples for the input and output is essential to generate better mocks.

Using the Swagger UI

The **Swagger UI** is a web-based interface that provides interactive documentation for APIs. It allows developers to visualize and execute API endpoints, making it easier to understand and experiment with the API.

Before accessing the Swagger UI, let's add one more annotation from Swagger: the `OpenAPI` annotation. This contains some metadata from our API. We can do that in Spring by defining the `OpenAPI` bean:

```
@Configuration
public class SpringDocConfiguration {
  @Bean
  OpenAPI apiInfo(
    @Value("${application.version}") String version) {

    return new OpenAPI().info(
      new Info()
        .title("Product Catalogue API")
        .description("API for managing product catalog")
        .version(version));
  }
}
```

In this code, we use the `OpenAPI` bean from Swagger to define metadata information. Specifically, we set the title, description, and version of the API. The version information is retrieved from the application properties, allowing us to synchronize it with the version specified in `pom.xml`.

This annotation supports the following metadata items:

- `openapi`: The version of the OpenAPI specification.
- `info`: This provides metadata about the API:
- `title`: The title of the API
- `version`: The version of the API
- `description`: A brief description of the API
- `termsOfService`: A URL to the terms of service for the API
- `contact`: Contact information for the API
- `license`: License information for the API
- `servers`: This specifies the servers where the API can be accessed:
- `url`: The URL of the server
- `description`: A description of the server
- `paths`: The available endpoints for the API; this is usually generated from the annotations shown previously.
- `components`: Components are reusable schemas or other objects that can be referenced from multiple places in the API specification. This is also generated based on the annotations shown previously.
- `security`: This defines the security mechanisms for the API. It is also generated from the `@SecurityRequirement` annotations.
- `tags`: This provides a list of tags used by the specification with additional metadata. It is also generated from the `@Tag` annotation.
- `externalDocs`: This provides external documentation about the API:
- `description`: A short description of the external documentation
- `url`: The URL for the external documentation

All the preceding annotations and configuration parameters can be used to enrich the API documentation. Now that our API is documented in the Java code, we can view the results by accessing the Swagger UI at `http://localhost:8080/swagger-ui/index.html`, as shown in *Figure 3.2*:

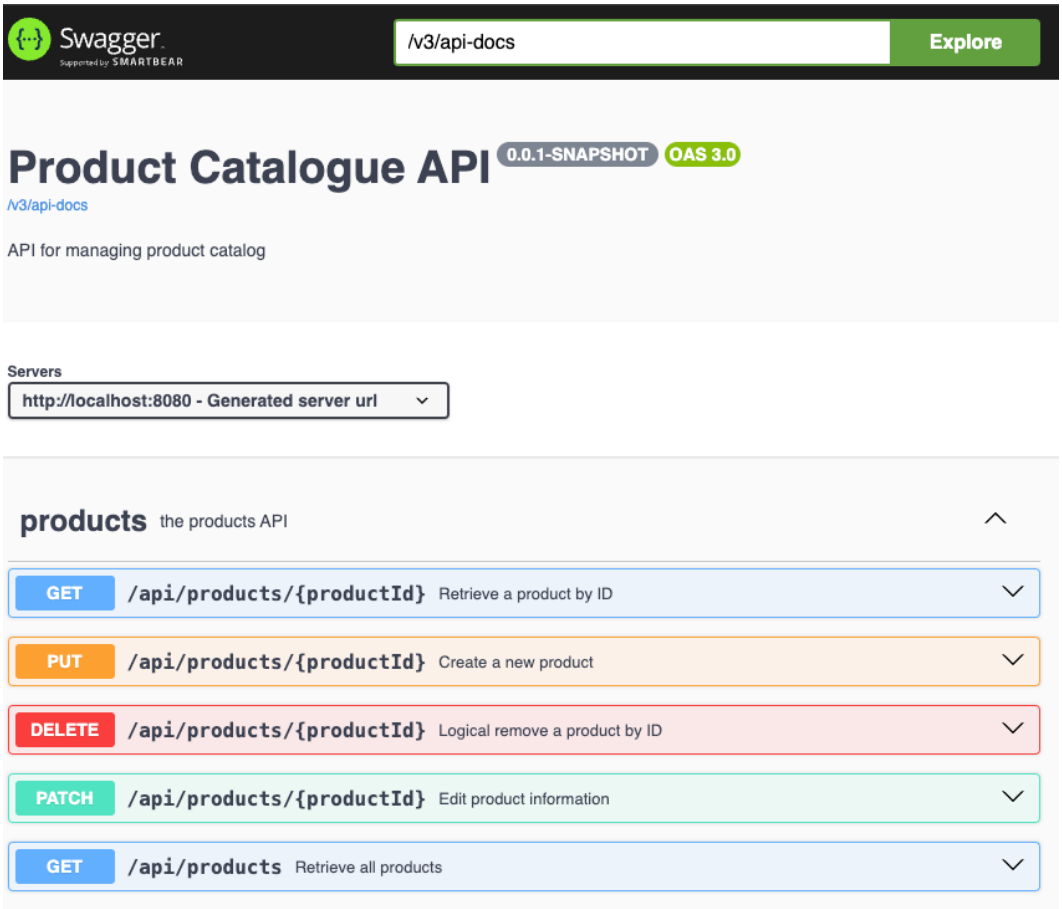


Figure 3.2 – Swagger UI

As we can see in the preceding figure, our API has more detail. It has the description of each endpoint and all the metadata provided in the `OpenAPI` bean.

In addition, we have schemas that provide detailed information about our data models. These schemas include details about each field, indicating whether the field is required, along with example values, as we can see in *Figure 3.3*.

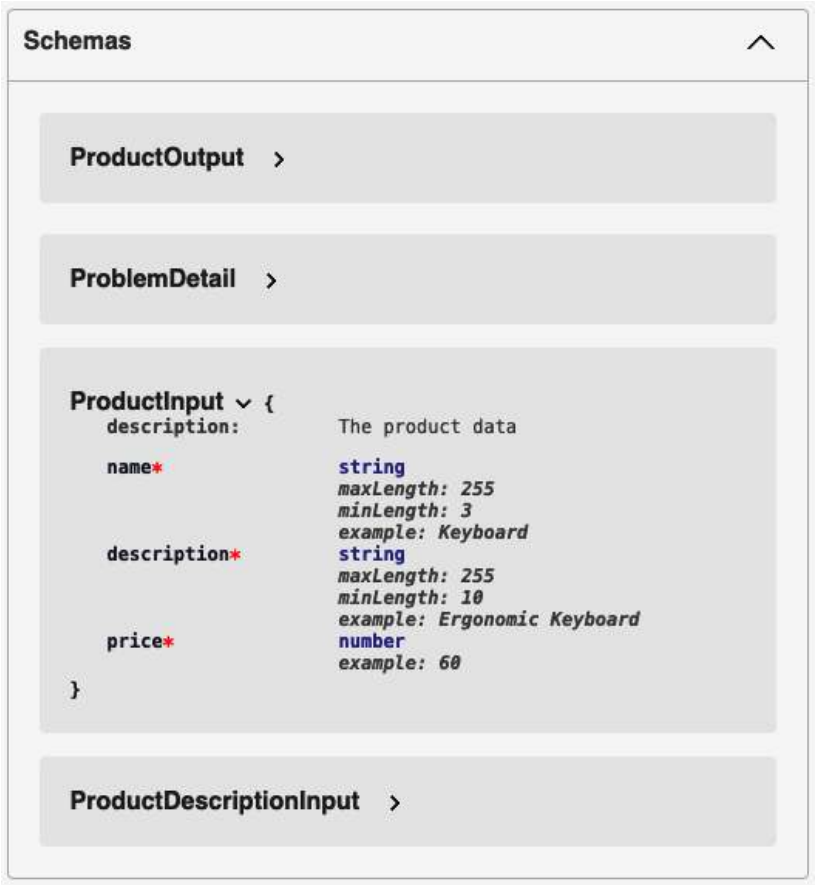


Figure 3.3 – Swagger schemas

With this information, API consumers can gain a good understanding of the API, simplifying the process of creating a client. In *Chapter 4*, we will explore how to use the generated documentation to help in generating the HTTP client.

Generating the documentation file is straightforward. In Swagger UI, there is a link that opens the OpenAPI Specification in JSON format, as shown in *Figure 3.4*:

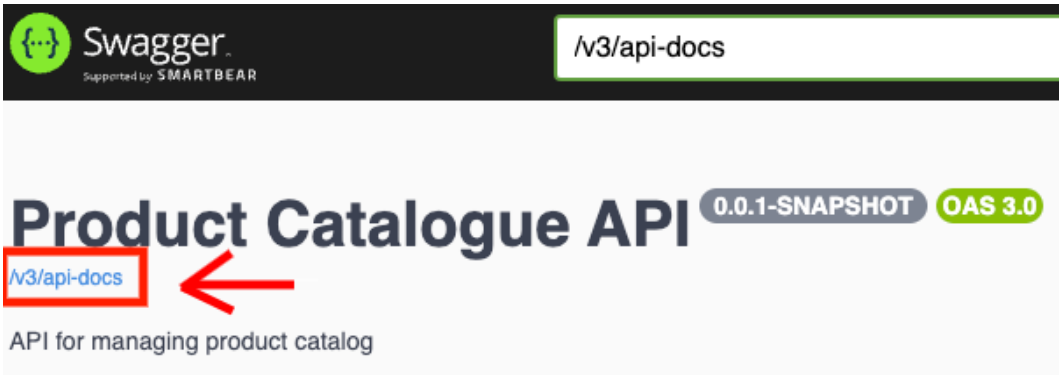


Figure 3.4 – Link to the documentation file

This specification file can be used by anyone who wants to integrate with our API. Many tools, such as Postman, IntelliJ IDEA, and others, seamlessly integrate with the OpenAPI standard, offering numerous benefits.

The wide range of uses of the OpenAPI Specification is the reason why we should prefer to share a good OpenAPI Specification rather than ready-made Java clients in the form of libraries (JAR files). Even if we would like to limit ourselves to only Java-implemented clients, ready-made libraries can cause problems:

- Incompatibility of used third-party libraries or their versions
- Code style not fitting with the rest of the client application (e.g., reactive versus imperative programming)
- Missing or incompatible annotations

The Swagger UI also offers a convenient way to interact with the API and view the results. This feature proves especially useful, as demonstrated in *Figure 3.5*, by allowing developers to easily test endpoints and observe responses directly within the interface.

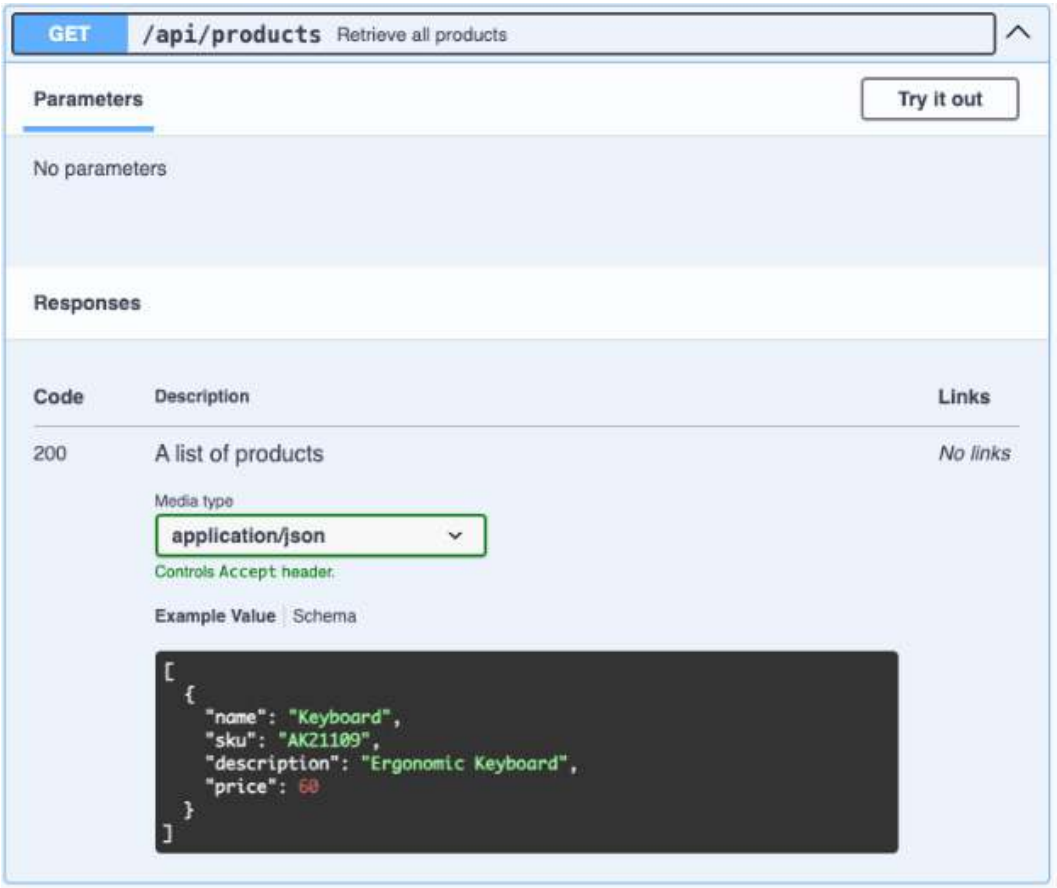


Figure 3.5 – Try it out – Swagger UI

In the preceding figure, we can see a **Try it out** button, which allows users to interact with the API directly from the documentation. This feature enables developers to send real requests and view the responses in real time, as illustrated in *Figure 3.6*. It's a straightforward way to validate API behavior without needing to write external scripts or tools.

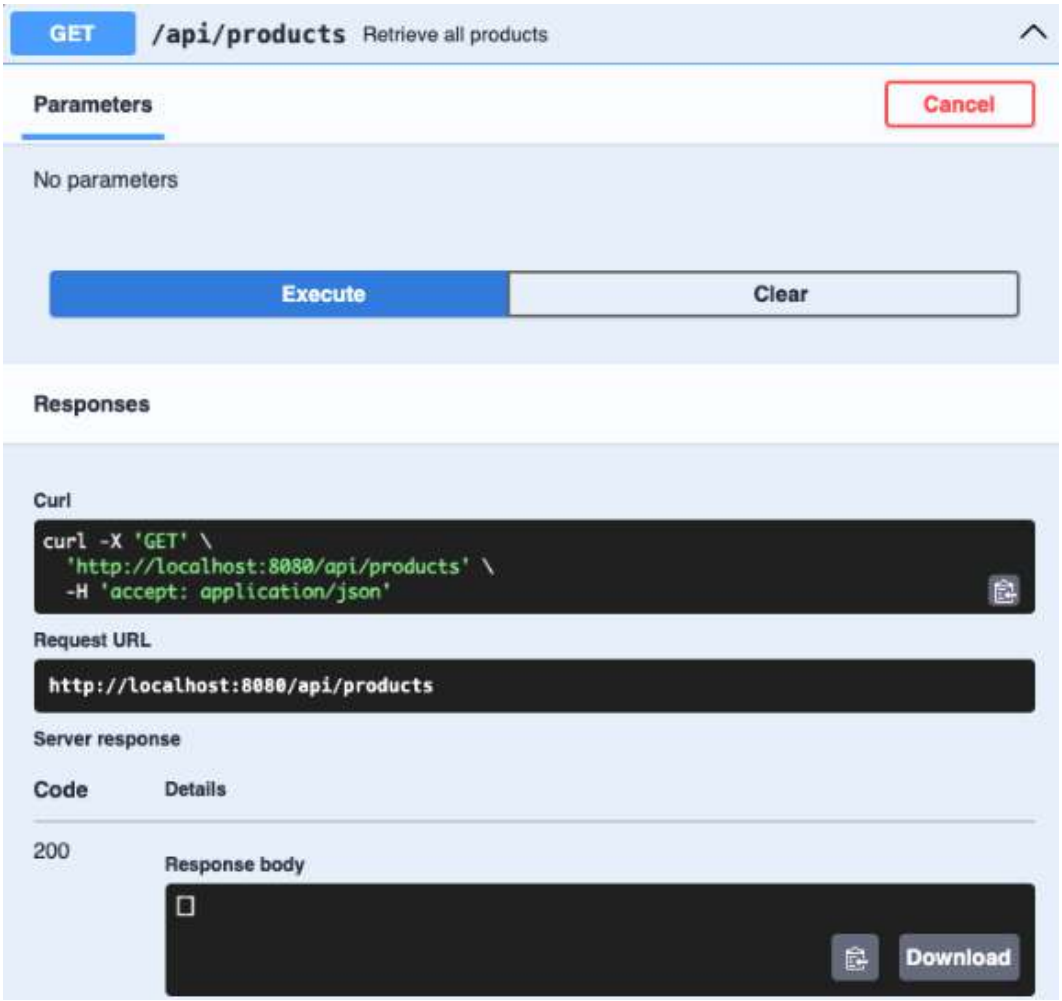


Figure 3.6 – Execute the Swagger UI

These features offered by the Swagger UI greatly enhance the developer experience by making it easier to test, understand, and integrate with the API. The interactive documentation allows developers to explore the API's functionality, ensuring they can quickly grasp how it works and incorporate it effectively into their own applications.

Summary

In this chapter, you learned about the OpenAPI Specification and how to utilize Swagger annotations to generate documentation. In the upcoming chapter, we will explore using this documentation to develop a client for it and delve deeper into the specification-first approach.

4

Generating Code with OpenAPI

In this chapter, we will start the development of a completely new API called Order Management that will interact with the Product API developed in *Chapter 2* and evolved in *Chapter 3*.

In the previous chapter, the Product API was developed following the *code-first* approach, wherein the API was implemented before the documentation.

In this chapter, we will adopt a *specification-first* approach, also known as API-first, wherein we will have the specification of the whole API done first, and then have the API implemented.

Firstly, we will build the OpenAPI specification for the Order Management API, declaring the paths, methods, and request and response schemas (some of them using polymorphism) for it.

Then, we will use OpenAPI tools to generate Java code stubs from the API specification. This generated code speeds up the implementation process, reduces manual maintenance of boilerplate code, and, most importantly, makes it easier to maintain consistency between the API specification and the service implementation for future changes.

Next, we will go through the package structure of the service and the code implementation bullet points for the build of this API, delivering it in a ready-to-work state.

Finally, we will look into establishing communication between the Order Management and Product APIs to validate the products that will be registered in the orders created in this service, by calling an external API and responding accordingly to the received condition.

By the end of this chapter, you will have built all the knowledge you need to get started with the design, development, and integration of specification-first APIs to greatly empower you and speed up your design and development phases in an unbelievable manner.

In this chapter, we will be covering the following topics:

- Specification of the Order Management API
- Generating code from the specification
- Package structure of the Order Management API
- Implementing the Order Management API controller
- Communicating with the Product API

Technical requirements

In this chapter, we will implement an example Order Management API. To be able to follow along and use the code examples as they are printed in the book, you should have the following:

- Intermediate knowledge of the Java language and platform
- At least a basic knowledge of Spring Boot or a similar framework
- Java 21 and Maven 3.9.0 installed

In this chapter, we are going to apply API-first REST principles to create our API; you can access the code for this chapter on GitHub at `https://github.com/PacktPublishing/Mastering-RESTful-Web-Services-with-Java/tree/main/chapter4`.

Specifying the Order Management API

In this section, you will develop the specification of the operations and data structures of the Order Management API. The Order Management API, as part of the project we are building over the course of this book, will handle the orders of products managed by the Product API.

As explained in the previous chapter, *specification-first* means an abstract API specification is created before the actual implementation of the API. Instead of starting with executable code in a programming language, you begin by defining the structure, behavior, and functionality of the API. That is why this is also known as API-first development.

In *Chapter 1*, we visited many key principles that should be considered when designing RESTful APIs. Let us look at how to apply those principles with the API-first approach.

Implementing HTTP principles in API-first development

When adopting an API-first approach, it is crucial to think in HTTP terms right from the design phase. Let us look at some of the principles:

- **Define API contracts first**: Use tools such as OpenAPI/Swagger to define your API structure, including endpoints, request/response formats, and data models. This contract serves as a blueprint for developers and helps ensure consistency and clarity. We are going to build an API contract with an OpenAPI specification file for the Order Management API in the next section.
- **Consistent use of HTTP methods and status codes**: Follow conventions for HTTP methods and status codes, as seen in *Chapter 2*. For example, use `GET` for retrieving resources, `POST` for creating, `PUT` for updating, and `DELETE` for removing resources. Use appropriate status codes to communicate the result of the request. These should also be described in the specification file.
- **Emphasize resource modeling**: Focus on accurately modeling your domain as resources. This involves identifying key entities and their relationships, identifying them by URLs that can be accessed using HTTP methods. For example, in our Order Management API, resources are `Customers`, `Products`, and `Orders`.
- **Plan for versioning**: Implement versioning strategies to handle API evolution. This can be done through URL path versioning (e.g., `/v2/books`) and standard or custom request headers. Versioning supports backward compatibility and smooth transitions as the API grows and changes. This will be defined in the Common API metadata of our specification file.
- **Consider security**: Define authentication and authorization mechanisms, such as OAuth, API keys, or JSON Web Tokens (JWTs), to secure your API. We will briefly look at how to add a security scheme to our specification, and you will learn how to implement it in depth in *Chapter 7*.

Now that we have a clearer picture of the principles we aim to achieve with this definition, let us start writing into the specification itself.

Designing the OpenAPI specification

We will write the Order Management API specification using the YAML syntax. You can use any editor of your personal preference to follow along: you can use the online Swagger Editor available at `https://editor.swagger.io/`, or if you work using IDEs such as JetBrains IntelliJ or Microsoft VS Code, you can use plugins to get a similar experience to the Swagger website and validate your specification while writing it inside your IDE.

The full specification that we will create is available on GitHub at `https://github.com/PacktPublishing/Mastering-RESTful-Web-Services-with-Java/tree/main/chapter4`.

Let us begin the specification file by defining the API's common metadata.

Common API metadata

We will start by defining the top-level context of the specification with the metadata about the API, defining the title, the description, and the version of the API. Also, here we describe the servers where the application will be running. In our case, since we want to run it locally, we will use `localhost` for testing purposes:

```
openapi: 3.0.0
info:
  title: Order Management API
  description: API for managing orders
  version: 1.0.0-SNAPSHOT
servers:
  - url: http://localhost:8080/api/v1
```

Order Management API paths

This API will have three paths:

- `/orders` with the `POST` and `GET` methods for creating and retrieving orders, respectively
- `/orders/{orderId}` with the `GET`, `PUT`, and `DELETE` methods for accessing, editing, and removing specific orders, respectively
- `/orders/{orderId}/status` with the `PATCH` method to change the status of an existing order

Methods for the /orders path

Let us start by defining the `POST` method with its request and response. Remember, the `$ref` notation is used to refer to the schema definitions in a separate section of the specification document. This allows the schemas to be reused, and the `paths` section to be shorter and more readable:

```
paths:
  /orders:
    post:
      tags:
        - OrderManagement
```

```
    summary: Create Order
    description: Creates a new order
    requestBody:
      required: true
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/OrderRequestBody'
    responses:
      '201':
        description: Created
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/OrderResponse'
      '400':
        description: Bad Request
```

In this `POST` definition, we are defining that when the controller receives a `POST` request with the `/orders` path, the request should contain a body as specified in the `OrderRequestBody` schema definition. If everything goes well, the API will respond with the `201 Created` HTTP status code, with the body defined in the `OrderResponse` schema. If the request body does not match `OrderRequestBody`, the API will respond with a `400 Bad Request` HTTP response code. We are going to define the request and response bodies later in this chapter.

Also, to help the OpenAPI generator name the classes that will be generated from our specification, we are specifying a tag in each method operation. In this case, the tag will be named `OrderManagement`. If we omit the tag, the generated class will be named `DefaultApi`. We will cover this in detail in the *Generating code from the specification* section when we have configured the generator and start generating code from the specification.

Next, we will define the `GET` method for the `/orders` path endpoint:

```
  get:
    tags:
      - OrderManagement
    summary: List Orders
    description: Retrieves a list of orders
    responses:
```

```
    '200':
      description: OK
      content:
        application/json:
          schema:
            type: array
            items:
              $ref: '#/components/schemas/OrderResponse'
```

This is a `GET` method and, when called, it returns the list of orders, if any, with HTTP status `200`, and with an array of objects of type `OrderResponse` that contains all the orders in the system. If there are none, the returned list will be empty still with HTTP status `200`, meaning that the request was successfully completed.

We will also specify an endpoint to retrieve a single order from the system in the next path.

Methods for the /orders/{orderId} path

Now, we are going to define another path to access specific orders, and we will define a `GET`, a `PUT`, and a `DELETE` method for this path, starting with the `GET` method:

```
/orders/{orderId}:
  get:
    tags:
      - OrderManagement
    summary: Get Order
    description: Retrieves a single order by its ID
    parameters:
      - name: orderId
        in: path
        required: true
        description: The ID of the order to retrieve
        schema:
         type: string
    example: 123456
    responses:
      '200':
        description: OK
        content:
          application/json:
```

```
            schema:
              $ref: '#/components/schemas/OrderResponse'
      '404':
        description: Not Found
```

In the `GET` method, we are defining a path to retrieve data from a single order. Here, we are receiving a single parameter in the path, to be able to get the data from a specific order, defined as the `/orders/{orderId}` path, containing the order ID in the `{orderId}` placeholder. We also need to have the parameter defined in `parameters`, under the `in: path` flag. This is where we specify that `orderId` is a parameter to be received.

In a successful case, an `OrderResponse` object will be returned containing the order itself with a `200 OK` HTTP status response. A `404 Not Found` HTTP status will be returned if there is no order with the given ID.

Now, let us have a look at the `PUT` method:

```
  put:
    tags:
      - OrderManagement
    summary: Update Order
    description: Updates an existing order
    parameters:
      - name: orderId
        in: path
        required: true
        description: The ID of the order to be updated
        schema:
          type: string
    example: 123456
    requestBody:
      required: true
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/OrderRequestBody'
    responses:
      '200':
        description: OK
```

```
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/OrderResponse'
      '400':
        description: Bad Request
      '404':
        description: Not Found
```

Similar to the `GET` method, with the `PUT` method, we are defining an operation to update a single order. Please note that we are demonstrating an approach different from that adopted when building the Product API, where we used the `PUT` method for creating entities. In the Order Management API, we use a separate `POST` method (without an ID in the request) for creating orders.

`PUT` is an idempotent method, meaning it can be called multiple times (with the same order ID) without affecting the result. On the other hand, `POST` is not idempotent, so every time you call it, it creates a new order in our service.

For the `PUT` method, we receive the order ID in the path, defined in the `/orders/{orderId}` path, receiving the order ID through the `orderId` parameter defined in the list of parameters and `OrderRequestBody`, the same way as seen in the `POST` method, containing all the details of the order to be updated.

In case of success, it will return a `200 OK` HTTP status code and `OrderResponse` containing the details of the persisted order. If the order is not found, `404 Not Found` will be returned. In case of a wrong format in the request, `400 Bad Request` will be returned.

Finally, let us see the `DELETE` method to cancel a specific order:

```
delete:
  tags:
    - OrderManagement
  summary: Cancel Order
  description: Cancels an existing order
  parameters:
    - name: orderId
      in: path
      description: ID of the order to be cancelled
      required: true
      schema:
```

```
        type: string
    example: 123456

responses:
  '204':
    description: No Content
  '403':
    description: Forbidden
  '404':
    description: Not Found
```

In this method, we are required to pass a parameter named `orderId`, which is the ID of the order to be cancelled, the same way we did with the `GET` and `PUT` methods.

In case of success, a response HTTP status code of `204 No Content` will be returned to the caller because there is no response body. `404 Not Found` will be returned if there is no order with the specified ID.

Method for the /orders/{orderId}/status

Finally, let us have a look at the last defined endpoint path to change the status of the orders, with the `PATCH` method:

```
/orders/{orderId}/status:
  patch:
    tags:
      - OrderManagement
    summary: Change Order Status
    description: Change an Order Status
    parameters:
      - name: orderId
        in: path
        required: true
        description: The ID of the order to be updated
        schema:
          type: string
    example: 123456
    requestBody:
      required: true
      content:
```

```
        application/json:
          schema:
            $ref: '#/components/schemas/OrderStatus'
    responses:
      '200':
        description: OK
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/OrderResponse'
      '404':
        description: Not Found
```

The goal of this endpoint is to update the status of an order. In a real-world production environment, the status of an order depends on external circumstances defined by other services in business logic, for example, the payment approval validated by a payment provider, availability of the items in a warehouse, or even fraud detection; it's important to have a way to change the status of the orders with external calls, and this endpoint is specifically to accomplish this.

Note that the body of this request is being defined by reference, using `$ref`, to the `OrderStatus` schema that will be defined in the next section. This exemplifies the reusability of the specification-first design, where you define a reusable schema that can be referenced across multiple path definitions, parameters, and endpoints in the OpenAPI schema definition file, eliminating the need to repeat any definitions. The next section covers this topic.

Now, we are going to define the schemas used as a body for the requests and responses and as params in the operations we have just finished defining.

Defining the API schemas

Now that we have finished defining the paths that identify the resources of our API, we are going to define the schemas that we use in the operations defined in the previous section. The schemas describe the structure of the request and response bodies for those operations and can also be used as parameters in the operations as seen before.

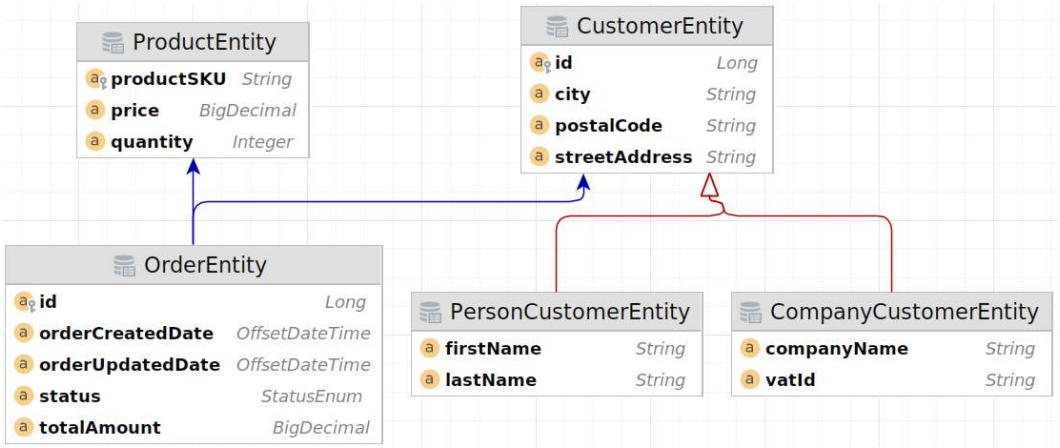


Figure 4.1: Entity-Relationship diagram from the defined API schema

In the preceding diagram, you can see the relationship between the entities that will be created later by extending from the schema definitions we are about to write and generate – that is why they differ in the name definition. This way, you can have the whole entity structure defined from the OpenAPI schema, reinforcing the API-first principle.

Schema definitions are located under the `components` section of the OpenAPI specification.

OpenAPI schemas use a language that supports defining complex structures of JSON data and the relationships between the structures. The schemas are entries inside the `schemas` section, with the first ones being `Product` and `Customer`:

```
components:
  schemas:
    Product:
      type: object
      properties:
        productSKU:
          type: string
        quantity:
          type: integer
    Customer:
      type: object
      properties:
        customerType:
          type: string
```

```
        streetAddress:
          type: string
        city:
          type: string
        postalCode:
          type: string
      discriminator:
        propertyName: customerType
        mapping:
          person: '#/components/schemas/PersonCustomer'
          company: '#/components/schemas/CompanyCustomer'
```

`Product` is the simplest schema in our specification. It just defines that its type is a JSON object with two properties: `productSKU` of type `string`, which will be the ID for our products, and `quantity` of type `integer`, which will represent the availability of this product.

Moving to the `Customer` schema, it is an object containing four properties: `customerType`, `streetAddress`, `city`, and `postalCode`. All the properties are of type `string`. But here, we also start to introduce the concept of inheritance and polymorphism.

The `Customer` schema will serve as a parent (base) schema, which will be extended by two other schemas—`PersonCustomer` and `CompanyCustomer`—that we will introduce later. This setup demonstrates inheritance, with one parent and two child schemas, and polymorphism, allowing `Customer` to represent either `PersonCustomer` or a `CompanyCustomer`.

This is where the `discriminator` property comes into play. The `discriminator` property is used by the JSON parser to indicate which of the child schemas it is inheriting from when it tries to deserialize the JSON data into the generated object while loading from JSON and converting it into an object in runtime. In our case, the discriminator will be the `customerType` property. This property will define whether `Customer` will be of type `PersonCustomer` or type `CompanyCustomer`.

This property is used when you place an order, as part of the `POST` request body that we previously defined. With the discriminator, a single endpoint can accept two different bodies, one with `Customer` of type `PersonCustomer` for individual customers, and another one with `Customer` of type `CompanyCustomer` for companies.

Now, let us define the child schemas, `PersonCustomer` and `CompanyCustomer`, and relate them to the parent `Customer` schema:

```yaml
PersonCustomer:
  type: object
  allOf:
    - $ref: '#/components/schemas/Customer'
  properties:
    firstName:
      type: string
    lastName:
      type: string
CompanyCustomer:
  type: object
  allOf:
    - $ref: '#/components/schemas/Customer'
  properties:
    companyName:
      type: string
    vatId:
      type: string
```

Both schema definitions look similar; both of them use the `allOf` keyword that refers to the `Customer` schema. It defines that the child schema should contain (inherit) all the properties of the `Customer` schema.

Since we have implemented hierarchy in our specification, let us briefly look at how to create new orders in this structure, using the `OrderRequestBody` that will be used for `POST` and `PUT` methods to create and update resources in the application.

Figure 4.2 – OrderRequestBody schema in OpenAPI Swagger UI

The `POST` method expects `OrderRequestBody` in the body of the request. Since we have used `discriminator` in the definition, here we will need to select one of the bodies in the request, as specified by `oneOf->` in the preceding figure: `PersonCustomer`, `CompanyCustomer`, or `Customer`. The value for the `customerType` field needs to be one of `person`, `company`, or `customer` to match one of the bodies defined. The usage of `Customer` alone is discouraged since it does not benefit from the hierarchy and the polymorphism, but it is important to know that you can use it alone too without the extra fields from `PersonCustomer` or `CompanyCustomer` if needed.

Then, you can add the corresponding fields of each of these, so if it is a company, you will have fields such as `vatId` and `companyName` in your request body, and if it is a person, you will have fields such as `firstName` and `lastName` in your request. The fields mentioned in the `Customer` specification are common to both types of customers.

To finish the schema section of our API specification, we will define the request and response bodies used by the Order Management API:

```
  OrderRequestBody:
    type: object
    properties:
      customer:
        $ref: '#/components/schemas/Customer'
      products:
        type: array
        items:
          $ref: '#/components/schemas/Product'
  OrderResponse:
    type: object
    properties:
      id:
        type: string
      customer:
        $ref: '#/components/schemas/Customer'
      products:
        type: array
        items:
          $ref: '#/components/schemas/Product'
      totalAmount:
        type: number
      orderCreatedDate:
        type: string
        format: date-time
      orderUpdatedDate:
        type: string
        format: date-time
      status:
        $ref: '#/components/schemas/Status'
  OrderStatus:
    type: object
    properties:
      status:
```

```
      $ref: '#/components/schemas/Status'
Status:
  type: string
  enum: [ Pending, Approved, Cancelled ]
```

In the `OrderRequestBody` object definition, there are two properties – one that is of the `Customer` type defined above, using `$ref` to refer to its definition, and the `products` property, which is a JSON array that will represent a list of `Product` JSON objects within `OrderRequestBody`.

In the response schema definition, `OrderResponse` contains an array of `Product` JSON objects, containing every product for a particular order, the `customer` property that references the `Customer` schema and returns the customer data for this order, the `orderCreatedDate` and `orderUpdatedDate` properties with the date and time of the order creation and update if any, and the `status` property that references the `OrderStatus` defined, which references the `Status` `enum`, containing all the possible statuses for an order: `Pending`, `Approved`, or `Cancelled`. Since we can reuse these definitions anywhere around the specification, the `OrderStatus` schema is also referenced in the order change status endpoint as the request body definition for allowed values in the request.

Sharing schemas across APIs

With Java and other programming languages, we are used to defining reusable libraries to avoid repeating the same code in multiple projects. OpenAPI specifications can also refer to external files, so it is possible to share common parts among multiple APIs.

However, you should be aware that this approach creates coupling among the APIs. In case the common part changes, it can lead to unexpected inconsistencies. So, it is advisable to try to keep the individual API specifications self-contained. If you want to reuse schemas, you should ensure that they are very stable.

One such example could be the structure used for reporting errors. But in our example APIs, we use the `ProblemDetail` schema based on the RFC 7807 internet standard. This way, we avoid maintaining the shared schema.

Defining security schemes

Besides schemas, the components section of the OpenAPI specification can be used to specify security-related requirements for our API. Here, we will specify the authentication mechanism to use: OpenID Connect with JWT. You will learn more about security in *Chapter 7*.

```
securitySchemes:
  JWTAuth:
    type: http
    scheme: bearer
    bearerFormat: JWT
```

The `securitySchemes` defines a security scheme named `JWTAuth` and applies it globally to all operations in the API. Here is a breakdown of its components and their implications:

- `JWTAuth`: This is the identifier for the security scheme
- `type: http`: This specifies that the security scheme is HTTP-based
- `scheme: bearer`: This indicates that the scheme uses bearer tokens
- `bearerFormat: JWT`: This specifies that the bearer tokens are in JWT format

With all that defined, we can move away from the `components` keyword and get to know our last keyword in this definition, the `security` keyword:

```
security:
  - JWTAuth: []
```

This security definition is at the root level, and it indicates that there are no specific scopes or permissions required within the JWT for accessing the API. So, a valid JWT must be provided in the Authorization header for all API calls, but the token itself does not need to specify any particular scopes.

If you prefer not to use JWTs for authentication, you can consider using session state IDs as an alternative. They were widely used before the advent of JWTs, which replaced session state IDs for many reasons.

Before looking at how to use session state IDs, let us see the key reasons that JWTs became the preferred authentication method for microservices:

- **Statelessness**: JWTs are stateless, meaning they do not require the server to maintain session state. This is beneficial for microservices, which are designed to be stateless and scalable.

- **Scalability**: Since JWTs are self-contained and do not require server-side storage, they are more scalable in distributed systems compared to session state IDs, which require centralized session storage.
- **Decentralized authentication**: JWTs can be verified by any service that has a public key, allowing for decentralized authentication across multiple microservices without the need for a central session store.
- **Interoperability**: JWTs are a standard (RFC 7519) and are widely supported across different platforms and languages, making them suitable for heterogeneous microservice environments.
- **Security**: JWTs can be signed and optionally encrypted, providing integrity and confidentiality. They can also include claims that provide additional context about the user or the session.

But for specific circumstances, which could be dealing with legacy services and integrations with already existing services that make use of session state IDs, this approach may be necessary.

Session state IDs involve maintaining a session on the server side, where each session is identified by a unique session ID. This session ID is stored on the client side, typically in a cookie, and sent with each request to the server. The server then validates the session ID and retrieves the associated session data.

Some of the advantages of session state IDs are that the server has full control over the session, including the ability to invalidate sessions at any time and there is no need to handle token expiration and renewal as with JWT.

On the other hand, it requires maintaining session state on the server, which can be challenging in a distributed environment, and, unlike JWT, which is stateless, session state IDs require the server to maintain state.

Here is an example of how you can define session state IDs in your OpenAPI schema:

```yaml
components:
  securitySchemes:
    SessionIDAuth:
      type: apiKey
      in: cookie
      name: SESSIONID
security:
  - SessionIDAuth: []
```

In this example, the `SessionIDAuth` security scheme specifies that the session ID will be sent in a cookie named `SESSIONID`. The server will then validate this session ID to authenticate the user.

We will not be covering session state IDs in our services since they break the stateless goal of a RESTful microservice and having JWTs with all the advantages explained above, but it is good to know how you can use them with the specification-first approach.

Also, as mentioned before, we will go into the security subject in more depth in *Chapter 7*.

With that, we have our specification completed. You can save your file with a name that represents your API, ending with the `.yml` extension (short for YAML). In our example, the file will be named `Order_Management_API.yml`.

As mentioned earlier, you can see the complete file in the GitHub repository at `https://github.com/PacktPublishing/Mastering-RESTful-Web-Services-with-Java/tree/main/chapter4`.

Next, we are going to generate code from this specification, using `openapi-generator-maven-plugin`. Let us find out how.

Generating code from the specification

In order to generate code from a specification, the application needs to be prepared for it. In this section, you will configure a plugin and generate code from what we have specified in the specification file.

The following XML fragment configures `openapi-generator-maven-plugin` for our Maven project, specifically designed to generate Spring-based Java code from the OpenAPI specification defined in the previous section. This plugin facilitates the automatic creation of API endpoints, models, and configuration classes.

We are going to configure the plugin in the `pom.xml` file inside our project. You can refer to the GitHub repository to see the full file, but here we will focus on the specific changes to accomplish our goals. Let us look at this configuration now:

```
<plugin>
    <groupId>org.openapitools</groupId>
    <artifactId>openapi-generator-maven-plugin</artifactId>
    <version>7.5.0</version>
    <executions>
        <execution>
            <goals>
```

```
                <goal>generate</goal>
            </goals>
            <configuration>
                <inputSpec>${project.basedir}/your_spec_API.yml
                </inputSpec>
                <generatorName>spring</generatorName>
                <apiPackage>[package where generated code is added]
                </apiPackage>
                <modelPackage>[new package where DTOs are added]
                </modelPackage>

                <modelNameSuffix>Dto</modelNameSuffix>
                <configOptions>
                   <documentationProvider>springdoc
                   </documentationProvider>
                    <interfaceOnly>true</interfaceOnly>
                    <oas3>true</oas3>
                    <openApiNullable>false</openApiNullable>
                    <serializableModel>true</serializableModel>
                    <useBeanValidation>true</useBeanValidation>
                    <useSpringBoot3>true</useSpringBoot3>
                    <useTags>true</useTags>
                </configOptions>
            </configuration>
        </execution>
    </executions>
</plugin>
```

Here is a breakdown of its key components:

- **Plugin identification**: `groupId` and `artifactId` identify the plugin within the Maven ecosystem:

  ```
  <groupId>org.openapitools</groupId>
  <artifactId>openapi-generator-maven-plugin</artifactId>
  ```

- **Plugin version**: Specifies the version of the plugin to use, ensuring compatibility and access to specific features available in this version:

  ```
  <version>7.5.0</version>
  ```

- **Execution configuration**: The `executions` block defines when and how the plugin should run. The goal named `generate` triggers the code generation process:

```
<goals>
    <goal>generate</goal>
</goals>
```

- **Specification input**: The `inputSpec` configuration points to the OpenAPI specification file. This path is relative to the project's base directory, indicating where the plugin should look for the API definition elaborated in the previous section:

```
<inputSpec>${project.basedir}/src/main/resources/Order_Management_
API.yml</inputSpec>
```

- **Generator configuration**: The `generatorName` specifies that the generated code should be tailored for Spring, influencing the structure and annotations used in the output:

```
<generatorName>spring</generatorName>
```

- **Package names**: The `apiPackage` and `modelPackage` configurations define the Java package names for generated API interfaces (API operations) and model classes (data structures), respectively. This helps in organizing the generated code within the project structure. Also, having a suffix added by `modelNameSuffix` helps to have the DTOs generated with this suffix in the name:

```
<apiPackage>com.packt.ordermanagementapi.adapter.inbound.rest
</apiPackage>
<modelPackage>com.packt.ordermanagementapi.adapter.inbound.rest.dto
</modelPackage>
<modelNameSuffix>Dto</modelNameSuffix>
```

- **Additional options**: The `configOptions` section provides further customization of the generated code. Let us briefly describe each of the options we are using here and their functionality:
    - `documentationProvider`: This specifies the documentation provider to use. In this case, `springdoc` is used to generate API documentation.
    - `interfaceOnly`: When set to `true`, only interfaces for the API are generated, without any implementation.
    - `oas3`: This indicates that the OpenAPI 3.0 specification is being used.

- `openApiNullable`: When set to `false`, the generator will not use the `@Nullable` annotation for optional fields.
- `serializableModel`: When set to `true`, the generated models will implement the `Serializable` interface.
- `useBeanValidation`: When set to `true`, the generated models will include annotations for bean validation (e.g., `@NotNull`, `@Size`).
- `useSpringBoot3`: When set to `true`, the generator will produce code compatible with Spring Boot 3.
- `useTags`: When set to `true`, the generator will use tags defined in the OpenAPI specification to group API operations.

```
<configOptions>
   <documentationProvider>springdoc</documentationProvider>
    <interfaceOnly>true</interfaceOnly>
    <oas3>true</oas3>
    <openApiNullable>false</openApiNullable>
    <serializableModel>true</serializableModel>
    <useBeanValidation>true</useBeanValidation>
    <useSpringBoot3>true</useSpringBoot3>
    <useTags>true</useTags>
</configOptions>
```

With that, we have covered every aspect of the configuration of the plugin in our `pom.xml` file.

Next, let us execute the build and have the implementation generated:

```
mvn clean install
```

If your build was successful, you will note that you have the DTOs and an interface generated in your target folder.

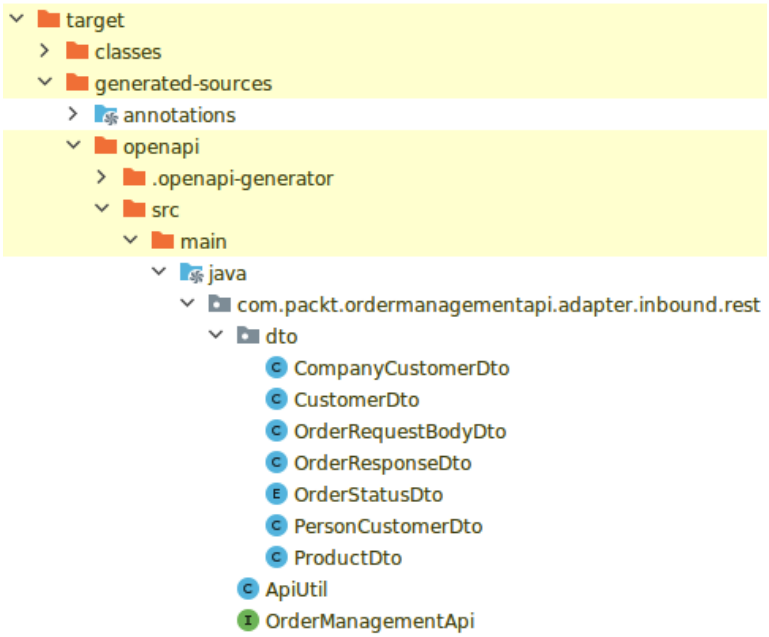


Figure 4.3 – Implementation generated in the target folder

If you use IntelliJ IDEA, switch to the **Packages** view, so you can see the generated sources along with the structure of the project and work seamlessly with the generated sources and your own implementation, as seen in the following screenshot:

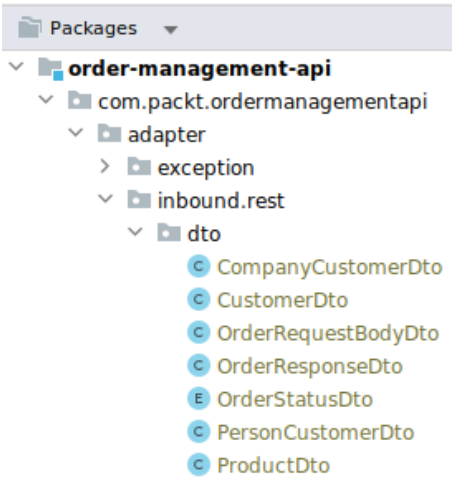


Figure 4.4 – Implementation generated along with the project structure

By generating the code in the `target` folder, we clearly separate it from the manually written code in the `src` folder. The generated code should not be stored in a version control system (Git or another) to avoid inconsistencies with the source specification it should be generated from. Instead, fresh code is generated during every Maven build.

Let us see the results in the generated `PersonCustomerDto` class:

```
@Generated(value = "org.openapitools.codegen.languages.SpringCodegen",
date = "2024-07-21T20:59:33.440034506-03:00[America/Sao_Paulo]", comments
= "Generator version: 7.5.0")
public class PersonCustomerDto extends CustomerDto implements Serializable
{

  private static final long serialVersionUID = 1L;

  private String firstName;

  private String lastName;

  public PersonCustomerDto firstName(String firstName) {
    this.firstName = firstName;
    return this;
  }
...getters and setters omitted for brevity
```

Here, we can see that for every field that was specified in the specification schema, a Java class field was generated accordingly, including the relationships with other schemas, such as the inheritance from the `Customer` schema, proving that we can also model object-oriented programming concepts using the API-first approach if we define the relationships in the specification file properly. The other classes look similar, each one generated according to its own schema definition.

Before moving on to the implementation of the operations listed in the API specification, let us see what our API paths and methods were translated to. For example, let us see the `DELETE` method:

```
/**
 * DELETE /orders : Cancel Order
 * Cancels an existing order
 *
 * @param orderId ID of the order to cancel (required)
 * @return No Content (status code 204)
 *         or Not Found (status code 404)
```

```
 */
@Operation(
    operationId = "ordersDelete",
    summary = "Cancel Order",
    description = "Cancels an existing order",
    tags = { "OrderManagement" },
    responses = {
        @ApiResponse(responseCode = "204", description = "No Content"),
        @ApiResponse(responseCode = "404", description = "Not Found")
    },
    security = {
        @SecurityRequirement(name = "JWTAuth")
    }
)
@RequestMapping(
    method = RequestMethod.DELETE,
    value = "/orders"
)

default ResponseEntity<Void> ordersDelete(
    @NotNull @Parameter(name = "orderId",
        description = "ID of the order to cancel", required = true,
        in = ParameterIn.QUERY) @Valid @RequestParam(value = "orderId",
        required = true) String orderId
) {
    return new ResponseEntity<>(HttpStatus.NOT_IMPLEMENTED);

}
```

Here, we can see that the generated default implementation code does nothing; it only returns a response entity with the status `NOT_IMPLEMENTED`. If we tried to invoke the API with this default code, the client would see a `501 Not Implemented` HTTP status.

The default implementation is expected to be overridden by the actual implementation. The path, its parameters, and responses are already defined and documented. The API documentation of the running API application is accessible using the Swagger UI interface or by downloading the OpenAPI JSON definition through the auto-generated `/v3/api-docs` path, like with the code-first API we developed in *Chapter 2*.

Just keep in mind that you will need to add the `springdoc-openapi-starter-webmvc-ui` dependency into your `pom.xml` file to be able to use the Swagger UI graphical interface. Please refer to the GitHub repository of *Chapter 4* to get the full list of dependencies needed to run the project.

This is what makes the specification-first approach handy, since after the design is defined, developers can focus on the implementation while consumers can start the integration with the system even before the actual development is finished.

Now that our specification is generated, let us have an overview of the package structure of this service.

Package structure of the Order Management API

Let us have a look at what our package structure looks like for the Order Management API:

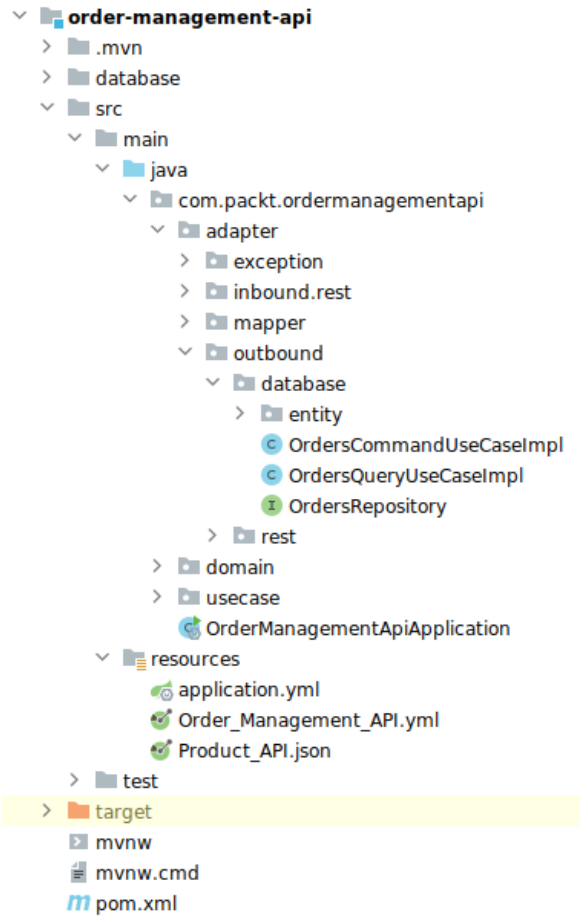


Figure 4.5 – Package structure of implemented Order Management API

Here, we can see that the Order Management API follows the same architectural approach as described in *Chapter 2*, following the principles of clean architecture. Let us look at the contained packages:

- `adapter`: This package contains the four following packages:
    - `exception`: This contains all class exceptions and exception handlers
    - `inbound.rest`: This is responsible for handling all the inputs, transformations, and generated data from the specification
    - `mapper`: This contains all the mapper transformations between objects, such as `toOrderRequest` from `OrderRequestBodyDto`, and the custom-defined mappers.
    - `outbound`: This package contains all elements that interact with external services, such as databases and other services. In this service, we will have two subpackages within it:
        - `outbound.database`: This contains all the elements that will interact with the database itself, the implementation of use cases that contain operations such as `createOrder` and `updateOrderStatus`, and the repositories also go here. It contains an internal package within it.
        - `outbound.database.entity`: This contains all the defined entities for the service that declares the tables and interacts with the database.
- `outbound.rest`: This package contains all the elements required to interact with external services through REST calls. In this service, we have the elements required to interact with the Product API, created in *Chapter 2*.
- `domain`: This package contains the POJOs that define the base structure for each domain contained by this service. Its structure will be inherited mostly by the entities, but they are not the entities themselves (those are defined in the `adapter.outbound.database.entity` package), so it contains POJOs from the `Order`, `Customer` (also `PersonCustomer` and `CompanyCustomer`), and `Product` definitions, along with the `StatusEnum` definitions as well.
- `usecase`: This contains the interface definitions for the use cases, containing the actions that will be implemented. In this service, we have two use cases:
- `OrdersCommandUseCase`: This contains the definitions of actions like `createOrder` and `updateOrder` that are implemented by the class `OrdersCommandUseCaseImpl` in the `adapter.outbound.database`

- `OrdersQueryUseCase`: This contains the definition of the actions that happen with the database through queries and is implemented by `OrdersQueryUseCaseImpl` in the `adapter.inbound.rest` package.

Now that we are aware of the package structure of the service, let us jump into the implementation of the controller itself.

Implementing the Order Management API controller

The Order Management API will follow the same architecture and package structure used in the Product API, so the controllers will be created within the `adapter.inbound.rest` package.

The first step is to create the `OrderManagementApiController` class, implementing the methods of the `OrderManagementApi` interface. Here is a sample of the code:

```
@Controller
public class OrderManagementApiController implements OrderManagementApi {

    @Override
    public ResponseEntity<List<OrderResponse>> ordersGet() {
        //Add your own concrete implementation
        return OrderManagementApi.super.ordersGet();
    }

    @Override
    public ResponseEntity<Void> ordersDelete(String orderId) {
        //Add your own concrete implementation
        return OrderManagementApi.super.ordersDelete(orderId);
    }

    @Override
    public ResponseEntity<OrderResponse> ordersOrderIdGet(String orderId)
    {
        //Add your own concrete implementation
        return OrderManagementApi.super.ordersOrderIdGet(orderId);
    }
```

```
    @Override
    public ResponseEntity<OrderResponse> ordersOrderIdPut(String orderId,
        OrderRequestBody orderRequestBody) {
        //Add your own concrete implementation
        return OrderManagementApi
            .super.ordersOrderIdPut(orderId, orderRequestBody);
    }

    @Override
    public ResponseEntity<OrderResponse> ordersPost(
        OrderRequestBody orderRequestBody) {
        //Add your own concrete implementation
        return OrderManagementApi.super.ordersPost(orderRequestBody);
    }
}
```

In the preceding snippet, there is no concrete implementation yet, but we are defining the methods that were specified previously in our specification file and generated in the `OrderManagementApi` interface. If we keep the `super` calls (invoking the default implementation of the parent interface), the server returns a `501 Not Implemented` HTTP status code to the caller. It can be useful when you want to deploy your application partially to let the users know that this feature is not yet available.

All the definitions in the OpenAPI specification file serve as documentation for the API. If you paste the specification into the online `editor.swagger.io` tool or run the application and open the Swagger UI with your browser at the URL `http://localhost:8080/swagger-ui/index.html`, you will be able to see every operation and schema defined until now. This documentation represents the API contract, describing all the available resources, methods available to interact with them, and what data the API is expected to receive and return.

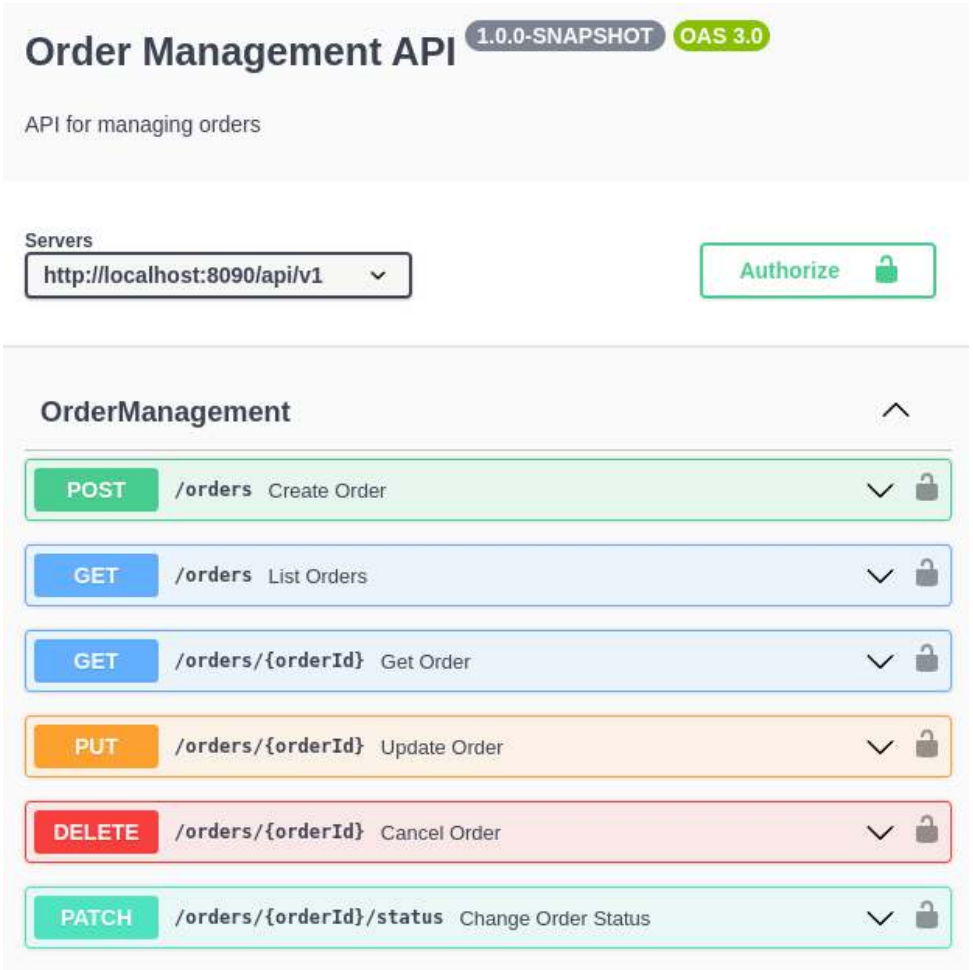


Figure 4.6 – Swagger UI from the Order Management API specification

This is the first step toward a concrete implementation of the operations defined in our API specification.

For the sake of brevity, we are not going to look at every step of the implementation process here, since the goal of this chapter is to show you how to have the code generated from the specification, as we saw in the previous sections. Feel free to refer to the source code in the GitHub repository for *Chapter 4* to have a deeper understanding of the implementation details, since this goes much deeper than what we are going to describe in this section alone.

This is what the controller looks like after being implemented:

```
@RestController
public class OrderManagementApiController implements OrderManagementApi {
    private final OrdersCommandUseCase ordersCommandUseCase;
    private final OrdersQueryUseCase ordersQueryUseCase;
    private final OrderMapper orderMapper;

    public OrderManagementApiController(
    OrdersCommandUseCase ordersCommandUseCase,
    OrdersQueryUseCase ordersQueryUseCase,
    OrderMapper orderMapper) {
        this.ordersCommandUseCase = ordersCommandUseCase;
        this.ordersQueryUseCase = ordersQueryUseCase;
        this.orderMapper = orderMapper;
    }

    @Override
    public ResponseEntity<List<OrderResponseDto>> ordersGet() {
        final var orders = ordersQueryUseCase.getAllOrders()
            .stream()
            .map(orderMapper::toOrderResponse)
            .toList();
        return ResponseEntity.status(HttpStatus.OK)
            .body(orders);
    }

    @Override
    public ResponseEntity<OrderResponseDto> ordersOrderIdGet(
        String orderId) {
        final var order = ordersQueryUseCase.getOrder(orderId);
        return ResponseEntity.status(HttpStatus.OK)
            body(orderMapper.toOrderResponse(order));
    }

    @Override
    public ResponseEntity<OrderResponseDto> ordersPost(
        OrderRequestBodyDto orderRequestBody) {
```

```
        final var order = ordersCommandUseCase.createOrder(
           orderMapper.toOrderRequest(orderRequestBody));
        return ResponseEntity.status(HttpStatus.CREATED).body(
            orderMapper.toOrderResponse(order));
    }

    @Override
    public ResponseEntity<OrderResponseDto> ordersOrderIdStatusPatch(
        String orderId, OrderStatusDto orderStatus) {
        final var order = ordersCommandUseCase.updateOrderStatus(orderId,
            StatusEnum.valueOf(orderStatus.getStatus().name()));
        return ResponseEntity.status(HttpStatus.OK).body(
            orderMapper.toOrderResponse(order));
    }

    @Override
    public ResponseEntity<OrderResponseDto> ordersOrderIdPut(
        String orderId, OrderRequestBodyDto orderRequestBody) {
        var orderRequest = orderMapper.toOrderRequest(orderRequestBody);
        final var order = ordersCommandUseCase.updateOrder(
            orderId, orderRequest);
        return ResponseEntity.status(HttpStatus.OK).body(
            orderMapper.toOrderResponse(order));
    }

    @Override
    public ResponseEntity<Void> ordersOrderIdDelete(String orderId) {
        ordersCommandUseCase.deleteOrder(orderId);
        return ResponseEntity.noContent().build();
    }
}
```

Looking at the implementation of the controller, we can notice some similar patterns as were implemented in the Product API, especially the usage of use cases and mappers.

But unlike the Product API, here we override the generated code of the `OpenManagementAPI` interface where all the paths and operations are defined. Unlike the code-first approach, the documentation that has been already defined previously in the OpenAPI specification does not need to be repeated manually here, since it is contained within the generated `OpenManagementAPI`. This is the power of a specification-first (API-first) implementation.

In this controller, we are injecting three main interfaces that are used for the processing of all the business logic, persistence, and data layer transformation:

- `OrdersCommandUseCase`
- `OrdersQueryUseCase`
- `OrderMapper`

Let us first focus on the use case implementations of this service, which are `OrdersCommandUseCase` and `OrdersQueryUseCase`, and their purpose within the Order Management API, starting with `OrdersCommandUseCaseImpl`.

Implementing OrdersCommandUseCase

Let us start by implementing `OrdersCommandUseCase` with the `OrdersCommandUseCaseImpl` class:

```
@Transactional
@Service
public class OrdersCommandUseCaseImpl
       implements OrdersCommandUseCase {

    private final OrdersRepository ordersRepository;
    private final OrderMapper orderMapper;
    private final ProductsApi productsApi;

    public OrdersCommandUseCaseImpl(OrdersRepository ordersRepository,
                                    OrderMapper orderMapper,
                                    ProductsApi productsApi) {
        this.ordersRepository = ordersRepository;
        this.orderMapper = orderMapper;
        this.productsApi = productsApi;
    }

    @Override
    public Order createOrder(OrderRequest orderRequest) {
```

```
        BigDecimal total = getProductPrice(orderRequest);
        Order order = orderMapper.toOrder(orderRequest);
        order.setTotalAmount(total);

        return ordersRepository.save(OrderEntity.fromOrder(order, null));
    }

    @Override
    public Order updateOrder(String orderId, OrderRequest orderRequest) {
        Optional<OrderEntity> optionalOrderEntity
                = ordersRepository.findById(orderId);
        if (optionalOrderEntity.isPresent()) {
            BigDecimal total = getProductPrice(orderRequest);
            Order order = orderMapper.toOrder(orderRequest);
            order.setTotalAmount(total);
            return ordersRepository
                .save(OrderEntity.fromOrder(order,
                    optionalOrderEntity.get()));
        }
        throw new EntityNotFoundException("Order not found with id " +
        orderId);
    }

    private BigDecimal getProductPrice(OrderRequest orderRequest) {
        return orderRequest.products().stream().map(orderProduct -> {
            ProductOutputDto catalogueProduct =
                productsApi.getProductById(orderProduct.getProductSKU());
            return catalogueProduct
                .getPrice()
                .multiply(BigDecimal.valueOf(
                    orderProduct.getQuantity()));
        }).reduce(BigDecimal.ZERO, BigDecimal::add);
    }

    @Override
    public Order updateOrderStatus(String orderId, StatusEnum status) {
        Optional<OrderEntity> optionalOrderEntity = ordersRepository.
findById(orderId);
```

```
        if (optionalOrderEntity.isPresent()) {
            return ordersRepository
                .save(OrderEntity
                    .updateOrderStatus(optionalOrderEntity.get(),
                                       status));
        }
        throw new EntityNotFoundException("Order not found with id " +
        orderId);
    }

    @Override
    public void deleteOrder(String orderId) {
        ordersRepository.deleteById(orderId);
    }
}
```

As described earlier in the package structure, the `OrdersCommandUseCaseImpl` class contains the business logic of the service itself, the actions, and interactions with other components such as the repository and mapper, and even other services such as the Product API, as we will see in detail in the *Communicating with the Product API* section. That is why this class is annotated with the `@Service` annotation from Spring, indicating that it is a `Service` stereotype of Bean.

Implementing OrdersQueryUseCase

Moving to the implementation of `OrdersQueryUseCase`, we will have `OrdersQueryUseCaseImpl`:

```
@Service
public class OrdersQueryUseCaseImpl implements OrdersQueryUseCase {

    private final OrdersRepository ordersRepository;

    public OrdersQueryUseCaseImpl(OrdersRepository ordersRepository) {
        this.ordersRepository = ordersRepository;
    }

    @Override
    public List<? extends Order> getAllOrders() {
        return ordersRepository.findAll();
    }
```

```
    @Override
    public Order getOrder(String orderId) {
        return ordersRepository.findById(orderId)
           .orElseThrow(
               () -> new EntityNotFoundException(
                   "Order not found with id " + orderId)
               );
    }
}
```

In `OrdersQueryUseCaseImpl`, we have a much smaller implementation, since here the focus is the interaction with the database. Any custom queries or requests that you would have just go into this class.

Configuring OrderMapper with MapStruct

And finally, there's the `OrderMapper` interface. Unlike the use case classes, we do not implement any class here; instead, we define an interface with the mappings we want to have in our project. Since this project uses MapStruct for object mapping, we define the mapping within this interface and any custom mappings separately.

MapStruct

MapStruct is a code generator that simplifies the process of mapping between Java bean types by relying on a convention-over-configuration strategy. That is why we are defining this interface; it contains the configuration that MapStruct needs to generate the mapping code. Also, we are going to define a custom mapping for the cases when the basic generated mapping code is not enough.

Let us start with the interface definition:

```
@Mapper(uses = CustomerCustomMapper.class, componentModel = "spring")
public interface OrderMapper {

    Order toOrder(OrderRequest orderRequestBody);

    @Mapping(target = "customer", source = "customer",
        qualifiedByName = "customerDtoToCustomer")
```

```
    OrderRequest toOrderRequest(OrderRequestBodyDto orderRequestBody);

    @Mapping(target = "customer", source = "customer",
        qualifiedByName = "customerToCustomerDto")
    OrderResponseDto toOrderResponse(Order order);
}
```

This is what the `OrderMapper` interface looks like. We are defining the mapping operations that we want MapStruct to perform, such as the `toOrderRequest` operation, which is a mapping from an `OrderRequestBodyDto` object to an `OrderRequest` object.

Simple mapping with `Mapstruct` is straightforward, but when things get complicated, there is a need to introduce a custom mapping implementation for it. This is where the `CustomerCustomMapper` class gets into our mapping definition.

To have a custom mapping used with MapStruct, you need to create a custom mapping class and tell the interface to use it.

Here, we are adding the `CustomerCustomMapper` class in the `uses` property within the `@Mapper` annotation at the class level. This way, MapStruct knows that there is a custom implementation mapper that it needs to refer to.

This approach allows for partial or complete object mapping, depending on your requirements. In this case, we will perform partial mapping by using parts of the generated mapping from MapStruct and applying custom mapping to handle a specific complex object that the simple mapping cannot handle correctly.

For every mapping operation, such as `toOrder`, `toOrderRequest`, and `toOrderResponse`, if you have a custom mapping that changes the default mapping behavior from MapStruct, you need to specify it with the `@Mapping` annotation at the method level to tell MapStruct that you have a custom mapping.

In our custom mapping example, both `target` and `source` properties specify that the `customer` property within the received parameter (`orderRequestBody` in the `toOrderRequest` method and `order` in the `toOrderResponse` method) is going to be mapped in a custom way that needs to refer to the custom mapper implementation that is implemented by the `CustomerCustomMapper` class.

Finally, the `qualifiedByName` property tells MapStruct which method to refer to for the mapping into the custom mapper, being identified by the `@Named` annotation.

Due to the complexity and polymorphism introduced to have distinct types of customers with inheritance, this custom map was needed.

So, let us look at this custom mapping specification:

```
@Component
public class CustomerCustomMapper {

    @Named("customerDtoToCustomer")
    public Customer customerDtoToCustomer(CustomerDto customerDto) {
        if (customerDto == null) {
            return null;
        }

        Customer customer = switch (customerDto.getCustomerType()) {
            case "person" -> {
                PersonCustomer personCustomer = new PersonCustomer();
                var personCustomerDto = (PersonCustomerDto) customerDto;
                personCustomer.setFirstName(
                    personCustomerDto.getFirstName());
                personCustomer.setLastName(
                    personCustomerDto.getLastName());
                yield personCustomer;
            }
            case "company" -> {
                CompanyCustomer companyCustomer = new CompanyCustomer();
                var companyCustomerDto = (CompanyCustomerDto) customerDto;
                companyCustomer.setCompanyName(
                    companyCustomerDto.getCompanyName());
                companyCustomer.setVatId(companyCustomerDto.getVatId());
                yield companyCustomer;
            }
            default -> new Customer();
        };

        customer.setStreetAddress(customerDto.getStreetAddress());
        customer.setCity(customerDto.getCity());
        customer.setPostalCode(customerDto.getPostalCode());

        return customer;
    }
```

```
    @Named("customerToCustomerDto")
    public CustomerDto customerToCustomerDto(Customer customer) {
        if (customer == null) {
            return null;
        }

        CustomerDto customerDto = switch (customer) {
            case PersonCustomerEntity personCustomer -> {
                PersonCustomerDto personCustomerDto = new
                PersonCustomerDto();
                personCustomerDto.setFirstName(
                    personCustomer.getFirstName());
                personCustomerDto.setLastName(
                    personCustomer.getLastName());
                personCustomerDto.setCustomerType("person");
                yield personCustomerDto;
            }
            case CompanyCustomerEntity companyCustomer -> {
                CompanyCustomerDto companyCustomerDto = new
                CompanyCustomerDto();
                companyCustomerDto.setCompanyName(companyCustomer.
                getCompanyName());
                companyCustomerDto.setVatId(companyCustomer.getVatId());
                companyCustomerDto.setCustomerType("company");
                yield companyCustomerDto;
            }
            default -> new CustomerDto();
        };
        customerDto.setStreetAddress(customer.getStreetAddress());
        customerDto.setCity(customer.getCity());
        customerDto.setPostalCode(customer.getPostalCode());

        return customerDto;
    }
}
```

Since this is a MapStruct component, it is annotated `@Component` so it can be created as a Bean in the Spring context and made available to be detected by the mapping generator during the build process.

But the key aspect of it is the `@Named` method-level annotation, as mentioned earlier. It specifies the unique qualifier mentioned in the `qualifiedByName` MapStruct interface we just saw. The `customer` object is the only object being custom-mapped in the `CustomerCustomMapper` implementation.

With that, when you generate your project using MapStruct, it will generate the mappings as specified in the `OrderMapper` interface, but it will call this custom-mapping implementation, as we can see in the generated code snippet in the following screenshot:

```
public OrderRequest toOrderRequest(OrderRequestBodyDto orderRequestBody) {  2 usages
    if (orderRequestBody == null) {
        return null;
    } else {
        Customer customer = null;
        List<Product> products = null;
        customer = this.customerCustomMapper.customerDtoToCustomer(orderRequestBody.getCustomer());
        products = this.productDtoListToProductList(orderRequestBody.getProducts());
        OrderRequest orderRequest = new OrderRequest(customer, products);
        return orderRequest;
    }
}
```

Figure 4.7 – Generated mapping to OrderRequest calling customer custom mapping

You can find the generated code from the preceding screenshot in the `OrderMapperImpl` generated class after building your project under the `target` folder, within the `classes` folder, in the `mapper` package, where the `OrderMapper` interface and the `CustomerCustomMapper` class are also defined. But if you are using the **Packages** view in IntelliJ, the generated classes will be shown to you automatically in the `mapper` package with your own implemented classes, as seen before in this chapter.

Now, let us look at how the Order Management API can communicate with the Product API to validate the products in the received orders.

Communicating with the Product API

To validate whether the products of an order exist, the Order Management API validates the inserted product SKUs against the Product API through an API call either when a new order is created or when an order update contains an insertion of a new product into the order.

In real-world production applications, the Product API may need to validate whether there are enough products available in stock to be added to the order and respond properly if there are not. For our purposes of demonstrating how to create APIs, we will show how to validate whether the products of an order exist within Product API service data and retrieve their prices to be added to our order. If a product is not found, the Order Management API will return a proper message with a `404 Not Found` HTTP status code and the error message `Product not found`, explaining that the order cannot be placed because that product does not exist and needs to be created in the Product API.

To make this call, we will generate the client from the Product API specification. Since this chapter is about generating code, why not generate the client to retrieve the data from the Product API itself?

You saw in the previous section, in the `OrdersCommandUseCaseImpl` implementation, the `updateProductPrice` method. This private method makes use of `productsQueryUseCase.getProductById` to retrieve the data of the product with the generated client and calculate the total value of the order based on the products it contains.

To have this client generated, we will need to update our generator plugin, adding the Product API specification to it, and generating a new build.

Let us start with updating the OpenAPI generator plugin. We will be adding a second execution to it, below the first execution created at the beginning of this chapter. This is what it looks like:

```
<execution>
    <id>generate-client</id>
    <goals>
        <goal>generate</goal>
    </goals>
    <configuration>
        <inputSpec>.../src/main/resources/Product_API.json</inputSpec>
        <generatorName>java</generatorName>
        <library>restclient</library>
        <apiPackage>[package where generated code is added]</apiPackage>
        <modelPackage>[new package where DTOs are added]</modelPackage>
        <modelNameSuffix>Dto</modelNameSuffix>
        <generateApiTests>false</generateApiTests>
        <generateModelTests>false</generateModelTests>
        <configOptions>
```

```
            <interfaceOnly>true</interfaceOnly>
            <useJakartaEe>true</useJakartaEe>
            <openApiNullable>false</openApiNullable>
        </configOptions>
    </configuration>
</execution>
```

The configuration here is similar to what we did in the first generator using the Order Management API specification, in the *Generating code from the specification* section, but now we will point to the Product API specification instead. There are only a few differences that we will detail now and, as we did for the first generator, let us go through each of the items to have them detailed:

- `<execution>`: This defines a specific execution of the plugin. Each execution can have its own configuration and goals. Here, we are adding a new one.
- `<id>`: This is a unique identifier for this execution. In this case, it is `generate-client`. In the previous execution, we used `generate-server` as the ID.
- `<goals>`: This specifies the goals to be executed. Here, the goal is `generate`, which triggers the code generation process.
- `<configuration>`: This contains the configuration options for this execution.
- `<inputSpec>`: This specifies the path to the OpenAPI specification file. Here, it is `${project.basedir}/src/main/resources/Product_API.json`, which is where the Product API specification is contained within this service. The specification is in the JSON format that we could download using the Product API Swagger UI.
- `<generatorName>`: This defines the generator to use. In this case, it is `java`, indicating that Java client code will be generated.
- `<library>`: This specifies the library to use for the generated code. Here, it is `restclient`, which generates a REST client.
- `<apiPackage>`: This defines the package for the generated API classes. Here, it is `com.packt.ordermanagementapi.adapter.outbound.rest`.
- `<modelPackage>`: This defines the package for the generated model classes. Here, it is `com.packt.ordermanagementapi.adapter.outbound.rest.dto`.
- `<modelNameSuffix>`: This adds a suffix to the names of the generated model classes. Here, it is `Dto`.
- `<generateApiTests>`: When set to `false`, API test classes will not be generated.
- `<generateModelTests>`: When set to `false`, model test classes will not be generated.

- `<configOptions>`: This contains additional configuration options for the generator.
- `<interfaceOnly>`: When set to `true`, only interfaces for the API are generated, without any implementation.
- `<useJakartaEe>`: When set to `true`, the generated code will use Jakarta EE instead of Java EE.
- `<openApiNullable>`: When set to `false`, the generator will not use the `@Nullable` annotation for optional fields.

Having our plugin configured and pointing to the Product API specification, we can proceed to generate a new build that will generate the REST client from the specification, along with the DTOs needed for it. Again, we will run the Maven command:

```
mvn clean install
```

After a successful build, this will generate the DTOs and the API classes into the specified packages that were configured in the plugin, as you can see in the following screenshot:

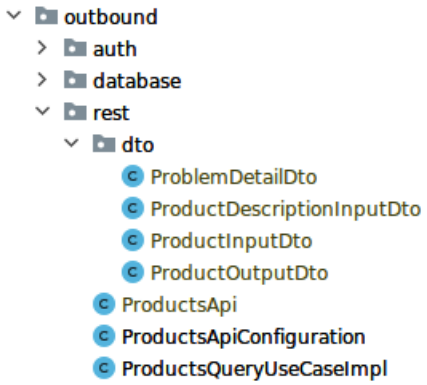


Figure 4.8 – Generated Product API client and DTO classes

This approach is extremely powerful since we already have the specification of the service that our API is going to consume. This can save hours of development time just by having the code and the client generated quickly for you.

You could think of an alternative solution where the client code would be generated as a module by the service providing the (Product) API and a client interested in consuming the API would just use the code as a library in the form of a JAR file.

This may look like making the life of the consumer implementers even easier, but it has serious drawbacks. It would force the consumer to be compatible with the library (and a particular version of it) used inside the JAR file. It would not support non-Java consumers.

One of the advantages of microservices architecture is the loose coupling between the services. Depending on JAR files to consume APIs of different microservices creates unwanted coupling and defeats the purpose of making the application distributed.

If you noticed when we looked at the `OrdersCommandUseCaseImpl` class in the previous section, there was a private method that uses this generated Product API client called `updateProductPrice`. Let us look at how it works and how to make the best use of it inside the Order Management API:

```
    private void updateProductPrice(List<Product> products) {
        products.forEach(orderProduct -> {
            ProductDetails cataloguProduct = productsQueryUseCase.
            getProductById(orderProduct.getProductSKU());
            orderProduct.setPrice(cataloguProduct.price());
        });
    }
```

Here, `productsQueryUseCase` is injected into `OrdersCommandUseCaseImpl`, where all related calls to the external Product API are handled. Within the `updateProductPrice` method, the `getProductById` method from `ProductsQueryUseCase` is called for each product in the order request, passing the unique product identifier (product SKU) to retrieve the product details.

Our goal is to validate whether these products exist in the Product API and get their value to be added to the total of the order.

If the product is found, it will be returned and attributed to the `ProductDetails` variable named `cataloguProduct`, and the price of the individual product will be set in the products list.

If any of the products in the order are not found in the Products API, `404 Not Found` will be returned from this external API call and will be caught by the `GlobalExceptionHandler` of the application, interrupting the order creation. This is the expected behavior since we do not want to allow any order to be created with a product that does not exist.

If you are curious about the code generated by the plugin and how the REST client was built, we encourage you to generate your own code to see it in action following these steps. You can also always refer to the chapter code in the official repository of this book at `https://github.com/PacktPublishing/Mastering-RESTful-Web-Services-with-Java/tree/main/chapter4`.

Summary

In this chapter, you learned about the main differences between specification-first, (API-first), and code-first development. We developed an OpenAPI specification from scratch in detail, going through each part of its structure, creating paths and methods with parameters, defining schemas for generating the DTOs to use in the requests, responses, and as parameters, and defining a security scheme.

Then we configured the Order Management API service with the OpenAPI code generator plugin for Maven and looked at each necessary parameter to have the code generated properly as expected and ran a successful build using the OpenAPI plugin.

After that, we started to look at the generated code, where it is located, and how to see it along with your own code implementation.

We went through an overview of the architectural package structure for the project and the objective of each package in the structure.

Next, we implemented our controller, using the generated interface from the specification, initially by overriding the methods and then by creating a real working implementation with all the features expected for this service, with a deep dive into the code.

Finally, we learned how to expand our code generator by adding a new execution into the OpenAPI plugin and generated a REST client to call the Product API.

We used the generated Product API client to call and validate the products in the creation or update flow of an order, checking whether it exists in the Product API and computing the total of the order. If any product is not found, it just aborts the creation or update of the order.

In the next chapter, you will get to know more about some advanced API concepts – how to handle timeouts, retries, and more – that could be used in a scenario where the Product API is unavailable at a certain request of the Order Management API, and how to handle it properly.

Part 2

Enhancing Your API

This part builds upon the foundation of RESTful API development by focusing on improving your API's maintainability, security, and reliability. You'll explore how to manage API versioning and evolution, implement advanced features such as pagination and resilience patterns, secure your endpoints effectively, and adopt modern testing strategies that include AI-assisted techniques.

This part will cover the following chapters:

- *Chapter 5, Managing API Evolution*
- *Chapter 6, Advanced API Concepts and Implementations*
- *Chapter 7, Securing Your RESTful APIs*
- *Chapter 8, Testing Strategies for Robust APIs*

5

Managing API Evolution

Updating an API is a frequent and necessary process as business requirements evolve and new features are introduced. However, it requires careful decision-making to avoid breaking existing client integrations. Even small changes to an API's structure, behavior, or response format can have significant impacts on consumers who depend on a stable, predictable interface.

In this chapter, we will show you how to evolve your APIs while ensuring backward compatibility. We will also discuss strategies for versioning APIs in cases when a breaking change is necessary, using the example of our product API.

We will begin the chapter by exploring different **versioning strategies** that can be applied to RESTful APIs. **Versioning** is a critical aspect of API design, as it ensures that your application can evolve over time without breaking existing client integrations. We will examine several versioning techniques, such as URL versioning, query parameter versioning, HTTP header versioning, and content negotiation, while discussing the challenges each approach presents, including maintenance, compatibility, and complexity.

Next, we'll take our existing product API and introduce a breaking change, which will require us to implement a versioning strategy. By evolving our API, we will ensure that clients using earlier versions remain unaffected while enabling new functionality for those adopting the latest version. This process highlights the importance of backward compatibility and minimizing disruption for users, even as the API continues to evolve.

Finally, we will explore the best practices for managing API evolution over time. We will discuss how to gracefully deprecate old versions, effectively communicate changes with clients, and maintain multiple versions of an API without creating unnecessary complexity. By the end of this chapter, you will have a solid understanding of how to implement versioning strategies that support long-term growth and stability for your APIs.

In this chapter, we will be covering the following topics:

- Versioning strategies
- Implementing versioning in the product API
- Managing API evolution

Technical requirements

In this chapter, we will evolve our product API. To be able to follow along and use the code examples as they are printed in the book, you should have the product API code that was created in *Chapter 2*.

You can access the code for this chapter on GitHub at `https://github.com/PacktPublishing/Mastering-RESTful-Web-Services-with-Java/tree/main/chapter5`.

Versioning strategies

Versioning refers to the practice of managing changes in an application over time by assigning distinct version numbers to different stages of the API's life cycle. It ensures that updates to an API—such as new features or breaking changes—can be introduced without disrupting existing clients that rely on earlier versions.

Versioning became critical for REST APIs as the architecture of software systems shifted from monolithic to distributed systems. In monolithic architectures, where the consumer and provider of a service typically reside within the same application, there is direct control over both ends of the communication. When changes are made, they are applied uniformly across the entire system, ensuring that the consumer is always in sync with the provider. Hence, the versioning of interfaces within a monolithic application is not necessary because the consumers are tightly coupled with the providers, and updates are deployed simultaneously without causing compatibility issues.

However, with the rise of distributed systems and the adoption of REST APIs, which serve multiple independent clients (such as web browsers, mobile apps, and third-party services), the situation changed dramatically. REST APIs expose services to the outside world, where different clients may use different versions of the API simultaneously. This introduces a challenge—how do you update the API to introduce new features or fix issues without breaking functionality for existing clients? Versioning solves this by allowing multiple versions of the API to coexist, ensuring backward compatibility, and enabling the API to evolve over time while still supporting older clients.

Let's see how we can apply versioning strategies to solve this problem.

Before exploring versioning strategies, it's essential to distinguish between backward-compatible changes and breaking changes. Understanding this distinction is crucial because, for breaking changes, updating the API version is necessary to ensure existing clients can continue to use the API smoothly.

Breaking changes are modifications that disrupt existing client interactions with the API. These changes typically necessitate a new API version to prevent breaking current integrations. These are some examples of breaking changes:

- **Removing an endpoint**: Deleting an existing API endpoint that clients rely on
- **Changing response data structure**: Altering the format of response data, such as renaming or removing fields in the JSON response
- **Modifying required parameters**: Changing required parameters for an endpoint, which could cause existing requests to fail
- **Altering authentication methods**: Changing or removing existing authentication mechanisms or security protocols
- **Changing endpoint behavior**: Modifying the behavior of an endpoint so that it no longer performs as it did previously

In contrast, **backward-compatible changes** enhance or extend the API without disrupting existing clients. Here are some examples:

- **Adding new optional fields**: Introducing new fields to the response payload that are not required for existing clients
- **Adding new endpoints**: Creating new API endpoints that provide additional functionality without affecting existing ones
- **Extending response data**: Adding extra information to the response data that does not alter the existing structure

- **Adding new query parameters**: Introducing optional query parameters to existing endpoints that offer additional filtering or functionality
- **Enhancing documentation**: Improving or expanding API documentation to provide more details without altering the API itself

Now that we understand the need for using versioning for breaking changes, let's explore various versioning strategies, along with their advantages and disadvantages. The most common strategies include the following:

- URL path versioning
- Query parameter versioning
- HTTP header versioning
- Content negotiation

Let's go deep into each versioning strategy and understand when to use one instead of the other.

URL path versioning

URL path versioning is one of the most common and straightforward strategies for versioning REST APIs. This strategy is used by many companies, such as X, Google, GitLab, and others. In this strategy, the version number is included directly within the API's URL path, such as `https://api.product.com/v1/products`.

This approach makes it easy for both developers and clients to identify and access specific versions of an API without any ambiguity. By embedding the version number in the URL path, API consumers have a clear and explicit way of requesting the desired version, ensuring that any integration remains stable even as the API evolves. Additionally, the server can easily route incoming requests to the correct version of the code base, simplifying maintenance and deployment.

URL path versioning provides several clear advantages for API management. One of its primary benefits is that version numbers are directly visible in the URI, making it immediately clear which version of the API is being accessed. Additionally, URL versioning integrates well with HTTP caching mechanisms, as different versions of the API are treated as distinct resources, allowing for the effective and independent caching of each version. It also facilitates bookmarking and sharing, enabling users to save and reference specific versions of endpoints easily, ensuring consistency in interactions.

However, URL versioning also presents certain drawbacks. Over time, maintaining multiple versions can lead to cluttered URIs, especially if multiple versions of the API are maintained simultaneously. Moreover, once a version is embedded in the URL path, it becomes part of the API's public contract. Furthermore, the overhead associated with managing numerous URLs can increase complexity in both development and documentation processes.

When implementing a new feature, breaking changes typically affect only a few specific endpoints, not the entire API. If you update the version number for endpoints that haven't changed, it can create unnecessary confusion for both clients and maintainers. This approach makes it harder to understand how different versions of various endpoints should be used together, potentially complicating the integration process and hindering overall API clarity.

Using URL paths for versioning is well-suited for many scenarios, particularly when clear and explicit versioning is crucial. It is an excellent choice for APIs that are stable, with minimal expected changes that could introduce breaking modifications. URL path versioning provides a straightforward approach to managing different versions, making it easier to track and maintain the API as it evolves.

Query parameter versioning

Query parameter versioning involves specifying the API version as a query parameter in the URL, such as `https://api.product.com/products?version=1`. This approach is commonly recommended in Microsoft's guidelines. This strategy makes it easier to implement and introduce new versions without modifying URL paths. This can be particularly useful when you want to maintain a consistent endpoint path structure while still providing version-specific functionality. It also allows versioning without creating a proliferation of URL paths.

However, it has some drawbacks. Clients must include the version parameter with every request, which adds an extra step in request construction. Additionally, the API server must handle versioning logic based on query parameters, potentially complicating implementation and increasing processing overhead. Furthermore, the version information is less visible in the URL, making it harder to quickly identify which version is being accessed. This lack of clarity can cause confusion, especially when the versioning affects critical parts of the API's functionality.

Another concern is how query parameter versioning interacts with caching. Since the version is embedded in the query string rather than the URL path, caching mechanisms may not handle it as effectively. This can lead to complications in how different versions of the API are cached, potentially impacting performance and efficiency for clients who rely on caching to improve response times.

HTTP header versioning

HTTP header versioning involves specifying the API version in the HTTP headers of a request. This approach allows clients to indicate which version of the API they wish to use by including a custom header, such as `X-API-Version`, in their requests. GitHub is an example of an application that uses the HTTP header:

```
curl --header "X-GitHub-Api-Version:2022-11-28" https://api.github.com/zen
```

This strategy keeps the URL clean and avoids cluttering it with versioning information. It also provides a flexible and less intrusive way to manage versions, as it does not alter the endpoint paths or query strings. Additionally, HTTP header versioning can facilitate smoother transitions between versions since the versioning information is separated from the URL being the resource identifier, reducing the risk associated with breaking changes for clients.

However, HTTP header versioning presents some challenges. Unlike URL versioning, the versioning information is not immediately visible in the URL, which can make debugging and documentation more complex. This lack of visibility can hinder developers and clients when troubleshooting issues or understanding API versions.

Additionally, custom headers can lead to inconsistent implementation practices across different APIs. When APIs use custom headers, it becomes challenging to establish uniform standards, which can confuse developers and users alike. For example, one API might use `X-Api-Version` while another might use `X-Version`, leading to ambiguity and potential integration issues. The use of the `X-` prefix for custom headers has been deprecated since the introduction of RFC 6648 in 2012. As an alternative, the content negotiation strategy leverages existing headers, avoiding the need for custom headers, and ensuring better adherence to standard practices.

Content negotiation

Content negotiation is one of the techniques used to handle versioning in RESTful APIs, allowing clients to specify the API version via HTTP headers rather than through the URL path. The idea is to let the client request a particular version of the API by sending a specific `Accept` header. For example, a client might request version 1 of the API in JSON format by sending the following request:

```
curl -X GET https://api.myproduct.com/resource \ -H "Accept: application/
vnd.myapi.v1+json"
```

This technique of API versioning via content negotiation is closely related to HTTP header versioning, as both strategies rely on HTTP headers to specify the version. The key difference is that content negotiation avoids creating custom HTTP headers, adhering to RFC 6648, which discourages custom headers. Instead, this method uses standard headers such as `Accept` and `Content-Type` to specify both the format and version of the response, promoting a cleaner and more standardized approach.

This strategy is widely adopted by large organizations such as Adidas and Mambu. The content negotiation strategy is aligned with the REST standard in that it follows the principle of content negotiation, where the client and server agree on the representation format and version via headers, keeping the interface flexible and compliant with RESTful best practices.

Since our API is already implemented, we will adopt a content negotiation strategy to enable seamless evolution, allowing new features to be introduced without disrupting existing clients or causing breaking changes.

Implementing versioning in the product API

As our product API is already in use by other applications, it's essential to maintain stability while continuing to evolve the API to meet new requirements. One of the challenges in API development is balancing the need for enhancements while not disrupting existing consumers. In this section, we will explore how to implement versioning in the product API to handle changes without causing breaking issues for clients.

The product API currently returns a complete list of products without pagination. To improve performance and scalability, we've decided to introduce pagination to the product list response. This modification will change the current behavior, potentially disrupting users who rely on the existing output. To prevent any impact on their workflows, we will implement a versioning strategy. This approach will allow us to support the new paginated format for future clients while maintaining backward compatibility for existing consumers, ensuring that the API can evolve without interrupting service.

As mentioned previously, we will use the content negotiation strategy to introduce pagination into our product API without breaking the existing functionality. This strategy will allow us to evolve the API while letting clients specify the version they prefer through request headers. By utilizing content negotiation, we can offer both the original product list format and the new paginated version, ensuring backward compatibility and giving clients flexibility in how they consume the API. This method ensures a smooth transition for existing users while accommodating new requirements.

Updating our product API

Let's enhance our API by adding new functionality to our list endpoint while maintaining backward compatibility.

In our current API, we have the following code:

```
@GetMapping
@Override
public ResponseEntity<List<ProductOutput>> getProducts() {
    final var products = productsQueryUseCase.getAllProducts()
        .stream()
        .map(productMapper::toProductOutput)
        .toList();
    return ResponseEntity.status(HttpStatus.OK)
        .body(products);
}
```

Let's create another method to handle version 2 of our API, which will include support for pagination. To achieve this, we'll add two `RequestParam` annotations: one to specify the current page and another to define the limit of the number of results per page.

Design strategy to prevent backward-incompatible changes

Adding pagination in our example cannot be done without a breaking change because the response is a JSON array of the products, so there is no place for pagination information that is common for all products returned (the total number of pages). A trick to prevent this situation is, instead of using a JSON array at the top level, to wrap the list (JSON array) in a JSON object even if it would have just one attribute. This minimal additional overhead would allow us to evolve the API without introducing a breaking change. We would just add optional pagination request parameters, and the pagination information could be added to the object at the root level of the response.

You can see these `RequestParam` annotations in the following code:

```
@GetMapping(produces = "application/vnd.packt-v2+json")
@Override
public ResponseEntity<PaginatedProducts> getProductsV2(@RequestParam(
    value = "page", defaultValue = "0") Integer page, @RequestParam(
    value = "limit", defaultValue = "10") Integer limit) {
        final var products = productsQueryUseCase.getAllProducts(
            PageRequest.of(page, limit));
    int totalPages = products.getTotalPages();
    List<ProductOutput> output = products.stream()
        .map(productMapper::toProductOutput)
        .toList();
    return ResponseEntity.status(HttpStatus.OK)
        .body(new PaginatedProducts(totalPages, output));
}
```

By adding `produces = "application/vnd.packt-v2+json"`, we enable Spring to route any request with the `Accept` header set to `application/vnd.packt-v2+json` to the appropriate endpoint. This allows clients to receive the response tailored to their version by specifying the `Accept` header in their API requests.

We have also updated the API response to return a `PaginatedProducts` class, which now includes the `totalPages` attribute, as we can see in the following code. This addition is crucial for the client to determine whether more resources are available or not:

```
public record PaginatedProducts(@Schema(name = "totalPages",
    example = "10") Integer totalPages, List<ProductOutput> products) {}
```

We opted to modify the response body to include this information, though other strategies could have been used to achieve the same goal, as we will explore in *Chapter 6*.

With the addition of this new endpoint (`getProductsV2`), we need to update the `ProductsQueryUseCase` class to include the necessary implementation code. In the following example, the `getAllProducts` method now accepts a `Pageable` object from Spring, which handles pagination details. The method returns a `Page` object containing the list of products along with metadata such as the total number of pages:

```
public interface ProductsQueryUseCase {
    Product getProductById(String productId);

    Page<? extends Product> getAllProducts(Pageable pageRequest);
}
```

Additionally, adding this endpoint requires us to change the `ProductsApi` interface and add the definition and documentation of our endpoint, as we can see in the following code block:

```
public interface ProductsApi {
....
@Operation(
        operationId = "getProducts_v2",
        summary = "Retrieve all products",
        responses = {
            @ApiResponse(responseCode = "200",
                description = "A list of products", content = {
                    @Content(mediaType = "application/json",
                        array = @ArraySchema(schema = @Schema(
                            implementation = PaginatedProducts.class)))
                })
        }
)
ResponseEntity<PaginatedProducts> getProductsV2(
     @Parameter(name = "page", description = "Number of current page",
        required = false, in = ParameterIn.QUERY, example = "0")
        Integer page,
        @Parameter(name = "limit",
            description = "Size of elements per page", required = false,
            in = ParameterIn.QUERY, example = "10") Integer limit);
....
}
```

Now that we have the documentation code, let's see what our Swagger documentation looks like:

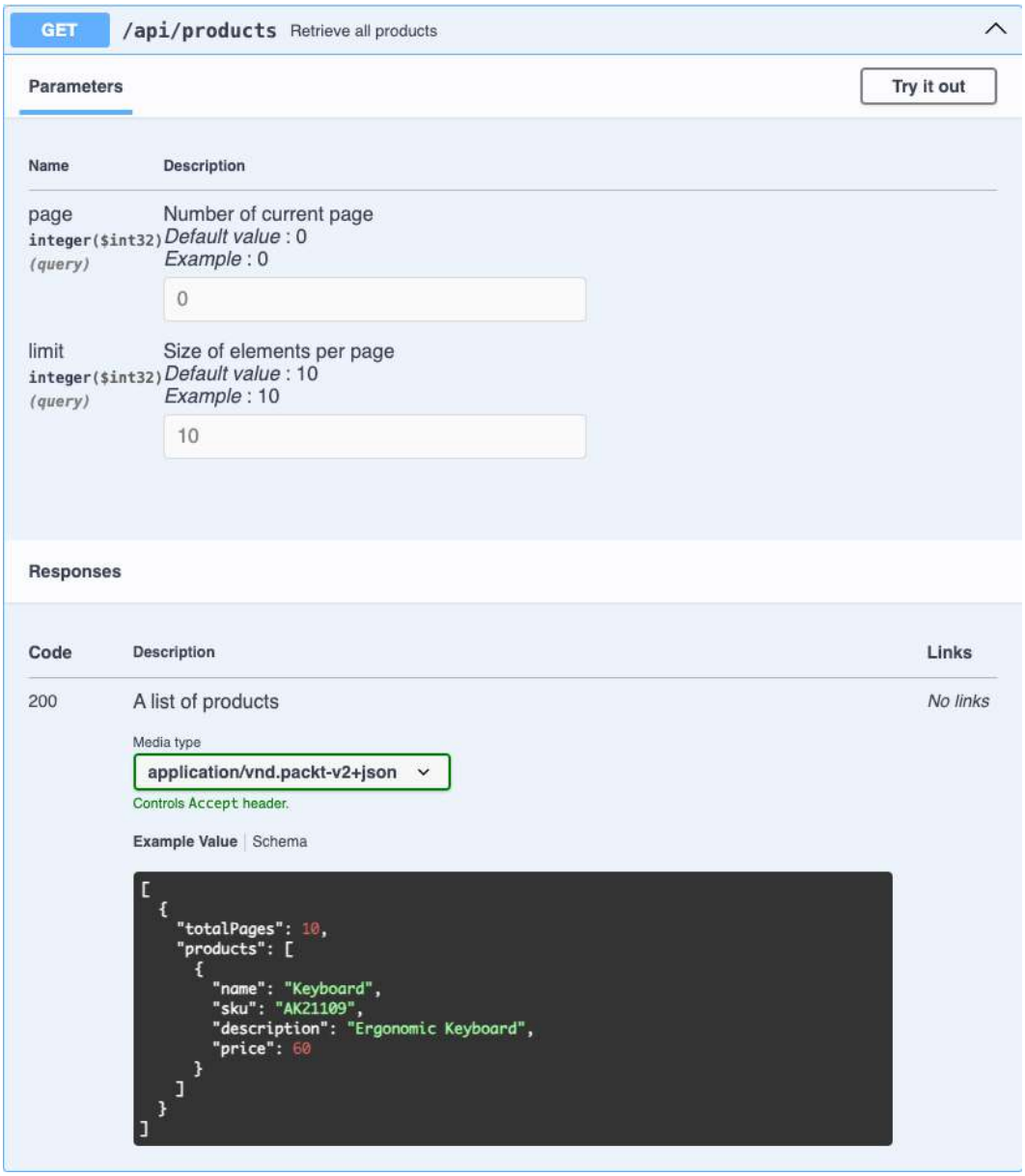


Figure 5.1 – Swagger UI

Figure 5.1 illustrates our API, showcasing the `/api/products` endpoint with two parameters and a new response format when the media type is `application/vnd.packt-v2+json`.

By switching the media type to `application/json`, as we can see in *Figure 5.2*, the API will change the response to its previous behavior:

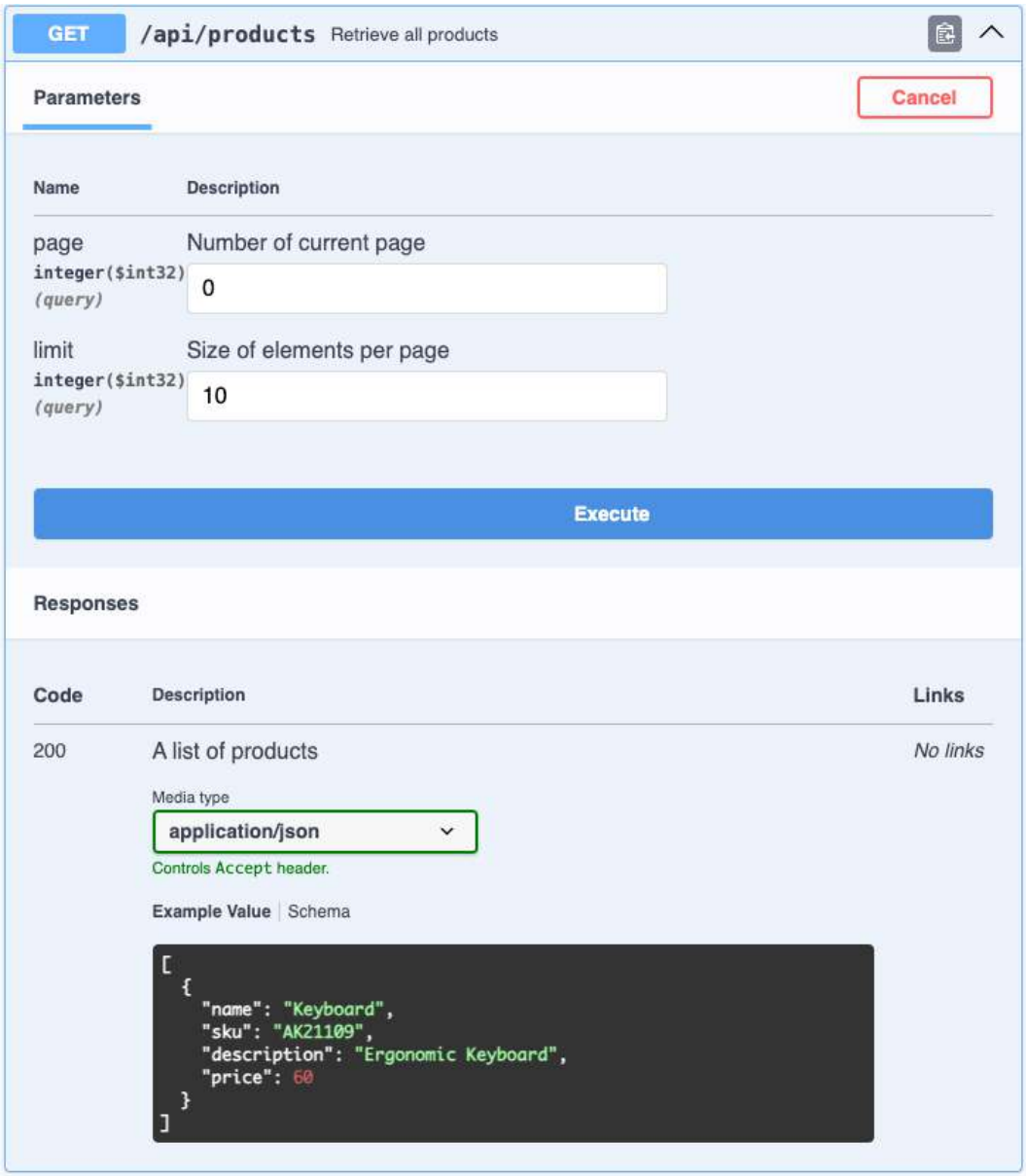


Figure 5.2 – Swagger UI

As shown, the API maintains backward compatibility by providing the original output for older clients. However, our documentation still enforces parameters, even though they aren't being used. Unfortunately, OpenAPI 3.x does not support varying parameters based on content type.

Note

The upcoming OpenAPI 4.x, expected by the end of 2024, will address the limitation of supporting different query parameters according to the content type. Since we are currently using 3.x, this feature is not supported in our implementation.

With the new version implemented, it's time to validate our changes and ensure everything functions as expected.

Testing and validating our product API

Now that our product API is equipped to support multiple versions, it's time to test and validate the functionality. We want to ensure that each version of the API behaves as expected, delivering the correct responses without breaking any existing functionality. To achieve this, we can use tools such as Swagger for the graphical interface, or we can interact with the API directly via the command line using `curl` commands. By testing both the original and updated versions, we can confirm that the API handles different versions correctly.

To retrieve the list of products using the first version of the API, you can execute the following `curl` command:

```
curl -X 'GET' 'http://localhost:8080/api/products' -H 'accept:
application/json'
```

This command sends a request to the API without specifying the version of the product API. The expected result is a list of products in the original format, without pagination:

```
[
  {
    "name": "Keyboard",
    "sku": "AK21109",
    "description": "Ergonomic Keyboard",
    "price": 60
  },
  {
```

```
    "name": "Keyboard 08",
    "sku": "AK21108",
    "description": "Ergonomic Keyboard",
    "price": 60
  },
  {
    "name": "Keyboard 07",
    "sku": "AK21107",
    "description": "Ergonomic Keyboard",
    "price": 60
  }
]
```

Now, let's try the same request passing the page and the limit, using the following `curl` command:

```
curl -X 'GET' 'http://localhost:8080/api/products?page=0&limit=2' -H
'accept: application/json'
```

The output will remain the same as before, even though we've specified a limit of two items. This occurs because the first version of our API wasn't designed to handle pagination query parameters. As a result, the API ignores these parameters and the full list of products is returned, adhering to the behavior defined in version 1.

The code in our first version will return all available elements. We implemented the second version by passing Spring's `Pageable` object. To adhere to the **Don't Repeat Yourself (DRY)** principle, we must update the code from version 1, where all items are returned by default, and we have to set the size to the maximum possible value. This ensures that the first version continues to function as expected without introducing pagination:

```
@GetMapping
@Override
public ResponseEntity<List<ProductOutput>> getProducts() {
    final var products = productsQueryUseCase.getAllProducts(
        PageRequest.ofSize(Integer.MAX_VALUE))
            .stream()
            .map(productMapper::toProductOutput)
            .toList();
    return ResponseEntity.status(HttpStatus.OK)
        .body(products);
}
```

Now that we've confirmed that our API functions as expected for the previous version, let's move on to testing the latest version. To do this, we will use the `curl` command shown here to verify the behavior of the updated API:

```
curl -X 'GET' 'http://localhost:8080/api/products?page=0&limit=2' -H
'accept: application/vnd.packt-v2+json'
```

The key difference in this command compared to the previous one is the `Accept` header, where we specify `application/vnd.packt-v2+json`. By doing so, the Spring Framework will route the request to the version 2 endpoint of our API. The expected output is a paginated list of products, reflecting the new behavior introduced in version 2:

```
{
  "totalPages": 2,
  "products": [
    {
      "name": "Keyboard",
      "sku": "AK21109",
      "description": "Ergonomic Keyboard",
      "price": 60
    },
    {
      "name": "Keyboard 08",
      "sku": "AK21108",
      "description": "Ergonomic Keyboard",
      "price": 60
    }
  ]
}
```

As we can see from the output, the response contains only two items, and the `totalPages` field reflects the pagination, adjusting the result according to the limit parameter we provided. This demonstrates that the latest version handles pagination as expected.

Now that we have confirmed the expected behavior for both version 1 and version 2, let us explore what happens when we attempt to access a non-existent version (let's call it version 3).

To do this, we can execute the following `curl` command and observe the result:

```
curl -X 'GET' 'http://localhost:8080/api/products?page=0&limit=2' -H
'accept: application/vnd.packt-v3+json'
```

After executing it, the output will be the same as that obtained with version 1:

```
[
  {
    "name": "Keyboard",
    "sku": "AK21109",
    "description": "Ergonomic Keyboard",
    "price": 60
  },
  {
    "name": "Keyboard 08",
    "sku": "AK21108",
    "description": "Ergonomic Keyboard",
    "price": 60
  },
  {
    "name": "Keyboard 07",
    "sku": "AK21107",
    "description": "Ergonomic Keyboard",
    "price": 60
  }
]
```

This behavior might not be ideal, as it could mislead the client into thinking version 3 exists. This happens because, by default, the Spring Framework allows two media types: `application/json` and `application/*+json`. As a result, any request with the `application/*+json` media type will be processed by the default method unless explicitly mapped to a different version, as in our second version. Therefore, without specific handling, requests for version 3 may still be routed to the default version, giving an unintended response.

To solve this problem, we can update our endpoint from version 1 and force it to accept only `application/json`, as we can see in the following code:

```
@GetMapping(produces = MediaType.APPLICATION_JSON_VALUE)
@Override
```

```
public ResponseEntity<List<ProductOutput>> getProducts() {
    final var products = productsQueryUseCase.getAllProducts(
        PageRequest.ofSize(Integer.MAX_VALUE))
            .stream()
            .map(productMapper::toProductOutput)
            .toList();
    return ResponseEntity.status(HttpStatus.OK)
            .body(products);
}
```

After applying this solution, when we execute the `curl` command with the `application/vnd.packt-v3+json` media type, we will encounter an error like so:

```
{
  "type": "about:blank",
  "title": "Not Acceptable",
  "status": 406,
  "detail": "Acceptable representations: [application/json,
    application/vnd.packt-v2+json].",
  "instance": "/api/products"
}
```

This error response helps the client easily identify the available versions of the endpoint. By returning a clear error when an unsupported version is requested, we provide immediate feedback, guiding the client toward the correct version and improving overall API usability.

Now that we've tested our solution and confirmed that it maintains the expected behavior from version 1 while introducing the new functionality in version 2, along with returning a `406 Not Acceptable` error for incorrect versions, we can shift our focus to managing these versions effectively as the API continues to evolve. Let's explore how to handle versioning as our API grows and adapts to future needs.

Managing API evolution

Managing the evolution of an API is a critical aspect of long-term API maintenance. As applications grow and customer needs change, APIs must adapt and introduce new features while ensuring the stability and continuity of existing services. Effective management of API evolution involves planning for changes, communicating clearly with API consumers, and implementing strategies that avoid breaking existing functionality while introducing improvements in a structured and predictable manner.

To manage API evolution successfully, we should consider the key practices described in the following subsections.

Avoiding introducing breaking changes

As we discussed earlier, the best approach to evolving your API is to ensure that new changes don't disrupt existing clients by maintaining backward compatibility. This minimizes potential service interruptions and provides a smooth transition for users.

To achieve this, it's essential to clearly differentiate between breaking and non-breaking changes. When introducing breaking changes, applying versioning is crucial to preserve the integrity of the current functionality while allowing new features to coexist.

Defining and applying a versioning strategy across your APIs

When a breaking change is necessary, applying a versioning strategy is essential. There are many ways to version your API so, as discussed earlier, maintaining consistency in versioning is crucial for ensuring that clients can easily understand and interact with your API. A clear and predictable versioning strategy enhances the user experience, making it simpler for clients to navigate and integrate with the API.

Having a well-defined REST guideline that outlines how to implement a versioning strategy consistently across all APIs, as mentioned in *Chapter 1*, is key, and establishing a unique versioning strategy provides numerous benefits, including improved usability and transparency for API consumers.

Updating and informing clients about new versions and deadlines

With multiple API versions in play, it is essential to establish clear migration guidelines and deprecation timelines. Providing detailed information on how to transition between versions helps clients understand the process and prepare for changes. This transparency is crucial for a smooth migration and the minimization of potential disruptions.

Maintaining an up-to-date changelog is equally important. It offers clients a comprehensive view of recent updates, bug fixes, and new features, ensuring they are informed of any modifications. A well-documented changelog supports clients in adapting to changes effectively and fosters a positive user experience.

Semantic versioning

Semantic versioning (`https://semver.org/`) is a widely adopted system that clearly communicates the scope and impact of changes in an API. It uses a three-part format: `MAJOR.MINOR.PATCH`. For example, a major version change (e.g., from 1.x to 2.0.0) alerts clients that breaking changes have been introduced, giving them time to adapt. In contrast, minor and patch updates (e.g., 1.1.0 or 1.0.1) reassure clients that improvements or bug fixes have been made without affecting compatibility. This structured approach allows developers to quickly assess the nature of an update, minimizing the risk of breaking existing functionality.

By implementing semantic versioning, we can offer clear guidelines and comprehensive changelogs to keep clients informed about updates. This structured approach helps clients understand the significance of each version change, making it easier for them to prepare for major updates that may require action.

For minor or patch updates, clients are notified of new features or bug fixes without immediate pressure to upgrade. This balance ensures that clients stay informed and can choose to update at their convenience, maintaining a smooth and flexible user experience.

Marking old endpoints as deprecated

As part of the API evolution process, it's important to clearly mark endpoints scheduled for deprecation to give clients ample time to transition to new versions. Deprecating old endpoints helps signal to clients that updates are required.

OpenAPI provides a way to mark any endpoint as deprecated, which is reflected in the Swagger UI. This can be easily done using the following code:

```
@Operation(
    deprecated = true,
    ....
)
```

When we add the `deprecated` attribute and view it in the Swagger UI, the output will look like *Figure 5.3*. However, we chose not to use this attribute here because OpenAPI 3.x currently doesn't support deprecated endpoints with versioning based on content negotiation.

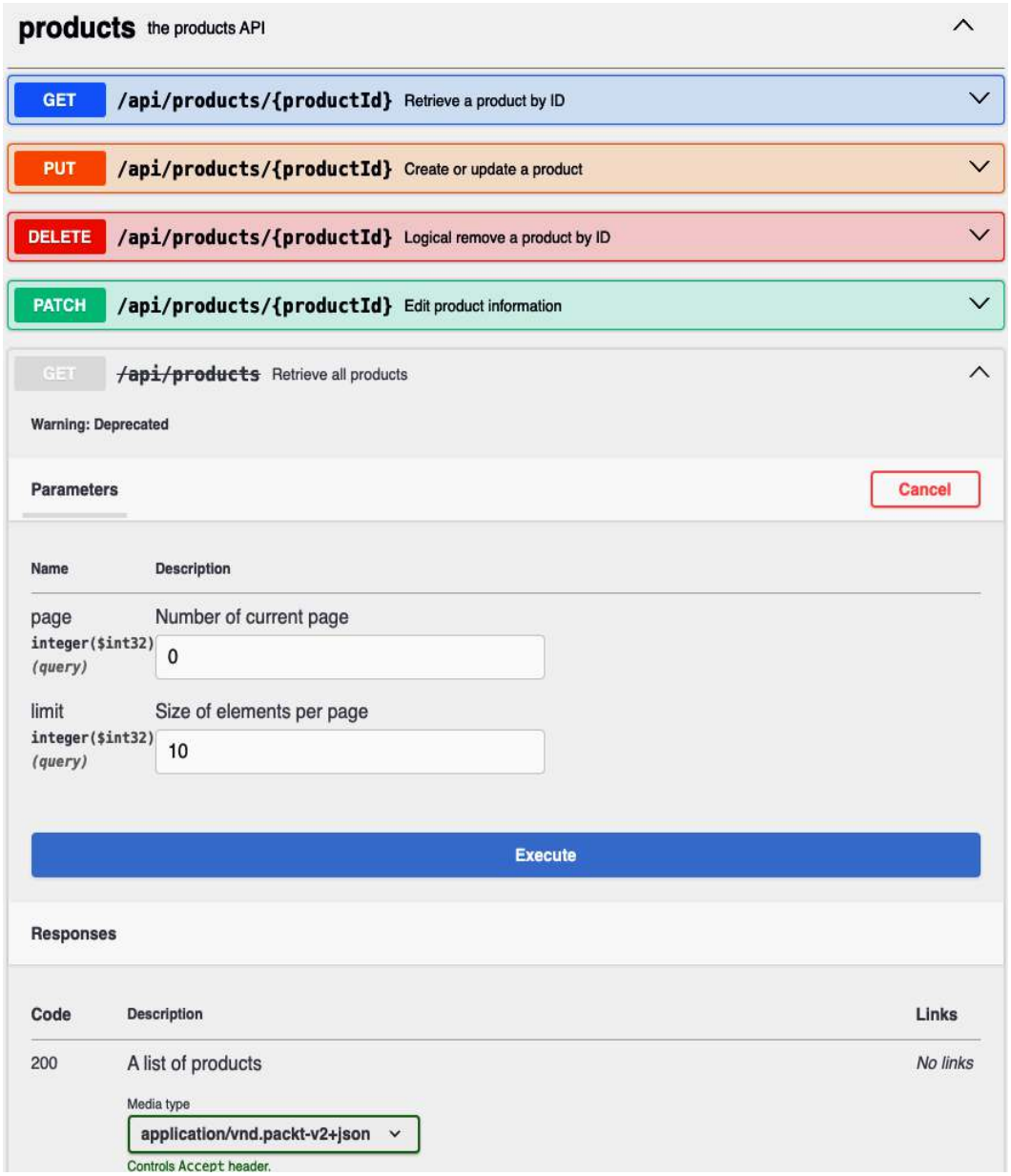


Figure 5.3 – Swagger UI: Monitoring the usage of deprecated endpoints

Monitoring the usage of deprecated endpoints is essential for ensuring a smooth transition when phasing out older versions of an API. By tracking requests to these endpoints, you can determine how many clients are still using outdated functionality and evaluate their progress in migrating to newer versions. This helps you make informed decisions about when to officially remove old versions, avoiding unexpected disruptions to your clients' operations. Additionally, it provides insights into which clients might need extra support or communication during the transition.

Monitoring tools, such as Grafana, New Relic, Datadog, and Splunk, not only facilitate a smooth deprecation process but also help confirm the adoption of newer API versions. These insights enable you to set practical timelines for deprecating endpoints, ensuring their removal does not disrupt clients and enhancing trust and reliability in your API's development.

Removing old endpoints

Once you have established a deadline, communicated with your clients, and provided clear instructions for upgrading to the latest version, the next step is to remove the deprecated endpoints. But, before finalizing the removal, it is crucial to verify that clients have successfully migrated to the newest version. This process involves not only deleting the old code but also ensuring that all references and dependencies related to these endpoints are thoroughly cleaned up.

By applying these principles, you can manage API evolution smoothly and effectively, minimizing the risk of disrupting existing clients while equipping them with the necessary tools and information to adapt seamlessly.

Establishing a well-defined versioning strategy, using semantic versioning, and properly deprecating old endpoints allows your API to grow with minimal disruption. This approach ensures that your clients have sufficient time to transition, while you continue to improve and expand your API's capabilities. Ultimately, it promotes long-term stability and improves the developer experience by allowing your API to evolve sustainably over time.

Summary

In this chapter, we explored various versioning strategies for REST APIs and learned how to evolve our API while maintaining compatibility and ensuring a smooth transition to newer versions. We also applied content negotiation versioning to our current API, specifically focusing on implementing pagination.

In the next chapter, we will delve into more advanced topics, including enhanced pagination techniques, filtering, file uploads, and additional features to further refine your API.

6

Advanced API Concepts and Implementations

Creating robust and high-performance RESTful APIs involves more than just setting up endpoints and handling CRUD operations. As applications grow and user demands increase, incorporating advanced strategies that keep your APIs efficient, reliable, and resilient under various conditions becomes essential. This chapter explores these strategies, focusing on data-handling techniques such as pagination, filtering, and file uploads and downloads, as well as resilience mechanisms crucial for modern API development.

On the data management front, we will examine key practices such as pagination, filtering, and the efficient uploading and downloading of files through REST APIs. These methods are designed to boost the responsiveness and scalability of your APIs, ensuring they can seamlessly manage higher loads and more complex data interactions. Some of these topics are related to performance, but we will cover performance optimization in greater detail in *Chapter 10*.

The chapter will explore critical resilience mechanisms, including timeouts, retry strategies, rate limiting, throttling, idempotency keys, circuit breakers, and bulkheads. These approaches help safeguard your APIs against failures, manage traffic effectively, and maintain stability even under unpredictable or adverse conditions.

From a developer's perspective, anticipating how systems will behave in real-world scenarios can be challenging. During development, we often operate in ideal settings: running the API locally, being the sole user, and conducting tests with minimal variables that might cause failures. This controlled environment can obscure the complexities and potential issues that emerge in production, where the API must support multiple users, handle varying loads, and respond to unexpected events. By incorporating the advanced concepts and implementations discussed in this chapter, you will be better equipped to build APIs that perform reliably and efficiently in any environment.

We are going to cover the following topics:

- Data handling
- Resilience

Technical requirements

In this chapter, you will find several different approaches that will help you to improve your API depending on your requirements. For this chapter, we recommend you catch up with the code from the previous chapter; some small enhancements for this chapter are available at the following link: `https://github.com/PacktPublishing/Mastering-RESTful-Web-Services-with-Java/tree/main/chapter6`. We use the `curl` command-line tool as a client to test the APIs.

Data handling

Data handling is the cornerstone of building robust and user-friendly APIs. In this section, we will explore advanced data-handling techniques that address common challenges faced in real-world API development.

Ensuring that an API delivers reliable performance is crucial for several interconnected reasons, including enhancing user experience, enabling system scalability, optimizing resource efficiency, and influencing the success of the business behind API development.

A high-performing API must respond quickly and communicate clearly with clients. This means returning appropriate HTTP status codes (such as 200 for success or 404 for not found) along with descriptive messages that explain what happened. Low latency is crucial because users expect fast responses, and slow APIs can create system bottlenecks that frustrate users.

When APIs take too long to respond, they cause delays throughout the entire application. Modern systems often need real-time data updates, so APIs must handle frequent requests efficiently while maintaining consistent response times. Clear communication is equally important—when errors occur, clients need informative messages that explain the problem and suggest solutions.

Effective APIs balance speed with clarity. They process requests quickly while providing meaningful feedback through proper status codes and helpful messages. This approach ensures both end users and developers can work with the API effectively, reducing confusion and debugging time.

Optimizing response speed and message quality helps APIs handle increased traffic without compromising performance or raising operational costs. These improvements support both user satisfaction and business growth.

In the following sections, we will explore data-handling techniques that improve API performance. We will start with pagination for managing large datasets, then cover filtering and efficient file operations through REST APIs.

Pagination

Retrieving a large number of records from one or multiple database tables and converting them into a data structure compatible with your language/framework, such as in Java with Spring Boot, consumes resources, including memory and CPU processing power. This process continues until the data is transformed into a standard format such as JSON for communication and is ready to be sent to the client. Additionally, even after the data is prepared, resources are still used to transmit it over the network to the client.

Often, it is unnecessary to send all the data represented by a resource in a single, large payload to the client. Doing so consumes more resources, degrades the system performance, and causes a poor user experience. These may be accompanied by the increased costs associated with cloud computing, which are billed based on usage. Processing and transferring larger amounts of data also takes more time. Instead, the data can be divided into smaller chunks, which require less time and fewer resources, thereby making the API more reliable and cost-effective to maintain. This approach is known as **pagination**, which enhances API responsiveness and conserves resources.

Consider the following example: our API contains 100,000 products, and we have a **single-page application** (**SPA**) that consumes this API to display products in a data table on the **Products** page. Imagine that every time a user opens this page, the API is called to access the database, load these records into memory, convert them to JSON, and then transfer this large amount of data via an internet request each time a user accesses the page. Now, imagine that this system is used by numerous users daily. Without pagination, each user action would trigger resource-intensive processes, leading to increased costs, slower response times, and a diminished user experience. By implementing pagination, the API can deliver smaller, more manageable datasets, improving performance and scalability while reducing operational expenses. All of that can be done without limiting the user interface view of the application, because the user will never see all the products displayed at the same time.

Different pagination approaches

The following are some common approaches to REST API pagination:

- Offset-based pagination
- Page-based pagination
- Cursor-based pagination
- Keyset pagination

Let's take a look at each of them

Offset-based pagination

Offset-based pagination is a straightforward and widely adopted technique. It works by defining an offset (the starting position) and a limit (the number of records to retrieve per page) to navigate through the dataset. This is the approach that we can find in the APIs of several different companies, such as OpenWeatherMap, Stripe, Adidas, and Mailchimp.

The following are the advantages of offset-based pagination:

- **Simplicity**: The method is straightforward to implement and easy to understand
- **Flexibility**: By adjusting the offset, you can directly jump to any position that you may want

The following are the disadvantages of offset-based pagination:

- **Performance issues**: As the offset increases, query performance declines because the database must scan and skip over a growing number of records.
- **Data consistency**: Data changes such as inserts or deletes between requests can lead to inconsistent results, causing duplicate or missing records across pages.

Clients can manipulate the business logic, which does not guarantee a single experience for all clients. It can be considered a benefit, though, if the API is meant to be very flexible.

Most likely, this will be passed to a database query such as SQL, which, without proper input sanitization and parameterization, can cause SQL injection.

The following is an example of an API call:

```
GET /api/products?offset=50&limit=25
GET https://<dc>.api.mailchimp.com/3.0/lists/{list_id}/
members?count=100&offset=200
```

Page-based pagination

Page-based pagination divides a dataset into pages, allowing clients to navigate the data by specifying a page number. This approach is simple and hence widely adopted by companies in the API market, for example, Salesforce and Microsoft.

To use it, you need to specify the page you want to retrieve (`page`) and the number of records per page (`page_size`).

The following are the advantages of page-based pagination:

- **User-friendliness**: Intuitive for users to navigate to specific pages
- **Simplicity**: Easy to implement and understand

The following are the disadvantages of page-based pagination:

- **Performance issues**: Like offset-based pagination, large page numbers can cause performance degradation.
- **Less flexibility**: The returned list of records must start at a page boundary. This may not be convenient for user interfaces using scrollbars instead of the classical pagination.

The following is an example of an API call:

```
GET https://api.github.com/repos/{owner}/{repo}/issues?page=2&per_page=30
GET https://yoursubdomain.zendesk.com/api/v2/tickets.json?page=5&per_
page=100
```

Page-based pagination for the `GET /products` endpoint of our example Product API was implemented while demonstrating API evolution in *Chapter 5*.

Cursor-based pagination

Cursor-based pagination uses a pointer (cursor) to keep track of the current position in the dataset. Instead of specifying an offset, clients use a cursor to request the next set of records and a limit that represents the number of records we want to return per chunk of data. That is the approach implemented by the X (formerly Twitter) API to paginate the data of tweets, followers, and other resources.

The following are the advantages of cursor-based pagination:

- **Performance efficiency**: Maintains reliable performance regardless of dataset size since it does not require skipping records
- **Data consistency**: Less susceptible to data inconsistencies between requests, ensuring more reliable results

The following are the disadvantages of cursor-based pagination:

- **Complexity**: The implementation is more complex when compared to offset-based pagination
- **Less flexibility**: It is suited for sequential navigation and may not be a good option if you need to jump to arbitrary pages

The following is an example of an API call:

```
GET /api/products?cursor=eyJpZCI6NzUsIm5hbWUiOiJQcm9kdWN0IDc1In0=&limit=25
GET https://api.twitter.com/2/tweets?pagination_token=XYZ123&max_
results=20
```

Keyset pagination

Keyset pagination is a variation of cursor-based pagination; it leverages a unique key—typically a timestamp or a **universally unique identifier (UUID)**—to navigate through records. Unlike offset-based pagination, which relies on numerical offsets that can become increasingly inefficient with larger datasets, keyset pagination uses the unique key to mark the position in the dataset. This approach ensures faster query performance and more consistent response times, making it particularly well suited for applications that require real-time data access and scalability.

The following are the advantages of keyset pagination:

- **Performance**: Keyset pagination offers better performance for large datasets by eliminating the need for the database to count or skip over large numbers of records, as is necessary with offset-based pagination
- **Scalability**: As datasets grow, keyset pagination maintains consistent response times, making it a scalable solution for high-traffic applications
- **Reliability**: By relying on a unique key for navigation, keyset pagination reduces the risk of encountering missing or duplicate records, ensuring data integrity across paginated results

The following are the disadvantages of keyset pagination:

- **Sequential navigation only**: Keyset pagination requires sequential data access. Unlike offset-based pagination, which allows users to jump directly to any page by specifying an offset, keyset pagination requires navigating through records in a linear order. This limitation can be restrictive for applications where users need to access non-sequential pages or perform random access within the dataset.

- **Bookmark dependency**: To access a specific page, the client must retain the unique key (cursor) from the preceding pages. This dependency can complicate client-side logic, especially in scenarios where users might want to revisit or share specific pages without maintaining a history of cursors.

The following is an example of an API call:

```
GET https://www.reddit.com/r/{subreddit}/comments.json?limit=25&after=t3_
abcdef
GET https://api.linkedin.com/v2/
connections?q=cursor&start=0&count=25&cursor=urn:li:person:123456789
Authorization: Bearer YOUR_ACCESS_TOKEN
```

After selecting the pagination strategy to apply to all our endpoints and APIs, we should also return the information about the pagination. Providing information such as total items, pages, and the current page helps clients better navigate large datasets.

Different approaches to return pagination information

There are many ways to return pagination information to the client. The most common are as follows:

- Using response headers
- Including pagination information in the response body
- Using hypermedia (HATEOAS)

Using response headers

This strategy involves embedding pagination information directly within the HTTP headers, such as details about the total number of pages, the current page, and other relevant information. This approach is not commonly used in isolation, and it is often combined with other strategies to enhance API functionality. For instance, GitLab employs this method alongside hypermedia strategies to provide a more comprehensive user experience. We can combine filtering with pagination to retrieve specific data, allowing us to manage large datasets by breaking them into paginated sections:

```
HTTP/2 200 OK
status: 200 OK
X-Total-Count: 150
X-Total-Pages: 15
X-Current-Page: 3
X-Page-Size: 10
```

In the preceding example, we can see four custom headers to inform the client about how to navigate through the API effectively.

Including pagination information in the response body

In this approach, the API includes paginated data within the response body alongside relevant pagination metadata, such as total pages and the current page. For example, the Stripe API employs this strategy by returning both the data and a `has_more` flag that indicates whether additional results are available. This method offers clarity by directly integrating pagination details with the response, letting clients understand their data context.

Some APIs separate the response data from the metadata, which can also be effective. An example structure is shown here:

```
{
  "data": [
    {
      "id": 1,
      "name": "Item 1"
    },
    {
      "id": 2,
      "name": "Item 2"
    }
  ],
  "pagination": {
    "totalItems": 150,
    "totalPages": 15,
    "currentPage": 3,
    "pageSize": 10
  }
}
```

In this structure, the `data` field contains the actual results from the API, while the `pagination` section provides essential metadata about the paginated response. This approach enhances usability by clearly organizing both the data and its associated pagination information, making it easier for clients to navigate and process the results.

Using hypermedia (HATEOAS)

HATEOAS enhances API responses by embedding navigational links that guide clients on how to navigate and perform actions based on the server's response. As we discussed in *Chapter 1*, HATEOAS represents the fourth level of maturity in REST architecture, helping to decouple the client from the server. We will dive deep into this subject in the *HATEOAS* subsection later in this chapter.

This strategy empowers clients to interact with the API dynamically, without requiring prior knowledge of the structure or additional documentation. For example, APIs such as GitHub incorporate HATEOAS to make navigation through resources, such as paginated data, straightforward and intuitive. In the following code block, we can see an example of using this strategy to help the client move through the data:

```
{
  "_links": {
    "self": { "href": "/orders?offset=200&limit=20" },
    "prev": { "href": "/orders?offset=180&limit=20" },
    "next": { "href": "/orders?offset=220&limit=20" },
    "first": { "href": "/orders?limit=20" },
    "last": { "href": "/orders?offset=1000&limit=20" }
  },
  "totalCount": 1020,
  "_embedded": {
    "orders": [
      { ... },
      { ... },

      ...
    ]
  }
}
```

As shown in the preceding code, the `_links` attribute provides key navigational details, including information about the current request and links to the previous and next pages, as well as the first and last pages. With these comprehensive links, clients can effortlessly navigate through the API without needing to construct additional requests manually.

This structure streamlines the client's interaction with the data, embedding the necessary navigation information within each response. By supplying these links, the API enables developers to create more intuitive and seamless user experiences.

In API development, efficient data handling and robust pagination strategies are not merely supplementary features—they are essential points that underpin the efficacy, scalability, and overall user experience of your applications. We have outlined some different pagination approaches with their details, to ensure that your APIs deliver reliable and high-performing services. Regardless of your chosen approach, the key is maintaining consistency across all endpoints. Ensuring that every API follows the same pagination standard helps simplify integration for clients and promotes a more cohesive experience.

In the next section, we will explore how to filter data returned by the API by applying some strategies that allow us to combine pagination and filtering.

While pagination alone enables us to manage large datasets by breaking them into manageable pages, it may not fully address the client's needs when specific subsets of data are required. To improve data handling further, we can incorporate filtering, which will allow clients to retrieve only the data they need within each page. By combining pagination with filtering, we offer a more flexible and efficient approach to data retrieval.

Filtering

Filtering allows API clients to inform our API that we need some specific subset of the data on that resource, which means that it will not waste resources processing data that we are not interested in at that moment; we can concentrate exclusively on the data that we need. We can combine filtering with pagination to retrieve specific data, allowing us to manage large datasets by breaking them into paginated sections.

The following are some principles for filtering:

- **Statelessness**: The REST API should be stateless, meaning that every request from a client must contain all the necessary information that will give the server the capability to fulfill the request. Filtering is achieved by including filter parameters in the query string of the request URL, without the need to save this state on the API server side.
- **Consistency and predictability**: The filtering syntax should be consistent across all API endpoints. This ensures predictability across different endpoints and makes it easy for clients to understand the filtering by looking at the request. For example, if we are filtering by a first name parameter using `` `?firstName=Peter` `` in one endpoint, it should work similarly across other endpoints where such filtering is relevant.

- **Granularity and flexibility**: An effective filtering mechanism allows clients to precisely define their requirements while being adaptable enough to handle different scenarios. For instance, filters should accommodate different data types (such as strings, numbers, and dates) and a range of operators (such as equals, not equals, greater than, less than, in, etc.).

Different filtering approaches

Filtering can be implemented via several different approaches. It can be implemented using the following:

- **Basic field filtering**: This is the most straightforward approach, where you can pass the field name to the resource and the value that you want to filter by, which is usually done by passing these values as a query parameter on the URL:

```
GET /customers?firstName=Peter or GET /products?category=electronics
```

- **Multiple field filtering**: This is the same as the previous one, with the only difference being that we will pass multiple field filters and their respective values for filtering the data. To separate the `field=value` set, we will use the `'&'` character:

```
GET /customer?firstName=Peter&age=30
```

  It is possible to achieve filtering by a range of values, combining the same field name preceded by the `min` and `max` prefixes, as follows:

```
GET /customer?minAge=35&maxAge=45
```

- **Filtering with comparison operators**: Some APIs also provide the capability of handling comparison operators; this gives a chance to build a more sophisticated set of filtering combinations than a simple field equals value:

```
GET /products?price[gte]=100&price[lte]=500
```

- **List filtering (inclusion/exclusion)**: Using this approach, we can use an operator of `in` or `nin` (not in) on a list:

```
GET /users?role[in]=admin,supervisor,user
GET /products?category[nin]=books,phones
```

As you will realize by now, pagination and filtering are two concepts that you cannot miss in your API implementation. Furthermore, they need to be combined to exploit their advantages on the endpoints of our APIs that are candidates to return lists of data. These are some best practices:

- Use significant, clear, and intuitive parameter names that guide the user to avoid errors
- Validate and sanitize the inputs to avoid attacks and typos
- Validate and implement default limits to avoid the user requesting large values and the server having to pass excessive volumes of data
- Keep consistent naming, response metadata, and approaches across the system to provide a good user/developer experience, making the structure and parametrization standardized and easy to consume

Filtering is a critical aspect of robust API design, enabling clients to retrieve the data they need without overburdening the server or the network. By implementing well-structured filtering mechanisms, APIs can achieve higher performance, scalability, and user satisfaction.

Employing various filtering approaches—from basic field filters to advanced comparison operators and list-based criteria—allows APIs to cater to diverse client requirements and complex querying needs. Moreover, integrating filtering with pagination not only enhances data management but also ensures that responses remain swift and manageable, even when dealing with extensive datasets.

Let us explore a practical example that builds upon our paginated endpoint. Here, we add filtering capabilities to allow clients to specify criteria such as categories or price ranges. Combining pagination and filtering not only enhances data management but also is a best practice that provides a streamlined experience for clients, keeping responses fast and relevant even as datasets grow.

Consider an e-commerce API endpoint that returns a list of products. This endpoint allows clients to request specific pages of data and apply filters to retrieve only the products they need, reducing data load on both the client and server.

The following is an example of an API call using pagination and filtering:

```
GET /api/v1/products?page=2&limit=10&category=electronics&price_
min=100&price_max=500&sort=price_asc
```

It should return something like the following:

```
{
  "data": [
    {
```

```
      "id": 101,
      "name": "Smartphone Model X",
      "category": "electronics",
      "price": 250.00,
      "available": true
    },
    {
      "id": 102,
      "name": "Wireless Earbuds",
      "category": "electronics",
      "price": 120.00,
      "available": true
    }
    // Additional products...
  ],
  "pagination": {
    "totalItems": 42,
    "totalPages": 5,
    "currentPage": 2,
    "pageSize": 10
  }
}
```

Best practices when paginating and filtering

Let's take a look at some of the best practices to follow when paginating and filtering:

- **Use clear and intuitive parameter names**: The parameter names, such as `page`, `limit`, `category`, `priceMin`, and `priceMax`, should be simple, self-explanatory, and aligned with common API design conventions, making the endpoint easy to use and understand.
- **Apply input validation and sanitization**: The API validates all parameters to ensure they are correctly formatted. For instance, `page` and `limit` must be integers, while `priceMin` and `priceMax` must be decimal numbers. Additionally, the `sort` parameter only accepts specific values, preventing injection attacks and typos.
- **Apply default limits and constraints**: In our example, a default limit of 10 items per page is applied, and the `limit` parameter cannot exceed `100`. These measures protect the server from handling overly large requests, enhancing performance and preventing abuse.

- **Employ consistent naming and metadata**: The response includes metadata—`page`, `limit`, `totalPages`, and `totalItems`—that provides clients with essential information about the pagination state, ensuring a standardized and predictable experience.
- **Combine pagination and filtering**: This example demonstrates how pagination and filtering work together. The client can specify page numbers and filters simultaneously, allowing them to retrieve exactly the data they need without overloading the server or client with unnecessary information.

By applying these best practices, the API achieves a balance of flexibility, performance, and user-friendliness, ensuring that clients can access data efficiently while maintaining robust, scalable API design.

With pagination and filtering, we have seen how to manage large datasets efficiently by selectively retrieving information. However, in many applications, handling data is not limited to managing structured information alone. Often, APIs need to support the uploading and downloading of files, allowing clients to transfer data in formats such as images, documents, and reports.

In the next section, we will dive into best practices and techniques for handling file uploads and downloads via REST APIs. These strategies will help ensure secure, reliable file transfers that integrate smoothly with the rest of the API.

Uploading and downloading files via the REST API

Web applications often need to support file uploads and downloads. This functionality is usually accompanied by additional requirements, such as validating file types, names, and maximum sizes, determining optimal storage solutions, providing meaningful responses, and handling large files efficiently through streaming. In this section, we will discuss these common topics in detail.

When designing controllers to handle file uploads, it is essential to ensure that the parameter for the uploaded file is defined using the Spring Web interface, that is, the Spring Boot interface that represents an uploaded file in a multi-part request—`org.springframework.web.multipart.MultipartFile`. Additionally, the request must have its `Content-Type` set to `multipart/form-data`, the standard format used by web browsers for uploading files, so this format is also expected by an API service using `MultipartFile`.

We can demonstrate the upload and download functionality by adding a subresource, `/products/{productId}/photo`, to our example Product API, which will represent a photo image of the product.

The following is an example HTTP request and method to upload a single file:

```
curl -X PUT -F "file=@C:/path/of/your/file.jpeg" http://localhost:8080/
api/products/AK12345/photo

// Single file upload Controller Method
@PutMapping("/{productId}/photo")
public ResponseEntity<String> uploadProductPhoto(
@PathVariable("productId") @ValidSku String productId,
@RequestParam("file") MultipartFile file) {

// Implementation logic
}
```

The `multipart/form-data` format represents a form that can contain multiple fields. It is also possible to use it to upload multiple files in one request. This is why the `"file"` parameter name in the preceding example is important for identifying the file our code wants to work with.

The following is an example HTTP request:

```
curl -X POST -F "file=@C:/path/of/your/file1.txt" -F "file=@C:/path/of/
your/file2.txt" http://localhost:8080/uploads
```

And the following is the method to upload multiple files:

```
// Multiple files upload Controller Method

@PostMapping("/uploads")
public ResponseEntity<String> uploadFiles(@RequestParam("files")
MultipartFile[] files) {

// Implementation logic
}
```

Our example code stores the uploaded file content as a **large object (LOB)** in the relational database for simplicity. Relational databases are not optimized for this kind of data, so in large-scale systems, other types of storage are commonly used.

The file could be saved in an external storage provider, such as AWS S3. This cloud-based storage solution efficiently manages files, providing a link for accessing the stored content. This approach is increasingly popular in the industry as it optimizes resource management and leverages the scalability of cloud infrastructure.

By saving files in cloud data storage and exposing only the link through the API, developers can avoid loading large resources directly. This not only enhances performance but also improves response times for clients, allowing them to retrieve files as needed without overloading the API.

In the preceding examples, the API returns a string in the response, so the returned string could be used to pass the cloud storage URL.

It is advisable to validate uploaded files to ensure security, data integrity, and performance. Some of the possible validations are described in the following section.

Validating the uploaded files

The validation step is critical not only for RESTful APIs but also for any system that you may be developing, to maintain the integrity and security of your application. The inputs you receive can make the application vulnerable to several distinct kinds of attacks. Let us see some examples of important validations to be performed when uploading files.

Validating the file content type

Accepting files of only a certain type would be a good measure to avoid receiving malicious files or even data that does not comply with the data you need to store. In the following example, you ensure that you will only receive PNG and JPEG files:

```
private static final List<String> ALLOWED_CONTENT_TYPES = List.of("image/
png", "image/jpeg");
String contentType = file.getContentType();
if (!ALLOWED_CONTENT_TYPES.contains(contentType)) {
    // report error using the 415 HTTP status code (Unsupported Media
Type)
}
```

Please note that the content type of the whole HTTP request body will be `multipart/form-data`, but each of the files it contains also has its own content type. This is why we need to check the content type using the `getContentType` method of the particular `MultipartFile` object we are going to process.

Besides the content of the file, our example Product API stores the content type of the uploaded file. The same content type is returned by the `GET` endpoint when the file is downloaded. Without the correct `Content-Type` header, opening the URL in a web browser would display gibberish because the browser would not know how to interpret the downloaded file bytes.

Validating the file size

Validating the size of a file is a measure to avoid processing files that are bigger than expected, which can cause denial of service by exhaustion of memory due to the overloading of the system with excessively large files, irrespective of whether the upload is intentional or not.

We can check the file size programmatically using the respective method of the `MultipartFile` class:

```
private static final long MAX_FILE_SIZE = 10 * 1024 * 1024; // 10 MB
if (file.getSize() > MAX_FILE_SIZE) {
  throw new FileSizeLimitExceededException(
    "File size exceeds the allowable limit of 10 MB.");
}
```

File uploads are also limited for the whole application by Spring configuration properties specifying the maximum acceptable multipart request size and the maximum size of one file within such a request. You can set the properties in the `application.yml` or `application.properties` file:

```
spring.servlet.multipart.max-file-size=10MB
spring.servlet.multipart.max-request-size=10MB
```

Validating filenames

To improve file upload security on your server, validating filenames is essential to prevent attacks such as path traversal, where attackers attempt to access restricted areas on your server's filesystem. For example, an attacker might use a filename such as `../../etc/passwd` to try to access sensitive files outside the intended upload directory.

Path traversal attacks can be particularly dangerous in file uploads. By manipulating the file path, an attacker may exploit weak validation to save files in unintended locations on the server, potentially leading to the exposure of sensitive data or even overwriting critical files. For example, if the server directly appends a user-provided filename to an upload path, an attacker could use `../` sequences to navigate up the directory structure, allowing them to access or modify restricted files. This is why validation is crucial when handling file paths, especially in upload scenarios.

While there are several ways to prevent path traversal attacks, here are some effective strategies:

- **Set user and file permissions carefully**: Restrict permissions to limit access only to necessary users and directories.
- **Store files separately**: Consider storing uploaded files on a different server or in a secure storage service such as AWS S3. By keeping files off the main server, you reduce the risk of unauthorized access to your system's file structure.
- **Validate the file path**: Ensure that the file path points to a specific folder that is expected for the files to be sent.
- **Use secure file upload resolvers**: In Spring, classes such as `CommonsMultipartResolver` and `StandardServletMultipartResolver`, as resolvers, are responsible for resolving or interpreting multipart data within a request and help manage file uploads securely. They separate file parts from other form data and make them accessible as `MultipartFile` objects within Spring's request-handling framework. Both of these classes are responsible for parsing multipart file requests, which are commonly used in file uploads. They handle the file data separately from user input, which helps prevent injection attacks:
- **CommonsMultipartResolver**: This class, part of the Apache Commons `FileUpload` library, allows you to configure file size limits and temporary storage directories. It offers flexibility for applications that require strict control over file storage and performance. By setting up limits and constraints through `CommonsMultipartResolver`, you can mitigate risks such as excessive file uploads or unauthorized access to temporary storage.
- **StandardServletMultipartResolver**: This is a resolver in Spring that leverages the built-in multipart support provided by Servlet 3.0, making it a reliable and efficient choice for handling file uploads in Spring applications. By relying on the servlet container's native multipart processing, this resolver avoids the need for additional libraries, simplifying setup and maintenance.

These resolvers not only simplify file handling but also help mitigate potential security vulnerabilities by handling multipart file requests in a standardized way. This ensures that files are processed according to predefined rules, which reduces the risk of path traversal and other injection attacks.

When working with files stored on your application's server, remember not to rely solely on methods such as `StringUtils.cleanPath`. Although commonly used, it should not be the only security measure because it does not fully protect against path traversal. As noted in the Spring documentation (v6.1.12), "`cleanPath` *should not be depended upon in a security context*."

To enhance filename sanitization, you can use the `FilenameUtils` class from the Apache Commons IO library. This utility class provides methods to manipulate and sanitize filenames safely.

Here are some examples of codes that would help to cover some file upload validations:

```
// Extract the base name of the file, removing any path information
String sanitizedFilename = FilenameUtils.getName(originalFilename);

// Enforce character restrictions
if (!sanitizedFilename.matches("^[a-zA-Z0-9._-]+$")) {
    throw new SecurityException("Invalid characters in file name.");
}

// Limit file name length
if (sanitizedFilename.length() > 100) {
    throw new SecurityException("File name too long.");
}

// Check allowed extensions
List<String> allowedExtensions = Arrays.asList("jpg", "png", "pdf");
String extension = FilenameUtils.getExtension(sanitizedFilename);
if (!allowedExtensions.contains(extension.toLowerCase())) {
    throw new SecurityException("File type not allowed.");
}

// Avoid leading dots and double extensions
if (sanitizedFilename.startsWith(".") || sanitizedFilename.contains(".."))
{
    throw new SecurityException("Invalid file name format.");
}

// Ensure path is within the intended directory
Path targetPath = Paths.get("/uploads/")
.resolve(sanitizedFilename)
.normalize();
if (!targetPath.startsWith("/uploads/")) {
    throw new SecurityException("Invalid file path.");
}
```

```
// Optionally, add a unique prefix to avoid conflicts
String uniqueFileName = UUID.randomUUID().toString() + "_" +
sanitizedFilename;
```

This approach provides a thorough sanitization and validation of filenames, ensuring that they do the following:

- Conform to expected characters and formats
- Have reasonable length limits
- Are restricted to allowed file types
- Are stored only within the intended directory
- Are optionally assigned a unique name to prevent conflicts

By combining these security measures and utilizing `FilenameUtils` for safe file name manipulation, you can ensure that file uploads remain secure. This reduces the chance of path traversal attacks and protects your application from unauthorized access. Even if your application stores files on a service such as AWS S3, validating filenames contributes to a consistent, secure, and user-friendly experience.

Using storage services such as AWS S3 can simplify file upload handling by offloading certain security concerns from your application server. Since S3 and similar services store files in a managed environment separate from your server's filesystem, they reduce the risk of path traversal attacks, where an attacker might attempt to save or access files in unauthorized locations. By isolating storage, these services protect your core infrastructure from direct interaction with uploaded files, reducing the potential impact of improperly validated file paths or names.

Additionally, AWS S3 automatically generates unique URLs or identifiers for uploaded files, which minimizes the need for name conflict resolution and additional filename validation. S3 enforces secure naming conventions and ensures that uploaded files are stored safely without affecting other areas of your application. This means your application can focus on verifying basic attributes, such as file type or size, without worrying as much about path traversal or directory restrictions, streamlining the file upload process and enhancing overall security.

After implementing thorough validation to secure filenames and mitigate risks such as path traversal attacks, it is crucial to consider how our API communicates the outcome of the file upload process. Providing clear, meaningful responses ensures that clients are informed about the status of their upload attempts, whether successful or not. Effective responses do more than acknowledge success—they also guide users when an error occurs, helping them understand what went wrong and how to correct it. By returning specific HTTP status codes and relevant metadata, we can make the file upload process more transparent, reliable, and user-friendly.

Providing meaningful responses on our API service for file upload

Provide clear feedback to the client regarding the result of the upload by using appropriate HTTP status codes:

- `201 Created`: When a file is successfully uploaded
- `415 Unsupported Media Type`: For invalid file types
- `413 Payload Too Large`: If the file size exceeds the allowed limit
- `400 Bad Request`: For other validation errors

Include metadata in the response, such as the file URL or identifier for future downloads.

After covering these fundamental yet essential details and highlighting specific precautions for handling files within your API, it becomes clear that even seemingly minor implementations can lead to potential vulnerabilities. Such weaknesses may act as critical points of failure, potentially destabilizing the entire API. For instance, a single malformed file could disrupt API operations, or an attack exploiting file-naming conventions might provide an entry point for malicious actors. The intention of covering this topic was to call your attention to these scenarios that you may encounter when dealing with files on your API.

Earlier in this chapter, we introduced HATEOAS while discussing pagination to demonstrate how hypermedia links can enhance navigation through large datasets. Now, we'll explore HATEOAS in greater detail as it represents a broader architectural pattern that goes beyond pagination, offering a more sophisticated approach to API interaction and resource discovery.

HATEOAS

HATEOAS, as mentioned in the *Pagination* section, is a key principle of RESTful APIs that improves the client's interaction by including navigational links within the response. These hyperlinks guide the client on how to access related resources without needing prior knowledge of the API structure. As we saw before, in a paginated response, HATEOAS can provide links to the next and previous pages, enabling smooth navigation through data without requiring hardcoded logic.

By using HATEOAS, APIs become more self-explanatory and adaptable to changes. Clients can discover available actions, such as editing or deleting, directly from the response, based on the provided links. This simplifies API usage and ensures flexibility, as clients do not have to rely on external documentation to understand the API's behavior, reducing the chance of errors when updates occur.

Let us see an example of using HATEOAS to expose available operations on a resource:

```
{
  "id": 123,
  "name": "John Doe",
  "email": "johndoe@example.com",
  "links": [
    {
      "rel": "self",
      "href": "/users/123",
      "method": "GET"
    },
    {
      "rel": "edit",
      "href": "/users/123",
      "method": "PUT"
    },
    {
      "rel": "delete",
      "href": "/users/123",
      "method": "DELETE"
    }
  ]
}
```

In the preceding example, the response contains three elements in the `links` section. The first element, `"rel": "self"`, indicates the link to the current resource itself. This allows the client to retrieve or interact with a specific user using the `GET` method at the URL provided in the `href` (which stands for hypermedia reference) attribute. The second element, `"rel": "edit"`, provides a link for updating the user's details with the `PUT` method. Lastly, the `"rel": "delete"` element allows the client to remove the user with the `DELETE` method. Each of these links guides the client through interactions with the resource, making the API self-explanatory and easier to use.

HATEOAS, as outlined in RFC 8288, formalizes the use of web links in APIs, ensuring consistency and clarity in how these links are structured and interpreted. This RFC defines key components such as the `rel` attribute, which specifies the relation type of a link (e.g., `"self"` or `"edit"`), and `href`, which defines the target URL for the interaction. By adhering to this standard, APIs become more robust, with clients able to navigate resources and perform actions without needing to hardcode endpoint paths or refer to external documentation. This approach enhances flexibility and future-proofs API interactions, as the server can evolve while still maintaining compatibility with existing clients.

The same example we saw before can also have a different response, where certain actions—such as deleting the element—are not available. For instance, in the following example, the response for the user `"Jane Smith"` does not include a `"delete"` link in the `links` section:

```
{
  "id": 456,
  "name": "Jane Smith",
  "email": "janesmith@example.com",
  "links": [
    {
      "rel": "self",
      "href": "/users/456",
      "method": "GET"
    },
    {
      "rel": "edit",
      "href": "/users/456",
      "method": "PUT"
    }
  ]
}
```

In this case, the absence of the `delete` link indicates that the `delete` operation is not available for this resource, due to permission restrictions or business rules. HATEOAS helps make these limitations clear to the client, ensuring that the client does not attempt unsupported actions. This allows the API to dynamically control which actions are allowed, preventing misuse and guiding the client's behavior based on the current state of the resource.

To simplify the implementation of HATEOAS, modern API frameworks offer built-in tools to support this pattern. For example, in Spring Boot, you can leverage the `spring-boot-starter-hateoas` dependency to easily incorporate HATEOAS into your API:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-hateoas</artifactId>
</dependency>
```

HATEOAS enhances the dynamism of the API, advancing it to a higher level of maturity by incorporating all the benefits we have mentioned so far. However, it is important to evaluate whether this resource is necessary for your use case, as HATEOAS is not a mandatory component for every API. It requires another dependency, additional development, and maintenance, and then requires attention to understand whether it makes sense to have it.

We have covered essential practices for building a reliable and user-friendly API by employing efficient data-handling techniques. Now, as we transition to resilience patterns, we will explore strategies that help APIs maintain stability and robustness, even in the face of unexpected challenges. Resilience is crucial for ensuring that an API continues to function reliably under various adverse conditions, from network issues to system overloads. In the upcoming section, we will dive into patterns that enable APIs to handle such scenarios gracefully, enhancing their dependability and user experience.

Resilience

Before discussing how to make an API resilient, let's look at the Oxford English dictionary's definition for resilience: "*(of a person or animal) able to withstand or recover quickly from difficult conditions.*" If we try to apply the same definition to our RESTful APIs or systems in general, we could say that a resilient API is an API that can withstand failures and disruptions while maintaining its functionality and performance, recovering from situations that could shut it down or cause a degradation in performance.

To ensure that an API serves as a reliable component of a larger, robust system, it must handle excess spikes, performance degradation of services it depends on, and infrastructure failures effectively. For that, we need to observe several aspects of the API design that we are going to approach. We will explore strategies and techniques to achieve these capabilities, ensuring your APIs can deliver consistent value.

Regarding API design, the key principles that we need to observe are the following:

- **Redundancy**: Have multiple instances of our components/services to avoid a single point of failure.
- **Decoupling**: Minimize dependencies in the design, aiming to limit impact during any system part failure.
- Minimize direct dependencies within a system's design by reducing the interconnections between its components, which helps limit the impact of any single component's failure. When an API is tightly coupled to other services or components, an issue in one part can cascade throughout the system, leading to widespread failures or degraded performance. By contrast, a decoupled system allows components to operate more independently, isolating faults and preventing them from affecting other parts.
- **Fault isolation**: Use techniques such as bulkheads and circuit breakers to confine failures to isolated parts of the system, preventing error states from propagating throughout the entire system, as would occur with a "poison pill."
- **Graceful degradation**: Keep limited functionality even when parts of the system fail or are underperforming. This way prioritizes the main functionalities, preventing the entire system from stopping.

Let us list some concepts and patterns that we will talk about and the problems that each one of them can help us to solve:

- Timeouts
- Retry mechanisms
- Rate limiting and throttling
- Idempotency key
- Circuit breaker
- Bulkhead

Timeouts

Establishing appropriate timeout settings is critical when designing APIs to ensure system resilience. API interactions often involve synchronous remote calls between services, whether they reside on the same network or across different networks. This synchronous communication means that the client remains unaware of the server's processing status, maintains the network connection, and waits until the call either succeeds or fails. Without well-defined timeouts, clients might experience prolonged wait times for responses, which can degrade user experience, introduce security vulnerabilities, and cause system instability or downtime.

The timeout can be set from two perspectives:

- Client configuration
- Server configuration

Client configuration

Configuring timeouts on the client side is essential to prevent clients from waiting indefinitely for a server response. By setting a specific timeout duration, the client ensures that it does not become unresponsive due to delayed server replies. When the server fails to respond within the designated timeout period, the client should do the following:

- **Alert users**: Inform users about the delay to maintain transparency and manage expectations
- **Log the event**: Record the timeout occurrence in logs for monitoring and troubleshooting purposes
- **Initiate retry mechanisms**: Depending on the application requirements, the client may attempt to resend the request to recover from transient issues

Let's implement the timeout for the API client using Spring Boot's `RestClient`. In our example, we will configure a timeout of six seconds, striking a balance between allowing sufficient time for normal operations and preventing excessive waiting periods. Here is how to configure `RestClient` with the appropriate timeout settings:

```
@Configuration
public class ProductsApiConfiguration {
    @Bean
    public ProductsApi getProductsApi(ApiClient apiClient)
    {
        return new ProductsApi(apiClient);
```

```
    }

    @Bean
    public ApiClient getApiClient(RestClient restClient) {
        return new ApiClient(restClient);
    }

    @Bean
    public RestClient getRestClient() {
        return RestClient.builder()

       .requestFactory(customClientHttpRequestFactory())
            .build();
    }

    private ClientHttpRequestFactory customClientHttpRequestFactory() {
        Duration duration = Duration.ofSeconds(6);
        ClientHttpRequestFactorySettings settings =
            ClientHttpRequestFactorySettings.DEFAULTS
            .withConnectTimeout(duration)
            .withReadTimeout(duration);

        return ClientHttpRequestFactories.get(settings);
    }
}
```

This example demonstrated how to configure `RestClient` with appropriate timeout settings to establish connections and receive data. Let us set up these timeouts to ensure our client avoids waiting too long for server responses.

Server configuration

When a server receives a request, it begins processing the necessary information, which may involve interacting with other servers, executing database queries, and performing various computational tasks. If these operations exceed the predefined timeout threshold, the server should do the following:

- **Terminate processing**: Stop any ongoing operations related to the request to free up resources

- **Release allocated resources**: Ensure that resources such as memory, threads, and database connections are properly released to prevent leaks and bottlenecks
- **Notify the client**: Inform the client that a timeout has occurred using the appropriate HTTP error code, `408 Request Timeout`, allowing the client to handle the situation appropriately (e.g., by retrying the request or notifying the user)

Before setting a timeout, it is important to observe the typical responses of your service and network conditions; this will allow you to have a proper notion of what is a "normal" expected time to respond, because setting a very short timeout can cause unnecessary failures, whereas a very long timeout can lead to poor user experience. The best application of this practice requires constant monitoring of the timeout metrics and adjusting configurations, both client and server, to maintain a healthy, resilient, and sustainable API.

To illustrate client timeout settings in a Spring Boot application, you can configure the connection timeout and read timeout within `ClientHttpRequestFactorySettings`. This class is responsible for setting up the HTTP request factory with specific timeout parameters. These settings are then used to create a `RestClient` instance, which is a component for executing HTTP requests to other services. For more detailed information on this configuration, please refer to the `ProductsApiConfiguration` class in the `chapter6` folder of the book's source code repository.

While the last example focuses on client-side configuration, it's worth noting that server-side timeout configuration in Spring Boot typically involves setting properties in your `application.properties` or `application.yml` file.

Take the following example:

```
server:
  tomcat:
    connection-timeout: 5000
```

This sets the server's connection timeout to five seconds. The specific properties and their effects can vary depending on the server you're using (e.g., Tomcat or Jetty).

The previous examples show basic client-side and server-side timeout configurations. However, in more complex applications, you might need more advanced timeout management. The example in this book is intentionally simple to focus on key concepts. For real-world applications, especially those with complex database operations or microservices, you might find more sophisticated libraries, such as Resilience4j, useful.

Resilience4j offers a `TimeLimiter` module that provides detailed control over timeouts and works well with Spring Boot applications. While this level of complexity is not necessary for our basic example, understanding these advanced techniques can be valuable as your applications become more complex. Here's an example using Resilience4j version 2.2.0 to manage timeouts in a more advanced scenario:

1. Add the Resilience4j dependency to your project:

```
<dependency>
    <groupId>io.github.resilience4j</groupId>
    <artifactId>resilience4j-spring-boot3</artifactId>
    <version>2.2.0</version>
</dependency>
```

2. Configure `TimeLimiter` in your `application.yml`:

```
resilience4j:
  timelimiter:
    instances:
      productServiceGetById:
        timeout-duration: 5s
        cancel-running-future: true
```

3. Implement the service with `TimeLimiter`:

```
@Service
public class ProductsQueryUseCaseImpl implements
ProductsQueryUseCase {
    private final ProductsRepository productsRepository;
    private final PhotoProductsRepository photoProductsRepository;
    private final TimeLimiter timeLimiter;

    public ProductsQueryUseCaseImpl(
        ProductsRepository productsRepository,
        PhotoProductsRepository photoProductsRepository,
        TimeLimiterRegistry timeLimiterRegistry) {
        this.productsRepository = productsRepository;
        this.photoProductsRepository = photoProductsRepository;
        this.timeLimiter = timeLimiterRegistry.timeLimiter(
            "productServiceGetById");
```

```
    }
    @Override
    public CompletableFuture<
        Product> getProductByIdAsync(String productId) {
            return timeLimiter.executeCompletionStage(
                Executors.newSingleThreadScheduledExecutor(),
                () -> CompletableFuture.supplyAsync(() ->
                    getProductById(productId)))
                    .toCompletableFuture();
    }
```

4. Use the service in a controller:

```
@RestController
@RequestMapping("/api/products")
@Validated
public class ProductsApiController implements ProductsApi {
    @GetMapping(value = "/{productId}")
    @Override
    public ResponseEntity<ProductOutput> getProductById(
        @PathVariable("productId") @ValidSku String productId) {
        try {
            final var product = productsQueryUseCase
                .getProductByIdAsync(productId).get();
            return ResponseEntity.status(HttpStatus.OK)
                .body(productMapper.toProductOutput(product));
        } catch (ExecutionException ex) {
            if (ex.getCause() instanceof TimeoutException) {
                throw new ResponseStatusException(
                    HttpStatus.REQUEST_TIMEOUT,
                    "Timeout to access the product id:
                     [%s]".formatted(productId), ex);
            }
            throw new UnexpectedServerError(
                "Error to access the product.", ex);
        } catch (Exception ex) {
            throw new UnexpectedServerError(
```

```
                "Error to access the product.", ex);
        }
    }
}
```

In this example, note the following:

- We have an instance of `TimeLimiter` that is built by a named `TimeLimiterRegistry` using the configuration named `productServiceGetById`, which has a configuration for a five-second timeout.
- The `ProductsQueryUseCaseImpl.getProductById()` method uses the `timeLimiter.executeCompetionStage()` method, which wraps the repository call in a `CompletableFuture` to make it asynchronous.
- In a real scenario, if a complex database query or high server load causes a delay, it will be detected by this `timeLimiter` call, throwing one exception.

That will be detected by the controller's `try/catch` and return a timeout HTTP code.

This example shows how to apply timeouts to specific service methods, such as those involving database operations. Although it is more complex than the book's main example, it demonstrates how you can improve your timeout strategies as your application's needs grow.

In practice, you would choose the appropriate level of timeout management based on your specific requirements. You might start with simpler approaches and move to more advanced solutions, such as Resilience4j, as your system becomes more complex.

While timeouts are crucial for maintaining system responsiveness, they often work hand in hand with another important resilience pattern: retry mechanisms. When a timeout occurs, it may be due to temporary issues that could be resolved quickly. In such cases, automatically retrying the operation can help maintain service continuity without user intervention. Let's explore how retry mechanisms complement timeout strategies and enhance overall system resilience.

Retry mechanism

Implementing a retry mechanism is an effective strategy to prevent the temporary unavailability of a system component from disrupting the entire chain of dependent processes. While retries can mitigate temporary failures, excessive or inappropriate retries can exacerbate issues, leading to resource exhaustion and degraded performance. We will present how to properly deal with this here.

There are countless reasons why an API might fail to complete a process cycle, affecting various tasks within a request. Here are some cases that you can handle with retries:

- Timeouts
- 5xx server errors (e.g., `500 Internal Server Error` or `503 Service Unavailable`)
- Rate limiting (e.g., `429 Too Many Requests`)

Consider an e-commerce platform where verifying the stock count of a product is a critical step in finalizing a sale. Suppose the product stock verification API typically responds within 2 to 5 seconds. If this API experiences a temporary delay or failure, implement a retry mechanism to ensure that the transaction can still proceed without unnecessary abandonment. Take the following example:

- **Initial request**: The system requests the stock count
- **First failure**: The API does not respond within the expected period
- **Retry attempt**: The system retries the request after a brief delay
- **Successful response**: On the second attempt, the API responds successfully, allowing the sale to proceed

This approach minimizes the risk of losing transactions due to temporary API unavailability. However, retries can become problematic if the product stock API is down and recovery takes longer. Continuous retries without communicating with other systems involved in the transaction can lead to a poor customer experience and cause considerable damage and loss to the company.

To not get trapped in the pitfalls that we mentioned, you need to consider some best practices when implementing your retry policy:

- Set a maximum retry limit
- Use exponential backoff with jitter
- Have idempotent requests
- Implement timeout mechanisms
- Log retry attempts

Set a maximum retry limit

Without a maximum retry limit, your system could enter an endless loop of trying to resend a failed request. Imagine a scenario where a server is down for an extended period. If there is no limit on retries, the system will keep trying to resend the request forever, wasting resources and potentially causing other issues.

Use exponential backoff with jitter

Exponential backoff is a method used to manage the timing of retry attempts after a failure. Instead of trying to resend the message at regular intervals, the system waits longer each time it fails to resend. The wait time increases exponentially, meaning it increases quickly.

Let us look at an example:

1. **First attempt**: If sending the message fails, wait for a brief period of time (e.g., one second) before trying again.
2. **Second attempt**: If it fails again, wait twice as long (e.g., two seconds).
3. **Third attempt**: Double the wait time, that is, four seconds.
4. **Continue depending on the limit of tries**: At every new retry, the wait time doubles (8 seconds, 16 seconds, etc.).

This approach helps prevent the system from repeatedly trying to send messages too quickly, which can cause more problems, such as overloading the server or network.

Jitter

Jitter is a random time used as a time interval for retrying failed requests. Instead of having the retry mechanism make all the calls at a fixed time or even with exponential progress, we have a random time for each new retry. Instead of waiting exactly 2 seconds, for example, it might wait between 1 and 3 seconds.

Combining exponential backoff and jitter

Using only exponential backoff can still cause issues. If many devices or systems are trying to resend messages at the same time, they might all wait the same amount and then retry together, creating a sudden peak of traffic. This can make the network or server even more overwhelming, which is why it is best practice to combine exponential backoff and jitter.

Adding jitter helps by spreading out the retry attempts. Each device waits for a slightly different time, reducing the chance that they all send requests at the same time. This makes the system more stable and increases the chances that messages will be successfully sent.

Have idempotent requests

Having idempotent requests when retrying API calls is vital for building resilient and reliable systems. It ensures that multiple attempts to perform the same operation do not lead to unintended consequences, such as duplicate actions or inconsistent data.

Implement timeout mechanisms

You are already aware of the importance of timeouts from the previous topic. A retry involves sending a new request to the service, and it is essential to manage how long we wait before taking the next action.

Log retry attempts

Logging each failure during retry attempts will help us understand recurring patterns or frequent issues causing the retries, and it will ease system troubleshooting.

A well-implemented retry mechanism significantly enhances the resilience of APIs by gracefully handling transient failures. By adhering to best practices—such as limiting retry attempts, employing exponential backoff, guaranteeing idempotent retries, and distinguishing error types—developers can ensure that retries contribute to system stability without introducing new vulnerabilities. Thoughtful retry strategies lead to improved user experiences, higher system reliability, and sustained operational efficiency.

Although retry mechanisms help handle temporary failures, unrestricted retries could overwhelm your system. This brings us to another crucial resilience pattern: rate limiting. By controlling the frequency and volume of requests, rate limiting helps maintain system stability even during high-load situations.

Rate limiting

Rate limiting is a critical strategy used to protect APIs from abuse and excessive resource access. It ensures the stability, reliability, and security of API systems by restricting the number of requests an API can handle from a user (user, IP, application, token, etc.) within a specified period. This approach helps to prevent **denial of service (DoS)** attacks and ensures the fair distribution of services among clients, preventing exhaustion of resources.

We can apply rate limiting on distinct levels of context; it can be based on the following:

- **Global level**: In this context, we can limit the total number of requests the API can handle by period
- **IP level**: By IP, we limit the number of requests based on the IP address of the client
- **User level**: Limits requests based on individual users or clients
- **Application level**: Limits based on a specific application using the API

In the following sections, we will explore some of the aspects of implementing rate limiting, such as quota management, time windows, and rate-limiting headers.

Rate limiting key aspects

To effectively manage rate limiting, it is essential to understand its core components and how they contribute to overall API resilience. Rate limiting is not just about restricting requests; it is about controlling access fairly, transparently, and scalably, ensuring a stable experience for all users while protecting resources.

In this section, we will explore three critical aspects of rate limiting:

- **Quota management**: Defines the maximum number of requests allowed per client within a given period (e.g., 1,500 requests per hour).
- **Time windows**: The period during which the request count is measured. Common intervals include per second, minute, hour, or day.
- **Rate limit headers**: APIs often include headers in responses to inform clients about their current usage and remaining quota. Examples include the following:
    - `X-Rate-Limit-Limit`: The maximum number of requests allowed
    - `X-Rate-Limit-Remaining`: The number of requests left in the current window
    - `X-Rate-Limit-Reset`: The time when the rate limit will reset
- **Provide the proper response HTTP code when it reaches the rating limit**: A `429 Too Many Requests` error is advised by RFC 6585.

Now that you know what these aspects are and understand what they mean, let us talk about different implementations of rate limiting on your API.

Rate limiting implementation strategies

Implementing rate limiting requires careful consideration of various strategies, each with its advantages and trade-offs:

- **Fixed window**: This strategy has the time split into fixed intervals (windows), such as one minute. The count of requests is reset at the start of each new window. This is the simplest implementation, although it might be susceptible to bursts at window boundaries (e.g., a client could send double the allowed requests in a brief period and have multiple requests rejected). Depending on the application, it may be acceptable to employ this simple strategy to implement and have occasional failures.

- **Sliding window**: Unlike fixed windows, sliding windows move continuously. For each request, the system checks the number of requests in the past defined period for readjusting itself. It will result in a smoother distribution of allowed requests over time, reducing the chance of overloaded periods and periods without any requests. It has a more complex implementation but shows precise control over request rates.
- **Token bucket**: A bucket holds tokens representing the number of allowed requests. Tokens are added, refilling the bucket based on the number of allowed requests per configured time. Each request consumes a token. If no tokens are available, the request is rejected. It has a slightly more complex logic, but will support the APIs that need to handle occasional spikes without compromising overall rate limits.
- **Leaky bucket**: This strategy is like a token bucket but processes requests at a fixed rate, queuing requests where the bucket limit is the mark that determines whether we accept new requests. If the queue/bucket is full, additional requests are rejected. Every time a request is processed from the queue, it opens space for a new, incoming request. It can cause an increase in latency due to the queuing. However, it will ensure a steady processing rate. This is advantageous for systems where maintaining a consistent processing rate is critical.

The rate limiting implementation will protect the API against abuse by malicious actors overwhelming the API, support resource management, ensure a fair distribution of resources among the clients, and maintain the API's responsiveness by controlling the traffic flow.

While rate limiting focuses on restricting the total number of requests, a related but distinct concept is throttling. Understanding the difference between these two approaches is crucial for implementing effective request management strategies.

Gateway-level rate limiting

While rate limiting inside individual API services provides good protection, implementing rate limiting at the gateway level offers important benefits that experienced developers should consider for production systems.

Single point of control

API gateways act as the main entry point for all client requests. This makes them perfect places to set up rate-limiting rules. Instead of adding rate-limiting code to each service separately, you can manage all policies from one central location. This approach reduces complexity and makes maintenance easier across your entire system.

Stopping requests early

Gateway-level rate limiting blocks excessive requests before they reach your application services. When clients send too many requests, the gateway immediately returns a 429 error code without using any backend resources, such as database connections or server processing power. This early blocking protects your system during traffic spikes and saves computing resources.

System-wide rate limiting

In complex systems where clients use multiple services, gateway rate limiting lets you set limits across your entire API. For example, you can limit a user to 5,000 requests per hour across all services, not just individual ones. This prevents clients from avoiding limits by spreading requests across different endpoints.

Implementation options

Popular gateway tools such as Kong, AWS API Gateway, and Nginx offer built-in rate-limiting features. These tools support different strategies, such as the token bucket and sliding window methods. When using multiple gateway servers, consider using shared storage such as Redis to track request counts accurately across all servers.

Gateway-level rate limiting works best when combined with service-level rate limiting. Use gateways for general limits and overall system protection, while keeping specific business rules and detailed limits within individual services.

Throttling

Throttling, on the other hand, manages the rate at which requests are handled. Instead of limiting the total number of requests, throttling makes sure that requests are processed at a steady pace. This is often done by queuing or delaying requests to keep the system stable.

For real-time data, the API limits how frequently each user can make requests to prevent server overload. In this case, each user can make only one request per second. If a user sends requests faster than this, throttling will slow down their requests, but will not block them completely.

This is how it works:

1. User X sends three requests in one second.
2. The API processes the first request immediately.
3. The API delays the second and third requests, ensuring each request is spaced one second apart.

Throttling controls the pace of requests, allowing the server to handle high-frequency requests without being overwhelmed.

What is the difference between rate limiting and throttling, then?

- **Rate limiting**: Controls the total number of requests over a period (e.g., 500 requests per hour)
- **Throttling**: Controls the speed of requests, slowing them down if they come in too quickly (e.g., one request per second)

Using both together ensures that the API remains responsive and fair, even with high demand.

Throttling implementation strategies

The following are some of the throttling implementation strategies:

- **Dynamic throttling**: Adjusts limits based on real-time system performance and load. This implementation is more responsive to changing conditions, although it requires sophisticated monitoring and adjustment mechanisms.
- **Priority-based throttling**: Assigns priorities to different types of requests or clients. In this way, it ensures critical operations have access to necessary resources; however, this approach is complex to manage.
- **Graceful degradation**: Reduces functionality or performance in a controlled manner when under high load. During peak usage, this strategy maintains some level of service. This strategy may come at the cost of some periods of bad user experience.
- **Circuit breaker pattern**: Temporarily stop requests flowing to some specific route, to prevent system overload and allow recovery. It will protect the system against cascading failures. As a side effect, it may require careful configuration to avoid unnecessary shutdowns.

Throttling helps ensure the reliability of APIs, maintaining stability under varying conditions. It allocates resources efficiently based on current demands, preventing slowdowns and outages, and maintaining the consistency of the service.

This strategy is used by many large API providers, such as Microsoft's GitHub API, which has a detailed rate-limiting system based on distinct levels. The level is determined by the number of simultaneous requests you make.

To simplify things, let us assume we are at the most basic level. In this case, rate limits depend on our authentication. For example, unauthenticated requests that can access public data have their specific limits. Authenticated requests, on the other hand, have different rate limits (bigger) depending on the type of authentication, such as using a token or OAuth.

This does not mean you need to implement something that complex. This is an example of how rate limits can help manage the demand on your API, depending on the level of implementation. It depends only on your needs and creativity.

A quite simple example where you can find rate limits widely used on the market of API providers is the limiting of the usage for the free user plan, as you can see on different Google APIs, for example.

Rate limiting and throttling are essential mechanisms for API management, helping to maintain the integrity and reliability of services. Rate limiting focuses on controlling the number of requests per client within a specific time, while throttling employs strategies to manage overall traffic flow and system performance.

For API providers, implementing effective rate limiting and throttling ensures fair usage, protects against abuse, and maintains optimal performance. For API consumers, understanding these mechanisms allows for the design of resilient applications that interact smoothly with APIs, handle limitations gracefully, and provide a seamless user experience.

Managing the request flow through throttling is important, but ensuring request consistency is equally crucial. This is where idempotency comes into play, particularly when dealing with retried requests that might have succeeded but failed to communicate their success.

Idempotency key

To understand why an idempotency key is necessary, it is essential to first understand the concept of idempotency. If you do not recall this from the second chapter, now is a good time to revisit that section. In simple terms, idempotency means calling the API multiple times with the same parameters while ensuring that the state remains unchanged and no unintended side effects occur.

According to the idempotency key RFC (there is no defined number yet) about the idempotency key header, "The HTTP Idempotency-Key request header field can be used to carry an idempotency key in order to make non-idempotent HTTP methods such as `POST` or `PATCH` fault-tolerant."

Imagine the following scenario: we have a frontend that sends an order to our management order API, and this happens when the user clicks on the **Save Order** button; this will trigger the code that will make a `POST` call with all the data required to create a new order. Let us imagine that due to internet instability, the user does not see the result on the screen as fast as usual and hits this button multiple times. This means we will receive multiple `POST` calls with the same data. As an undesirable side effect, we would have the same order repeated multiple times, which causes an inconsistency in our system.

If our frontend added on each call a header item `"Idempotency-Key"`, a key that satisfies the requirement that its value is the same if and only if it is the same request (order) from a business perspective, it would prevent this undesirable side effect of having multiple wrong records stored on the management API because `"Idempotency-Key"` would serve as a unique key to avoid these repetitions.

Here is an example of one call to an API that requires the usage of an idempotency key:

```
curl https://api.service.com/v1/orders \
  -u Ao24N7La2PDTkdtegf553lJI: \
  -H "Idempotency-Key: AF1GvyNVeriLWuDU" \
  -d description="My Order Test"
```

In the preceding example, we have the `"Idempotency-Key"` header, and this header is sent by the request client. The Stripe documentation recommends generating a key with V4 UUIDs, or another random string with enough entropy to avoid collisions. By doing so, we can guarantee idempotency even for methods that are not idempotent.

In summary, using an idempotency key is crucial for maintaining the reliability and consistency of APIs, especially when handling non-idempotent requests such as `POST` or `PATCH`. By assigning a unique key to each request, developers can prevent duplicate actions, such as multiple orders being created unintentionally. This approach ensures that even if a user sends the same request multiple times due to network issues or repeated clicks, the system remains stable and accurate. Implementing idempotency keys not only safeguards the integrity of the data but also enhances the user experience by avoiding errors and inconsistencies. Adhering to best practices, such as generating unique and secure keys, further strengthens the API's resilience and fault tolerance.

UUID V4

A UUID V4 is a 128-bit value used to uniquely identify information in computer systems. It is randomly generated and follows a specific format to ensure uniqueness. Unlike other versions, a V4 UUID relies on random numbers, which makes it highly unlikely to generate duplicates. This type of identifier is often used when a unique reference is needed, such as in database entries or API requests, to ensure data consistency and prevent collisions.

While idempotency ensures request consistency, it does not address the broader challenge of handling failing downstream services. The circuit breaker pattern offers a solution by preventing cascading failures in distributed systems.

Circuit breaker

Inspired by electrical circuit breakers, which protect electrical systems from damage caused by overloads or short circuits, the circuit breaker pattern in software works analogously. It helps prevent system failures by stopping repeated attempts at unsuccessful operations until the system recovers. Just like in an electrical circuit, when the switch is closed, energy flows through, but opening the switch disrupts the flow.

When a certain number or percentage of failures is detected, the pattern opens the faulty circuit or component and redirects all requests to a fallback routine. After some time, it tries again, and if not all but some requests succeed, then it assumes the state of half open. Then, depending on the stability of the subsequent requests, it can open or close, and so on.

Some frameworks also allow you to ignore certain types of failures (Java exceptions) and only count specific ones, offering more control over how the system responds to avoid issues. Depending on the library or framework you choose, the circuit breaker configuration can have an extensive list of parameters to customize how the circuit breaker needs to behave. In the following figure, we have an illustration of the workflow and an example of the Resilience4j configuration for our circuit breaker.

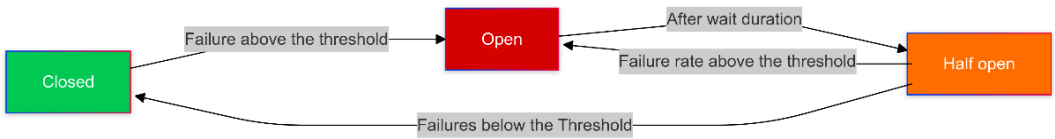


Figure 6.1 – Circuit breaker flow

The following is an example of implementation using Resilience4j:

```
CircuitBreakerRegistry circuitBreakerRegistry = CircuitBreakerRegistry.
ofDefaults();

// Configure the CircuitBreaker
CircuitBreakerConfig circuitBreakerConfig = CircuitBreakerConfig.
ofDefaults()
        .failureRateThreshold(50) // 50% failure rate
        .waitDurationInOpenState(Duration.ofSeconds(10))
        .slowCallDurationThreshold(Duration.ofSeconds(5));

// Create a CircuitBreaker instance
CircuitBreaker circuitBreaker = circuitBreakerRegistry.circuitBreaker(
    "circuitService", circuitBreakerConfig);

// Use the CircuitBreaker to protect a downstream service call
try {
    circuitBreaker.decorateSupplier(() -> downstreamService.call()).get();
} catch (CircuitBreakerOpenException e) {
    // Fallback or other suitable action
}
```

This code sets up a Resilience4j circuit breaker to monitor a downstream service call. It does the following:

- Opens after the service fails more than 50% of the time
- Remains open for 10 seconds before allowing test requests
- Treats calls taking longer than five seconds as failures

By using the circuit breaker pattern, the API becomes more resilient, as it limits repeated calls to failing services and recovers once the service stabilizes.

In the previous example, we saw how a circuit breaker can help manage failures in downstream services, protecting the API by isolating and handling service disruptions. However, implementing circuit breakers within each API service can lead to duplicated code and maintenance challenges, especially in larger systems with multiple services.

Let us explore this further by understanding a different way of implementing a circuit breaker that is implemented at the level of the system gateway.

Circuit breaker implemented in the gateway

A gateway functions as a central control point for managing communication between various services or systems. This makes it an optimal location for implementing patterns such as circuit breakers, rate limiting, and file upload validation. By incorporating these mechanisms at the gateway level, operational concerns are separated from business logic, allowing for isolation and promoting the standardization of system configurations, all managed through the gateway.

Here are some benefits of implementing a circuit breaker on a gateway:

- **Centralized control**: A single circuit breaker can protect multiple downstream services, simplifying management and monitoring
- **Improved resilience**: By isolating failing services, a circuit breaker can prevent cascading failures and maintain overall system stability
- **Reduced latency**: When a service is deemed unhealthy, the circuit breaker can quickly return a fallback response, reducing latency for clients
- **Enhanced reliability**: A circuit breaker can help ensure that critical services remain available even in the face of temporary failures

The circuit breaker pattern is a crucial tool for building reliable and stable APIs, especially those that rely on multiple services or external APIs. You should consider using this pattern in several key situations to enhance resilience and stability.

Firstly, when building distributed systems or microservices, distinct parts of your application communicate with each other through APIs. In such setups, if one service becomes slow or fails, it can negatively impact the entire system. By implementing a circuit breaker, you can monitor these API calls and stop attempts to communicate with the failing service temporarily. This prevents one service's issues from causing a domino effect, ensuring that other parts of the system continue to function smoothly.

Another important scenario is when your application relies on external services. Many applications depend on third-party APIs, such as payment gateways, social media platforms, or data providers. These external services can sometimes experience downtime or slow responses due to several reasons, such as maintenance or high traffic. A circuit breaker helps by detecting these failures early and avoiding long waits for responses that may never come. Instead, your application can provide fallback options, such as default messages or alternative actions, maintaining a good user experience even when external services are unreliable.

To prevent transaction failures, systems often cache frequently calculated values such as delivery tax, which may depend on external services. In situations where a quick response is crucial, a cached value can serve as a fallback option, avoiding lost transactions. A circuit breaker pattern can be effectively used to manage these scenarios. Promote a fallback solution until the part of the circuit that is opened gets to a normal state again.

Circuit breakers help prevent system failures by stopping repeated attempts at unsuccessful operations. However, sometimes we need to isolate different parts of our system to prevent resource exhaustion. This is where the bulkhead pattern becomes valuable.

Bulkhead

Imagine you have a system where one small part fails, but you do not want the entire system to go down because of it. This is where the bulkhead pattern comes in. Let us explore how this pattern helps improve fault tolerance and keeps your system running smoothly.

The bulkhead pattern is inspired by ships. Ships have bulkheads, which are walls that divide the ship into separate sections. If one section takes on water, the bulkheads prevent the water from spreading to other parts of the ship. This keeps the ship from sinking entirely.

In software, the bulkhead pattern works similarly by dividing an application into various parts or services that operate independently. If one part fails or slows down, the other parts continue to function without any issues. This isolation ensures that a problem in one area does not affect the entire system.

As mentioned, the bulkhead pattern is a structural design pattern used to partition a system into isolated units or modules. Each module can handle its own load and failures independently, ensuring that an issue in one does not cascade to others. This isolation can be applied at various levels, such as the following:

- **Service level**: Separating different microservices within an architecture
- **Resource level**: Allocating separate thread pools, memory, or database connections to different components
- **Functional level**: Dividing functionalities within a service to prevent one failing feature from affecting others

The key characteristics of this pattern include the following:

- **Isolation**: Ensures that failures in one compartment do not impact others
- **Resource allocation**: Allocates separate resources to each compartment to prevent resource exhaustion from affecting the entire system
- **Fault tolerance**: Enhances the system's ability to handle partial failures gracefully

A common example of using the bulkhead pattern is managing "hot paths" in your system. Hot paths are parts of your system that receive more traffic than others. If a hot path gets too many requests, it can overload that part of the system, causing it to slow down or crash. This can make the entire system unresponsive, affecting users who are not using the hot path.

To understand how the bulkhead pattern enhances resilience, we will dive into different areas where this approach can be applied effectively. By isolating components, we can prevent failures in one part of the system from affecting the entire application. This flexibility allows us to apply the bulkhead pattern across various levels, such as services, resources, and functionalities, making it a versatile solution for fault tolerance.

Let's explore some practical ways to implement bulkheads in different system architectures and at different levels, from monolithic systems to microservices, as well as through database and queue management.

Implementing bulkhead in different architectures

You can apply the bulkhead pattern whether you are using a monolithic architecture or microservices:

- **Monolithic systems**: Even if your system is a single application, you can create isolation by directing heavy traffic to specific instances. For example, you can have certain instances handle the hot paths while other instances manage different parts of the system. This way, if the hot path gets overloaded, the rest of the system remains unaffected.
- **Microservices**: In a microservices architecture, each service can act as a separate bulkhead. If one service fails, the others continue to operate normally.

Implementing the bulkhead pattern through database isolation

Another way to implement the bulkhead pattern is by isolating your databases. Instead of using a single database that handles all requests, you can create separate database instances for different parts of your system. For example, a high-traffic feature can have its own database, preventing it from overwhelming the main database and affecting other features.

Implementing the bulkhead pattern through queue management

You can also apply isolation to message queues by creating different queues with varying priorities. High-priority queues can be monitored and scaled independently to ensure they process messages quickly, while standard queues operate normally. This prevents a surge in one queue from impacting others.

External services and throttling

When your system makes calls to external services, latency and timeouts can cause problems. By using the bulkhead pattern, you can limit the number of external requests, ensuring that slow responses from third-party services do not overwhelm your system. Implementing throttling in your code can help manage the flow of these requests effectively.

Using the bulkhead pattern increases your system's resilience and availability. It makes your system easier to manage by isolating different parts. However, adding bulkheads also introduces some complexity. You need to carefully design your system to ensure that the added complexity is justified by the improved fault tolerance.

The bulkhead resilience pattern is a powerful technique for building robust and reliable systems. By creating isolation between different parts of your architecture, you can prevent small failures from cascading and affecting the entire system. Whether you are working with monolithic applications or microservices, the bulkhead pattern can enhance your system's stability and performance.

The bulkhead pattern is a strong architectural method used for designing and building APIs. It helps make systems more resilient, scalable, and able to handle faults. Separating distinct parts of the system keeps failures limited, uses resources efficiently, and ensures the system stays strong even if some parts go down. However, it is important to weigh these benefits against the extra complexity and resource use it can bring. Careful planning, managing resources well, and continuous monitoring are key to successfully using the bulkhead pattern in your API design.

Summary

This chapter covered two important areas for building strong RESTful APIs: data handling and resilience.

The *Data handling* section focused on managing large amounts of data efficiently. Key techniques include pagination, which splits data into smaller pages to improve performance, and filtering, which allows clients to request only the data they need. The chapter also explained how to handle file uploads and downloads securely, ensuring that APIs can manage large files without slowing down. Additionally, we introduced HATEOAS, which adds navigational links to API responses, making it easier for clients to interact with the API.

The *Resilience* section showed us how we can keep APIs stable and reliable under various conditions. Important strategies included setting timeouts to avoid long wait times, using retry mechanisms to handle temporary failures, and implementing rate limiting to prevent abuse and manage traffic. Throttling helps control the flow of requests, while idempotency keys ensure that repeated requests do not cause errors. Advanced patterns, such as circuit breakers, stop failing requests to protect the system, and bulkheads isolate different parts of the API to prevent failures from spreading. By applying these resilience techniques, APIs can maintain high performance and reliability even during unexpected challenges.

Overall, by focusing on effective data handling and robust resilience strategies, developers can create APIs that are both efficient and dependable, capable of supporting growing applications and delivering a consistent user experience.

Now that you have mastered advanced API concepts and implementations, the journey ahead will focus on securing, testing, and deploying your APIs in production environments. *Chapter 7* will guide you through securing your RESTful APIs against common vulnerabilities and attacks, teaching you to implement HTTPS encryption, proper authentication and authorization mechanisms, JWT tokens, and protection against OWASP vulnerabilities and CORS issues. *Chapter 8* shifts focus to comprehensive testing strategies, covering everything from Spring MVC unit testing to integration testing, GenAI-assisted test creation, and contract testing to ensure API reliability.

These upcoming chapters will transform your APIs from functional prototypes into secure, well-tested services ready for production deployment. By the end of *Part 2*, you will have the knowledge to build APIs that not only perform well but also protect user data and maintain reliability under various conditions, preparing you for the final part of the book, which covers deployment and performance optimization.

7

Securing Your RESTful API

Welcome to the part of the book that talks about software security, within the larger field of cybersecurity, which is what you, as a software engineer, are most responsible for. This may seem a little different from what you have read so far because it's not always seen as a core responsibility of a software engineer. That is a sad misconception. With the digitization of our society, software security is no longer just the domain of cybersecurity experts; it's an integral part of software development and should be a major concern at all levels of development.

Understanding security can be complex, so let's start with a mental image. Metaphors help simplify abstract concepts, and one of the most common metaphors in cybersecurity is the onion. If you've read about security before, you've likely encountered this idea: security is built in layers, just like an onion, with each layer adding a protective barrier.

While useful, this metaphor has its flaws. It suggests that every layer is uniform when, in reality, cybersecurity defenses vary significantly. A more accurate comparison is a fortress under siege, where different layers of defense respond to different threats. Long-range artillery targets distant attackers, walls hold back intruders, and hand-to-hand combat is the last line of defense. Likewise, in security, each layer serves a distinct purpose, using different tools and techniques to protect against specific risks.

But what does this mean for you, the software engineer? Security isn't just about protecting systems—it's about safeguarding user data, ensuring reliability, and preventing costly breaches that can damage both a company's reputation and its bottom line. Vulnerabilities in your code can be exploited to steal sensitive information, disrupt services, or even compromise entire networks. As software engineers, we are the first line of defense. Writing secure code, enforcing proper access controls, and understanding common attack vectors are not optional skills—they are essential responsibilities. In this chapter, we'll explore the layers of security you are responsible for and how to integrate security into your development process effectively.

The following topics will be covered in this chapter:

- Anatomy of an HTTP API call
- Authentication
- OWASP API Security Top 10 overview
- Understanding Common Vulnerabilities and Exposures
- Strategies to manage CVEs

Anatomy of an HTTP API call

Almost all API calls today are made over HTTP. So, let's take a look at what happens during such a call and many of the components that participate in it.

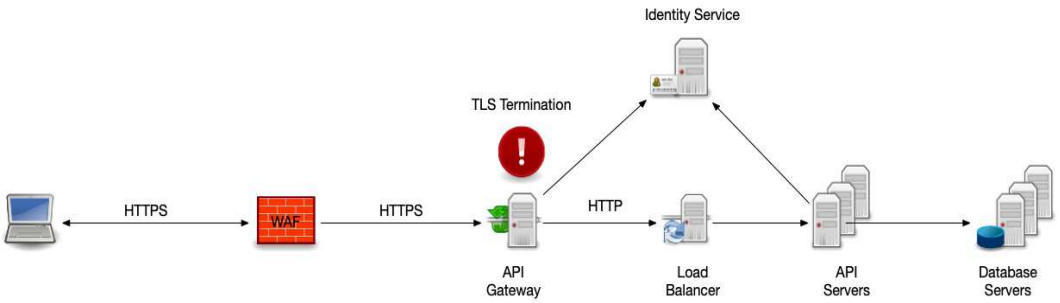


Figure 7.1 – A typical secure API call over HTTP

Encrypting the communication

Although the protocol is called HTTP and we usually say it is an HTTP call or an HTTP connection, the fact is that all modern API calls are made over HTTPS.

The "*S*" at the end stands for "*secure*," indicating that the HTTP communication is encrypted using **Transport Layer Security (TLS)**. TLS operates at the transport layer (Layer 4 in the OSI model), encrypting data between the client and server to prevent eavesdropping and tampering. Using TLS in an HTTP call ensures that only the intended receiver can read the message correctly, practically eliminating man-in-the-middle attacks because even if a message is intercepted, it cannot be decrypted and read.

An HTTPS call will therefore enforce the following:

- **Data confidentiality**: All data transmitted over the network is encrypted and meaningless without the correct set of keys
- **Data integrity**: Ensures the data hasn't been altered during transition
- **Server authentication**: Validates the server's identity to the client

All three features of HTTPS (HTTP over TLS) communication are implemented using cryptographic algorithms to perform encryption and create digital signatures and secure digests (hashes) of the transferred data. How the cryptographic algorithms are used and how their inputs and outputs are transferred over the network are specified by the particular TLS protocol version used.

Some of the algorithms and protocols used in the past are considered vulnerable given the current scientific knowledge and computing power available. The set of algorithms that are considered safe keeps evolving. There are already algorithms available today that are expected to stay safe when we reach the quantum computing era.

Always use strong cryptographic algorithms and protocols when implementing HTTPS for your APIs. Avoid deprecated protocols such as SSL and old versions of TLS. Encryption algorithms are constantly evolving, and old ones have become obsolete due to the ease with which they are broken. Create and enforce a policy to review your encryption selections frequently and update your TLS certificates. We will discuss this later.

In modern organizations (mid-size and up), it is unlikely that a software engineer will be responsible for any TLS implementations, but it is important to have a general idea of how HTTPS works and its implications.

An essential element used in HTTPS communication is the use of public key cryptography. The public part of the public-private key pair is part of a TLS certificate. TLS certificates have a life cycle from issue to expiry or invalidation. Hence, we need to manage these certificates.

Managing your HTTPS certificates

One of the consequences of using HTTPS/TLS encryption is the need for certificate management. Managing security certificates is a complex and crucial part of any API-based application. Effective certificate management involves handling renewals and expirations, and ensuring certificates are correctly configured to prevent service interruptions. Additionally, automated tools for certificate life cycle management can help streamline compliance, especially in environments with frequent communication between APIs or external services. This is such a significant problem that many groups have tried to create mechanisms to mitigate it. Let's Encrypt (`https://letsencrypt.org/`) has achieved great success, having been used even in some large corporations. Most cloud providers also have some form of automatic certificate management, such as AWS Certificate Manager (`https://aws.amazon.com/certificate-manager/`).

The first line of defense – Web Application Firewall (WAF)

A WAF acts as a barrier between the client and your server, filtering out malicious traffic and offering a first layer of protection against common web exploits such as SQL injection and **Cross-Site Scripting** (**XSS**). This is not usually controlled by software engineers, but it is important to know about it as the WAF can cause some interference on HTTP traffic.

These are the functions of a WAF:

- **Traffic monitoring**: Inspects incoming requests for suspicious patterns
- **Rule-based filtering**: Blocks traffic based on predefined security rules
- **Anomaly detection**: Identifies unusual behavior that may indicate an attack

Best practices

Regularly updating WAF rules is essential to ensure that it can adapt to new and emerging threats. Additionally, it's important to customize these rules to align with the specific needs and architecture of your application, ensuring optimal protection. Monitoring WAF logs consistently provides valuable security insights, helping to identify patterns, anomalies, or potential attacks, thereby enabling proactive defenses.

API gateways

An API gateway is usually the entry point for all client requests, handling tasks such as request routing, composition, and protocol translation. In terms of security, an API gateway frequently performs many important tasks:

- **TLS termination**: Decrypts incoming HTTPS traffic
- **Authentication**: Validates credentials and permissions
- **Rate limiting and throttling**: Controls the number of requests to prevent abuse

Benefits

An API gateway centralizes security policies, allowing consistent enforcement across all services. It simplifies client interactions with APIs by providing a single entry point that manages tasks such as routing and authentication. Additionally, it enhances performance by enabling caching and load balancing, ensuring efficient resource utilization and improved response times.

Load balancers

Load balancers distribute incoming network traffic across multiple servers to ensure availability and reliability. While not directly involved in security logic, they contribute to overall security in the following ways:

- **Health monitoring**: Removing unhealthy servers from the pool.
- **TLS termination**: Also handling TLS encryption to reduce server load and system complexity if an API gateway is not present.
- **Mitigating Denial of Service (DoS)**: Distributing the load helps to mitigate DoS attacks. This is often the last resort in a DoS attack. It would be expected that the WAF and the API gateway would have minimized the impact of an attack at this point.

When to decrypt the call — TLS termination

If a secure call is encrypted, it needs to be decrypted at some point so that its content can be interpreted and processed. That is called **TLS termination**. It refers to the process of decrypting an HTTPS connection and converting it back to a clear-text HTTP communication. This typically occurs at the network's edge, such as within an API gateway, a load balancer, or a reverse proxy server. By terminating the TLS encryption at a single point, organizations can offload the computational overhead associated with encryption and decryption from backend servers. This process usually happens only once during the life cycle of an API call, ensuring that data remains encrypted while traversing insecure networks but can be processed more efficiently within trusted internal

networks. TLS termination enhances performance and simplifies certificate management without compromising the security of data in transit over public channels. It is important to be aware of when the TLS termination happens, especially when troubleshooting some problems with an API.

All the security elements that we have briefly discussed so far are undeniably very important, but in modern mid-size and larger organizations, they are usually not the direct responsibility of the software engineer. Other professionals oversee them, and therefore they are not the focus of this chapter. In contrast, the following security elements are very much the responsibility of the software engineer and we all should have a clear understanding of them.

As we have discussed, securing API communications is a multi-layered process involving various tools and technologies, each playing a distinct role in protecting data in transit. From the encryption provided by HTTPS and TLS to the filtering and monitoring capabilities of WAFs, every layer contributes to a robust defense against cyber threats. TLS termination and certificate management further underscore the importance of maintaining both performance and security in modern API ecosystems.

While these responsibilities are often overseen by dedicated teams in large organizations, it is essential for software engineers to understand how these components work and their implications. Having this foundational knowledge allows better collaboration with security teams and DevOps engineers to troubleshoot issues effectively when they arise and to make better decisions during design and development. With security threats constantly evolving, maintaining an awareness of these practices ensures APIs remain resilient and reliable, forming a secure backbone for any application.

Authentication

Authentication is the foundational process of verifying the identity of a user, device, or other entity attempting to access your API. Authentication is a crucial element in security because it ensures that only legitimate and recognized parties can interact with your services. Without proper authentication mechanisms, APIs become vulnerable to unauthorized access, data breaches, and malicious activities that can compromise system integrity and user data.

The primary goal of authentication is to establish the identity of the client requesting access. By confirming it, the API can enforce access controls, apply rate limiting, and provide personalized experiences where appropriate. Moreover, robust authentication mechanisms are essential for compliance with legal and regulatory requirements, such as the GDPR, HIPAA, LGPD, and other data protection laws.

Different authentication methods can be employed depending on the level of security required, the nature of the application, and user experience considerations. Next, we will break down some of the most common methods.

Password-based authentication

Password-based authentication is the most traditional and is still one of the most widely used methods of authentication. Users provide a username and password, which are then verified against credentials stored on the server. By its nature, this method is mostly used when a person is authenticating with a system. It is hardly ever used in any form of machine-to-machine interaction.

How it works

When the user attempts to log in, they send their credentials, typically a username and password, to a security service API. Upon receiving the credentials, the server hashes the password and compares it with the stored hash in its database. If the credentials match, the user is successfully authenticated, and the server generates an authentication token, such as a session cookie or a **JSON Web Token** (**JWT**), to facilitate future requests without the need for repeated credential submission.

Security considerations

Passwords should never be stored in plaintext; instead, they must always be hashed using a secure algorithm. The hashing process is only as good as the algorithm used. Two modern and widely used ones are *Bcrypt* and *Argon2*. A deeper discussion on hashing algorithms is not in the scope of this chapter, but it is highly recommended to look at how they work.

Another consideration is brute-force attacks. It's important to implement account lockout mechanisms after a certain number of failed login attempts. Additionally, users should be required to create strong passwords by enforcing password policies that include minimum length requirements, the use of special characters, and the avoidance of common or easily guessable words.

Here are the advantages of password-based authentication:

- Simple to implement
- Easy for users to understand

And here are the disadvantages:

- Passwords can be easily stolen or guessed, especially if they are weak
- It requires robust security measures such as rate limiting and password resets
- Password reuse across different services makes accounts more vulnerable

Token-based authentication

Token-based authentication relies on the use of tokens—small pieces of data passed between the client and server to verify identity. A widely adopted token format is JWT, and this is the format we will discuss.

It is important to understand how a JWT works, specifically the lifespan of a token.

JWT life cycle and utilization

The user logs in with their credentials, and the server generates a token (often a JWT) containing encoded information about the user.

The server will create and sign the JWT, ensuring its integrity, and send it back to the client. The internal mechanism of signing and creating a JWT is beyond the scope of this book. See this article for a more complete discussion: `https://www.freecodecamp.org/news/how-to-sign-and-validate-json-web-tokens/`.

From that moment on, the client will include the JWT in the header of all future API requests, typically in a pre-defined header defined by the API development team.

The server validates the token on each request without needing to check the user credentials again.

One important point to understand is that all tokens (JWT or not) must be temporary. They must expire at clear and well-defined intervals. As a software engineer working on the API layer, you usually don't have to do anything special to handle this, but frontend applications will have to define a token renewal strategy, and you may have to collaborate and support such tasks.

Structure of a JWT

A JWT is an encrypted record that follows a well-defined structure. So, let's look inside a JWT and discuss its structure.

A good tool for viewing and manipulating JWTs can be found at `https://jwt.io/#debugger-io`.

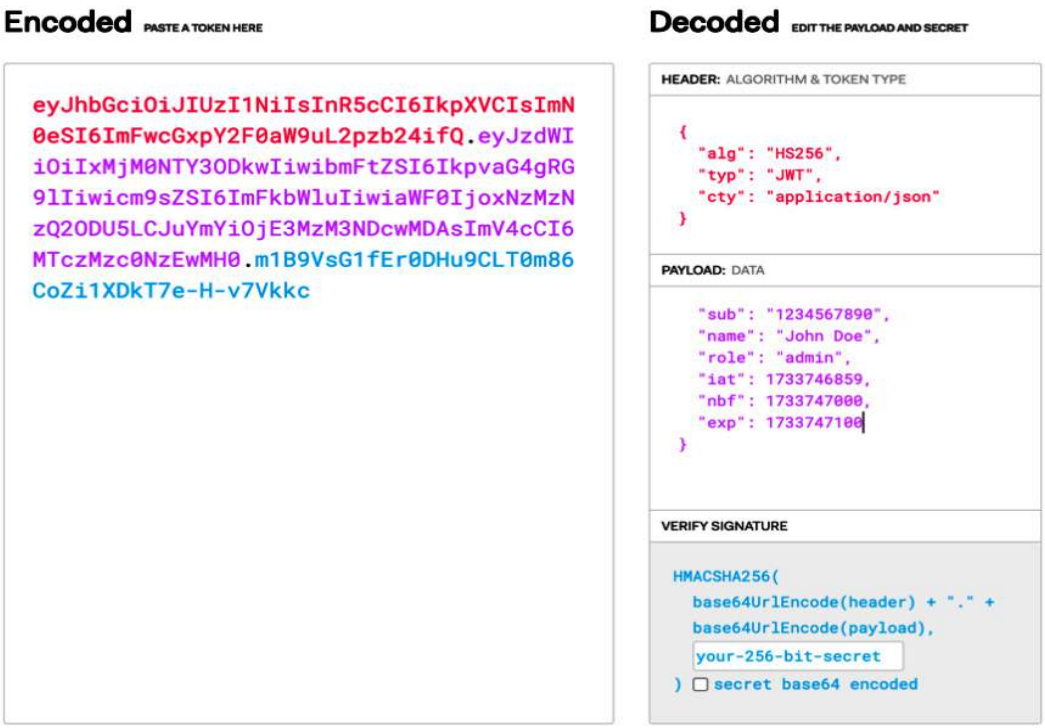


Figure 7.2 – Example JWT

A JWT consists of three distinct parts separated by a dot: the header, payload, and signature. As an API developer, your focus will be on the payload, as it contains claims (fields) about the user (e.g., user ID, permissions). The signature ensures the token hasn't been tampered with.

Once decrypted, the data inside a JWT looks like this:

Part	Field	Description	Example Value
Header	`alg`	Algorithm used to sign or encrypt the token	`"HS256"`
Header	`typ`	Type of token	`"JWT"`
Header	`cty`	Content type of the JWT payload	`"JWT"` or `"application/json"`
Header	`kid`	Key ID used to identify the key used in signature	`"abc123"`
Payload	`iss`	Issuer of the token	`"auth.example.com"`

Part	Field	Description	Example Value
Payload	`sub`	Subject of the token (usually a user ID)	`"1234567890"`
Payload	`aud`	Audience for which the token is intended	`"example.com"`
Payload	`exp`	Expiration time (in Unix time)	`1516239022`
Payload	`nbf`	Not valid before this time (in Unix time)	`1516239022`
Payload	`iat`	Issued at time (in Unix time)	`1516239022`
Payload	`jti`	JWT ID; a unique identifier for the token	`"unique-token-id-123"`
Payload	`name`	User's full name	`"John Doe"`
Payload	`email`	User's email address	`"john.doe@example.com"`
Payload	`role`	Roles or permissions assigned to the user	`"admin"` or `["admin", "user"]`
Signature	N/A	Digital signature for verifying the token's integrity	(Result of cryptographic operation)

Table 7.1 – An example JWT

The header can be customized to adjust to the needs of each organization. But at a minimum, you should see `"alg"` and `"typ"` entries present.

The payload is also mostly defined by the organization. Each entry in the payload is called a **claim** and is used to identify the users, their roles, unique information, and any other data element that can be used by the APIs to best control access to the data.

Special attention should be given to the three time-based claims – `exp`, `nbf`, and `iat`. They are crucial to verify whether the JWT is valid at the time of execution of the API. Ideally, they should not be omitted, although `"nbf"` is sometimes not defined. In that case, it is assumed that the JWT is valid from the moment it is created (`"iat"`).

In general, there are three classifications for claims:

- **Registered claims**: Predefined claims widely used in the industry, such as `iss`, `exp`, and `sub`

- **Public claims**: Custom claims agreed upon by parties using the JWT
- **Private claims**: Custom claims used in contexts specific to your application

Statelessness

One of the key benefits of token-based authentication is that the server doesn't need to store session state validation information. All necessary information is contained in the token itself and is sent over in each call.

Frequently, it is necessary to store session state on the server side, but no credentials or user identification should be stored, only references to the tokens used.

Security considerations

JWTs are part of a general security strategy, and as such should be treated as data to be secured in themselves. Therefore, here are some specific security concerns about the JWTs:

- Always use HTTPS to transmit tokens, as tokens are susceptible to interception in plaintext
- Set token expiration times to limit the window of vulnerability
- Use short-lived tokens with refresh tokens to minimize the risk of token theft
- Securely store tokens on the client side, avoiding local storage if possible

These are the advantages of a JWT strategy:

- Stateless authentication scales well in distributed systems
- There's no need to store session information on the server
- It allows granular control over access rights using token claims

And these are some disadvantages:

- Tokens, if stolen, can be used to impersonate the user
- Managing token expiration and secure refresh cycles will add complexity to the application, both the frontend and the backend

Caveats

One problem that sometimes happens is JWT inflation, where the organization keeps adding more claims or long lists of values to claims to the point that the JWT grows beyond the limits defined on the server. Even if the JWT specs do not define an upper limit, remember that the token will be sent over as part of the HTTP header, and there is always a limit for them. If you start receiving **HTTP 413 Content Too Large** errors, one possible culprit could be the size of your JWT.

In summary, using JWTs to manage authentication and authorization is a well-established strategy that has good industry support and relies upon solid technologies. However, a large portion of its effectiveness is due to how it is used by the application. It is your responsibility as the software engineer to make sure good and well-managed policies are put in place.

Multi-factor authentication

Multi-factor authentication (**MFA**) adds an additional layer of security by requiring the user to provide more than one form of identification. This typically includes something they know (e.g., password), something they have (e.g., mobile phone or security token), or something they are (e.g., fingerprint or face recognition). This is mostly a client-side strategy and has little effect on API development, but it is important for backend engineers to understand its general concepts.

MFA is part of the authentication dialog, so, after a user has successfully entered their password, they are prompted to provide a second factor, such as a code generated by a mobile app (e.g., Google Authenticator) or sent via SMS. The server then verifies the second factor before granting access, usually by generating a JWT to be used by the client when accessing the APIs.

What can be used in MFA?

Generally speaking, there are three types of MFA:

- **Something you know**: A password, PIN, or answer to a security question
- **Something you have**: A physical device such as a phone or hardware token that generates a temporary passcode
- **Something you are**: Biometric data such as a fingerprint, facial recognition, or iris scan

Security considerations

Let's see what we should consider with regard to security when using MFA:

- MFA drastically reduces the likelihood of account compromise, even if a password is stolen.
- **Time-based one-time passwords** (**TOTPs**) are commonly used for MFA. Ensure that those tokens are properly time-synchronized and expire after a short period.
- SMS-based MFA is vulnerable to SIM swapping and should be avoided where possible. Use app-based authenticators or hardware tokens instead.

Let's look at the advantages of MFA:

- It significantly enhances security by requiring multiple forms of verification
- It protects against common attacks such as password phishing and brute-force attacks

And let's see some of the disadvantages:

- It can be cumbersome for users, especially in environments with frequent logins
- It requires infrastructure to support the second factor, such as maintaining SMS gateways or integrating with an authenticator app

Biometric authentication

Biometric authentication uses physical characteristics like fingerprints, facial recognition, or voice to verify a user's identity. With the increasing availability of biometric sensors on mobile devices and laptops, it's becoming a more common method of authentication.

Let's see how it works:

1. The user's biometric data (fingerprint, face scan, etc.) is captured and stored securely on the device.
2. When the user attempts to authenticate, the API compares the captured data with the stored biometric template to verify their identity.

Here are some common biometric authentication methods:

- **Fingerprint scanning**: Used extensively on mobile devices
- **Facial recognition**: Built into many smartphones and laptops
- **Voice recognition**: Sometimes used for telephonic authentication

Let's see some of the security considerations:

- Biometric data is unique to each user, making it difficult to forge
- Store biometric data locally, not on the server, to prevent mass breaches
- Ensure biometric systems have fallback options (such as passwords) for cases where biometrics fail (e.g., injury or hardware malfunction)

These are some advantages of biometric authentication:

- It provides a high level of security
- It is convenient for users—there's no need to remember passwords

Here are some of the disadvantages:

- Biometric data is irreversible—once compromised, it cannot be changed
- There are potential privacy concerns around the collection and storage of biometric data
- It is less reliable in certain conditions (e.g., wet fingerprints or poor lighting for facial recognition)

Choosing the right authentication method depends on the application's requirements and the sensitivity of the data it handles. While password-based authentication remains prevalent, more secure methods, such as token-based authentication, MFA, and biometric authentication, are rapidly gaining ground. Combining these methods, such as using MFA in conjunction with JWTs, can provide a robust and flexible authentication mechanism for APIs, ensuring the security of both users and data.

Authorization

Once you have your API guarded against unknown users by authentication, the next step is to apply authorization mechanisms. This is where the system determines whether an authenticated entity (user, device, or service) has permission to access a specific resource or perform a particular action. While authentication verifies identity, authorization governs access, ensuring that the authenticated entity only has access to the resources and operations for which they have been granted permission.

In API security, proper authorization is essential for enforcing business rules, safeguarding sensitive data, and ensuring that each user or service only performs the actions and accesses the information they are allowed to. So, let's look at the different techniques we can use to implement authorization.

Role-based access control

One of the most common authorization mechanisms is **role-based access control** (**RBAC**). RBAC works by assigning predefined roles to users or entities and associating permissions with those roles. Each role is granted access to specific API endpoints or functionalities, depending on the application's security requirements.

Let's see how RBAC works:

- Users are assigned one or more roles, such as admin, user, or editor.
- API endpoints or resources are protected by role checks. For example, only users with the admin role might have access to the /admin endpoints.
- When an API request is made, the server checks the user's assigned roles against the roles required for the requested resource.

These are some of the advantages of RBAC:

- It simplifies permission management by grouping permissions into roles
- It is easy to implement and scale, particularly in enterprise applications where roles can be defined centrally
- It is fully supported by many solutions, such as the claims in JWT

Here are some of the disadvantages:

- It can be inflexible in complex systems where fine-grained access control is required, leading to role explosion or data leaks
- It can cause unexpected exposure in complex systems, as roles may be inconsistent or incompatible with each other

Attribute-based access control

Attribute-based access control (**ABAC**) provides more granular control than RBAC by using attributes (user attributes, environment attributes, resource attributes, etc.) to determine access rights. Attributes are metadata that describe entities or actions and are evaluated against policies to decide whether access is allowed.

Let's see how ABAC works:

- Policies are written to define the conditions under which access is granted.
- Attributes such as user role, time of request, resource type, or location are considered.
- Access decisions are made dynamically, based on real-time evaluation of attributes and policies.

These are some of the advantages of using ABAC:

- It provides flexibility by allowing fine-grained control based on dynamic conditions
- It is scalable for complex systems in which RBAC would become difficult to manage

And here are some disadvantages:

- It is more complex to implement and manage than RBAC.
- It can result in performance overhead due to real-time policy evaluations.
- It can cause unexpected security issues due to mismanaged attributes. For example, an employee changes positions in the company but the system still allows them to see data from the previous job.

OAuth 2.0

OAuth 2.0 is one of the most widely used authorization frameworks for APIs. It allows a user to grant a third-party application limited access to their resources without exposing their credentials. OAuth 2.0 is commonly used for delegating access to APIs and provides a standardized method for authorization across web, mobile, and cloud applications.

Let's see how OAuth 2.0 works:

1. The user authenticates with an identity provider (e.g., Google, Facebook) and grants permission to a third-party application.
2. The third-party application receives an access token, which it uses to make authorized API requests on behalf of the user.
3. The access token contains scopes, which define the level of access the third-party application has (e.g., read-only, write).

Let's see some security considerations:

- Always use HTTPS to transmit OAuth tokens
- Use short-lived access tokens and refresh tokens to minimize risk in case of token theft
- Implement token revocation mechanisms to invalidate tokens if necessary

Here are the advantages of using Oauth2.0:

- It allows delegated access without sharing user credentials
- It has been widely adopted, providing standardization across many services

And here are some disadvantages:

- It requires the implementation of secure token storage and transmission mechanisms
- It can add complexity, especially when managing refresh tokens and token expiration

JWT for authorization

JWTs, which were discussed in the *Authentication* section, also play a key role in authorization. After authentication, a JWT can be issued containing claims that specify the user's roles or permissions. These claims can then be used by the API to enforce access controls.

Let's see how JWT authorization works:

1. The user is authenticated, and a JWT is issued containing claims such as user roles or permissions.
2. The client includes the JWT in the authorization header in future API requests.
3. The server decodes the JWT and checks the claims to determine whether the user is authorized to access the requested resource.

JWTs allow stateless authorization, meaning the server doesn't need to maintain session state because the necessary authorization information is embedded in the token itself.

Let's see some of the advantages of using JWT authorization:

- It scales well in distributed systems by eliminating the need for server-side session storage
- It enables fine-grained control using claims inside the JWT

As for disadvantages, it requires careful management of token expiration and revocation to prevent unauthorized access. This is the biggest drawback of JWT and any other attribute-based authorization. It is important to ensure the claims in the JWT reflect the actual roles of this user. Once again, the case of an employee changing positions in the company comes to mind, potentially causing data access violations.

Fine-grained access control

In some systems, the level of access control required goes beyond the basic capabilities of RBAC or even ABAC. **Fine-grained access control (FGAC)** allows more nuanced control over access to resources, based on attributes such as user identity, resource ownership, and contextual data.

These levels of FGAC are usually found in complex business domains; healthcare and finance are two very common cases. Healthcare, being a highly regulated industry, has many different rules that need to be applied to any data access.

Here are some examples of FGAC:

- A user can access only their own profile data, not other users' data
- A service can only modify resources that it has created
- Access to sensitive resources might be restricted based on location, time of day, or other contextual information
- A customer on a healthcare insurance system can see all payments made to their account but can only see claims payments for their claims and other dependents that they are legally allowed to

Let's take a closer look at that last example, as it illustrates the challenges of FGAC.

Case: A client is required to keep their ex-spouse on their healthcare policy for a specified time. During this time, the main client cannot see any of the spouse's medical information, but can see the costs associated with their treatment. There are several challenges in this case. Where do we look for the mandatory amount of time to keep the spouse on the policy? How do we isolate the spouse's data in a family request? How do we allow some financial data to be exposed but not medical information?

Those are not simple questions to answer and may make that piece of the system more complex than it would be otherwise.

This case requires careful design and policy enforcement. It could increase the complexity of the application, particularly in large systems.

Best practices for API authorization

In summary, here are some guidelines to make sure your API is as secure as you can make it:

- **Use the principle of least privilege**: Only grant users and services the minimal permissions necessary to perform their tasks
- **Implement layered authorization mechanisms**: Combine RBAC with ABAC or OAuth for more flexible and secure access control
- **Regularly audit permissions**: Periodically review and update roles, permissions, and access policies to ensure they align with current business needs
- **Secure tokens**: Always use secure storage and transmission methods for tokens (e.g., OAuth tokens, JWTs) to prevent unauthorized access

Authorization, when designed and implemented effectively, ensures that your API remains secure, responsive, and compliant with regulatory requirements. By choosing the right authorization strategy—whether RBAC, ABAC, OAuth, or JWT-based—you can tailor access control to your system's specific needs while protecting data and ensuring users can only perform the actions they are entitled to.

OWASP API Security Top 10 overview

OWASP stands for **Open Web Application Security Project**, a non-profit organization that focuses on web application security. They do a phenomenal job at documenting known security issues and helping developers and organizations to secure web applications. One of the main contributions is the **OWASP API Security Top 10**, which identifies the most critical security risks for APIs.

OWASP is a crucial resource for all software engineers, and you should periodically look at the published lists of concerns and new security white papers.

The most recent list was published in 2023 and can be found here: `https://owasp.org/API-Security/editions/2023/en/0x11-t10/`.

- API1:2023 - Broken Object Level Authorization
- API2:2023 - Broken Authentication
- API3:2023 - Broken Object Property Level Authorization
- API4:2023 - Unrestricted Resource Consumption
- API5:2023 - Broken Function Level Authorization
- API6:2023 - Unrestricted Access to Sensitive Business Flows
- API7:2023 - Server Side Request Forgery
- API8:2023 - Security Misconfiguration
- API9:2023 - Improper Inventory Management
- API10:2023 - Unsafe Consumption of APIs

As you can see, all the top OWASP concerns are the topics we have been discussing in this chapter so far. It is important to be well acquainted with the OWASP list as it evolves, so visit frequently the OWASP Top Ten API Security Risks at the link provided previously.

Understanding Common Vulnerabilities and Exposures

The phrase "our society runs on software" is not new and describes most modern societies, and consequently, software security has emerged as a core concern for governments and organizations worldwide. Cyber threats are evolving at an unprecedented pace, and vulnerabilities in software can lead to significant financial losses, reputational damage, and legal repercussions. In the same week as this chapter was written, the largest DDoS attack in history targeted Cloudflare (you can find a detailed discussion here: `https://www.linkedin.com/pulse/cloudflare-mitigates-historic-world-record-56-tbps-hoccc/`). It is suspected that the scale of this attack was achieved by exploiting vulnerabilities in many IoT devices, but the details are still being discovered. Therefore, for Java software engineers, understanding and managing these vulnerabilities is fundamental for developing secure and robust applications.

One of the fundamental aspects of software security is the identification and mitigation of known vulnerabilities, commonly referred to as **Common Vulnerabilities and Exposures (CVEs)**. This chapter aims to provide an in-depth exploration of CVEs from the perspective of a professional Java software engineer. We will delve into the definition of CVEs, the mechanisms used to discover and document them, and their structural composition. Additionally, we will highlight the best resources for tracking CVEs and discuss software scanners that help identify these vulnerabilities.

Furthermore, we will examine effective strategies for managing CVEs, including continuous dependency upgrades, dependency management practices, proactive dependency updates, and reactive upgrades based on scan reports. By understanding and implementing these strategies, software engineers can enhance the security posture of their applications and contribute to a safer digital ecosystem.

What are CVEs?

The CVE system is a standardized method for identifying and cataloging security vulnerabilities and exposures in software and firmware. Managed by the MITRE Corporation, CVE provides a unique identifier for each disclosed vulnerability, allowing consistent references across various security tools, databases, and communications. A CVE entry typically includes an identification number, a brief description of the vulnerability, and references to related advisories, reports, and patches.

For Java software engineers, CVEs are particularly significant due to the widespread use of Java libraries and frameworks, which can introduce vulnerabilities into applications if not properly managed. Understanding CVEs enables developers to do the following:

- **Identify vulnerabilities**: Recognize known security issues in the libraries and components they use
- **Assess impact**: Evaluate the potential impact of a vulnerability on their applications
- **Implement mitigations**: Apply patches or updates to address vulnerabilities promptly
- **Maintain compliance**: Adhere to security policies and regulatory requirements by managing known vulnerabilities

How are CVEs found and documented?

The discovery of vulnerabilities is a collaborative effort involving security researchers, developers, and automated tools. The following are common mechanisms for finding vulnerabilities:

- **Security research**: Security professionals and researchers actively analyze software to identify potential vulnerabilities. This process may involve the following:
    - **Code auditing**: Manually reviewing source code for insecure coding practices
    - **Penetration testing**: Simulating attacks to find weaknesses in running applications
    - **Fuzz testing**: Providing invalid or random data to programs to trigger unexpected behavior
- **Bug bounty programs**: Many organizations offer bug bounty programs, incentivizing independent researchers to find and report vulnerabilities. These programs help uncover issues that might be missed by internal teams.
- **Automated scanning tools**: Tools such as static code analyzers and **dynamic application security testing (DAST)** tools automatically scan code bases and running applications for known vulnerability patterns.
- **User and developer reports**: End-users and developers may encounter security issues during regular use or development activities and report them to the maintainers or security teams.

Documentation process

Once a vulnerability is discovered, documenting it involves several steps:

- **Vulnerability disclosure**: The finder reports the vulnerability to the software vendor or maintainers, following responsible disclosure practices to allow time for a fix before public disclosure.
- **CVE assignment**: An authorized **CVE Numbering Authority (CNA)** assigns a unique CVE identifier to the vulnerability. This identifier allows consistent tracking and reference.
- **Description and analysis**: A detailed description of the vulnerability is created, including the following:
    - **Summary**: A brief overview of the issue
    - **Technical details**: In-depth information about the vulnerability, including affected versions and components

    - **Impact assessment**: The potential risks and consequences if the vulnerability is exploited

- **Publishing**: The CVE entry is published in the CVE database and shared with various security databases such as the **National Vulnerability Database (NVD)**. Vendors may also release security advisories and patches.

Structure of a CVE

A CVE entry consists of several key components that provide essential information about the vulnerability:

- **CVE identifier**: A unique identifier in the format CVE-YYYY-NNNN, where YYYY is the year the CVE ID was assigned or made public and NNNN is a sequential number, for example, CVE-2023-12345.
- **Description**: A concise summary of the vulnerability, including the following:
    - **Type of vulnerability**: Such as buffer overflow or injection flaw
    - **Affected products**: Specific software, versions, and components impacted
    - **Impact**: The potential effects of exploitation, such as unauthorized access or denial of service
- **References**: Links to additional information, including the following:
    - **Vendor advisories**: Official statements or patches from the software vendor
    - **Security Bulletins**: Reports from security organizations
    - **Technical Analysis**: In-depth articles or reports analyzing the vulnerability
- Finally, when imported into the NVD, additional metadata is added (if available):
    - **CVSS scores**: The **Common Vulnerability Scoring System (CVSS)** provides a quantitative measure of the severity of the vulnerability.
    - **Impact metrics**: Details about confidentiality, integrity, and availability impacts.
    - **Affected configurations**: Specific environments or configurations that are vulnerable.

You need to understand that documenting a CVE is a process, and a CVE that affects your code may have been identified, but no current solution is yet available.

Best resources to track CVEs

Staying informed about the latest vulnerabilities is essential for proactive security management. The following are some of the best resources for tracking CVEs. You should bookmark each of them and become familiar with their structure and searching capabilities:

NVD

Managed by the **National Institute of Standards and Technology** (**NIST**), the NVD provides comprehensive information about CVEs, including detailed analyses, CVSS scores, and impact metrics. It is the main resource for CVEs:

`https://nvd.nist.gov/`

MITRE CVE list

The official CVE list is maintained by MITRE. It provides basic information about each CVE, including descriptions and references. Frequently, the information here is easier to navigate than the NVD:

`https://cve.mitre.org/`

Security advisories from vendors

Software vendors often publish their own security advisories and patches. Monitoring these sources ensures you receive vendor-specific information promptly. There are many such resources, but for a Java software engineer, Oracle and Apache are two of the most important:

- **Oracle Java SE Critical Patch Updates**: `https://www.oracle.com/security-alerts/`
- **Apache Security**: `https://apache.org/security/`
- **VulDB**: `https://vuldb.com/`

These databases aggregate vulnerability information from various sources and often provide additional context and analysis. Not all current vulnerabilities may be available there yet.

Software scanners that help to identify CVEs

Software scanners are tools that have been designed to automatically detect vulnerabilities within code bases, applications, or systems. For Java developers, these tools are invaluable in identifying CVEs associated with dependencies and code. They work in a few different forms and, depending on the project, you may have to use more than one to cover your code.

- **Static Application Security Testing (SAST) tools**: These tools analyze source code for security vulnerabilities without executing the code. Examples are SonarQube and Checkmarx
- **Software Composition Analysis (SCA)**: These tools identify open source components and dependencies in your application and check for known vulnerabilities. Examples include OWASP Dependency-Check, Snyk, Black Duck, and WhiteSource.
- **Dynamic Application Security Testing (DAST)**: These tools analyze running applications to detect vulnerabilities by simulating attacks. Examples include OWASP ZAP and Burp Suite.

How these tools work

As CVEs are security problems with external dependencies, how can the scanners look at your code and find the issues?

First, the scanner analyzes project files, such as `pom.xml` for Maven and `build.gradle` for Gradle, to identify all direct and transitive dependencies. It then extracts the version numbers of each dependency and compares them against a vulnerability database containing CVE information. Based on this comparison, the scanner generates a detailed report highlighting any vulnerable dependencies, classifying them by severity levels according to CVSS scores. Additionally, it provides upgrade recommendations, suggesting versions that resolve the identified vulnerabilities.

The next step is to integrate the dependency scanners into the development workflow. Some scanners provide plugins for **Integrated Development Environments (IDEs)** such as Eclipse and IntelliJ IDEA, allowing developers to detect vulnerabilities directly within their coding environment. Additionally, scanners can be incorporated into **Continuous Integration/Continuous Deployment (CI/CD)** pipelines, automating vulnerability detection during the build process. To enhance security monitoring, they can also be configured to send automated alerts whenever new vulnerabilities are identified. By identifying and addressing security issues early in the development cycle, these integrations enhance agility and help shorten the **Software Development Life Cycle (SDLC)**, reducing delays caused by late-stage security fixes.

Popular scanners for Java

It is always a little dangerous to publish lists of software tools in a printed format as some projects may disappear or become obsolete over time, but as of April 2025, these are the most widely used tools for identifying vulnerabilities in Java projects:

- **OWASP Dependency-Check**: This open-source tool identifies a project's dependencies and checks whether there are any known, publicly disclosed vulnerabilities (CVEs) associated with the scanned dependencies. It supports projects that use Maven, Gradle, and Ant and can be integrated into most build tools and CI/CD solutions.
- **Snyk**: This is a commercial tool that provides vulnerability scanning and remediation for open source dependencies. It provides real-time scanning, allowing the continuous monitoring of dependencies. It also proposes code fixes automatically with suggestions directly in pull requests. Plugins are available for most popular IDEs (such as IntelliJ and VS Code).
- **Black Duck**: This is a comprehensive SCA tool that scans for open source vulnerabilities and license compliance. It enforces policies regarding the use of open source components in the system and generates in-depth analysis of vulnerabilities and risks.

Strategies to manage CVEs

All these tools are very important and are a crucial part of managing the CVEs in your project. But how can we manage this workload, and what considerations are needed?

The effective management of CVEs involves not just detection but also strategic planning and processes to mitigate risks.

One important point to understand is that writing secure software is expensive, and managing CVEs in particular has become a hard and expensive part of software security. It is not uncommon for teams to allocate 15% to 20% of their capacity to security.

With that in mind, let's look at a couple of strategies that help Java developers maintain secure applications.

Continuous upgrade dependencies

Regularly updating dependencies to their latest versions ensures that applications benefit from security patches and improvements made by library maintainers.

This is a proactive approach to CVEs. By always keeping your dependencies up to date, you guarantee that any security update will be implemented and used in your code as soon as it is available.

There are various ways to implement continuous upgrades:

- **Automated updates**: Use tools such as Dependabot to automatically create pull requests for dependency updates
- **Regular maintenance schedule**: Establish a routine (e.g., weekly or monthly) to review and update dependencies
- **Testing**: Implement comprehensive automated tests to ensure that updates do not introduce regressions
- **Monitoring release notes**: Keep track of dependency release notes for any security-related updates

These are the benefits of continuous upgrades:

- **Security**: It reduces the window of exposure to known vulnerabilities
- **Feature enhancements**: It provides access to new features and performance improvements
- **Community support**: It aligns with the active versions supported by the community

Costs and requirements

The main cost with this approach is the time that must be invested in keeping the code base current. Remember, you will be changing dependencies in your code, and the API you are calling may have changed, sometimes in ways that are not immediately obvious, forcing you and your team to refactor portions of your code that would not need any change otherwise.

To be able to make so many potential breaking changes so frequently, it is necessary to have a comprehensive set of automated tests that can give the developer team the confidence to make those changes in the first place.

The time to add the tools and integrate them with the specific ticketing system used in your organization is another cost that needs to be considered. The initial investment can be quite large for legacy code bases with not enough test coverage and automated build pipelines.

Dependency management

Now that we know that CVEs are caused by the dependencies we as software engineers add to the code base, what can we do to limit our exposure to them?

Clearly, a proper policy for the management of dependencies is crucial to control the quality, security, and performance of your application. The goal here is to prevent exposure to CVEs as much as possible.

Sometimes, software engineers add dependencies to projects without analyzing the consequences. To be honest, this happens more often than we'd like to admit! Having a policy in place that will require validation of the need for this new dependency and a discussion about more secure and better-maintained alternatives can significantly help with security vulnerabilities. To avoid introducing unnecessary risks, it is important to follow these best practices when adding and managing dependencies:

- **Minimal dependencies**: Include only necessary dependencies to reduce the attack surface. The introduction of new dependencies must be discussed with the team and its implications must be identified.
- **Version pinning**: Specify exact versions of dependencies to ensure consistent builds. Avoid having dependencies on the "latest" version.
- **Use trusted sources**: Retrieve dependencies from reputable repositories (e.g., Maven Central). Choose packages and libraries that are actively maintained and have a good history of security updates.
- **Transitive dependency control**: Monitor and manage transitive dependencies brought in by direct dependencies.

License compliance: Ensure that the licenses of dependencies are compatible with your project.

Proactive dependency upgrade

As much as we may want to avoid dependencies, it is nearly impossible to build a modern API system without them. So, proactively upgrading dependencies before vulnerabilities are disclosed or before they become outdated is a good way to keep your system CVE-free. To achieve this, it is necessary to do the following:

- **Stay informed**: Subscribe to mailing lists and monitor updates from dependency maintainers
- **Run beta tests**: Test new versions in a controlled environment before full integration
- **Contribute to dependencies**: Engage with open source projects you use the most to understand upcoming changes
- **Automate the scanning**: Use tools that notify you about new releases of dependencies

By doing all of this, you will stay ahead of most CVEs, which reduces the likelihood of being affected by new vulnerabilities. As secondary gains, these policies eliminate large jumps in dependency versions that may require significant refactoring of your own code and all the performance improvements and optimizations of newer versions.

The drawback is the cost. Implementing these proactive policies can cause frequent disruptions that can be hard to justify to business owners and investors because there are no new features being created by all the work.

Reactive upgrade based on scan results

This is the process of upgrading dependencies in response to vulnerability reports from scanners or security advisories. This is probably the most common process used by development teams. To effectively manage upgrades based on scan results, the teams should follow the following steps:

- **Scan regularly**: Use automated scanners to detect vulnerabilities in dependencies
- **Assess severity**: Prioritize vulnerabilities based on severity and impact
- **Plan remediation**: Schedule upgrades or apply patches for affected dependencies
- **Validation**: Test the application after upgrades to ensure functionality is intact
- **Documentation**: Keep records of vulnerabilities found and actions taken
- **Risk management**: Balance the urgency of upgrades with the potential impact on the application
- **Communication**: Coordinate with stakeholders to schedule necessary updates
- **Fallback plans**: Have strategies in place in case an upgrade introduces issues

There is nothing inherently negative about this process, but there are risks that need to be understood.

It is easy to have too many CVEs to mitigate at the same time. When the team decides to delay the security tasks, they can accumulate just before a release. This is a very common case and can cause important delays to a release.

Summary

The ever-evolving landscape of cybersecurity necessitates that Java software engineers remain vigilant and proactive in managing vulnerabilities within their applications. CVEs play a critical role in identifying and communicating known security issues, and understanding how to find, document, and remediate them is essential.

By leveraging tools such as vulnerability scanners and implementing strategies such as continuous dependency upgrades, proactive updates, and reactive measures based on scan reports, developers can significantly enhance the security posture of their applications. Effective dependency management further reduces risks by controlling the components integrated into the software.

Staying informed through reputable sources such as the NVD and vendor advisories ensures that developers are aware of the latest threats and can act promptly. Integrating security practices into the development life cycle transforms security from a reactive afterthought into a proactive cornerstone of software development.

In summary, managing CVEs is a multifaceted process that requires diligence, strategic planning, and the right tools. By embracing these practices, Java software engineers can contribute to creating more secure applications, protecting both their organizations and the end-users who rely on their software. In the next chapter, we will look at how generative AI can revolutionize the way we write tests for APIs, providing insights and techniques

8

Testing Strategies for Robust APIs

While researching and drafting this chapter, we had a completely different vision in mind. The original plan was to write a traditional piece on testing APIs, a structured, methodical exploration of established practices. The focus was going to be on the tools and techniques that have been staples of **API testing** for years, such as Postman, Bruno, and various mocking frameworks. These tools have been essential in helping engineers ensure API reliability and are still widely used, particularly in projects with legacy systems.

However, the more we thought about the topic, it became clear that the landscape of software testing has undergone a significant shift. The rise of **generative AI** and **large language models (LLMs)**—such as ChatGPT, Gemini, and similar tools—has fundamentally transformed how we approach software development, including testing. What was once a slow and often repetitive process has now evolved into a dynamic, AI-enhanced workflow. Generative AI doesn't just make writing tests faster; it makes the entire process more engaging, efficient, and effective.

This transformation isn't just about convenience. The integration of generative AI into API testing has far-reaching benefits for everyone involved in the software development lifecycle. Engineers can now focus on higher-level problem solving rather than being bogged down by tedious test creation. Product managers and owners gain greater confidence in the quality of their APIs, knowing that testing has become more thorough and adaptive. The results are more meaningful, ensuring that APIs meet real-world needs while maintaining the highest quality standards.

This chapter, therefore, is not the one originally set out to be written. We have shifted focus entirely to reflect this new reality. Here, we'll explore how generative AI can revolutionize the way we write tests for APIs, providing insights and techniques to help you leverage this game-changing technology. By embracing these advancements, we can create better APIs, foster collaboration, and deliver software that exceeds expectations.

In this chapter, we're going to cover the following topics:

- Types of tests
- Test format and tooling
- Prompt engineering for testing
- Preparing a development environment
- Running and evolving the code

Technical requirements

One of the advantages of focusing on these new ideas for integration and regression tests is in tools and installations. If you can run a JavaJUnit test on your local machine, you already have everything needed to run a test.

We will also utilize Docker to deploy our applications and dependencies locally.

You will also need access to an LLM. Any of the freely available LLMs should work well (ChatGPT, Gemini, Claude, ...).

Types of tests

With a few minutes of online search, one can identify 19 different types of tests commonly referenced in the industry, and I'm sure there are a few more:

- **Functional testing**: Validates that the API functions as expected by checking its behavior against specified requirements
- **Integration testing**: Ensures the API interacts correctly with other software components, systems, or APIs
- **Unit testing**: Tests individual units or components of the API in isolation
- **Performance testing**: Evaluates the API's responsiveness, scalability, and stability under different conditions
- **Load testing**: Checks how the API handles expected user traffic
- **Stress testing**: Pushes the API beyond its limits to find breaking points

- **Spike testing**: Assesses the API's reaction to sudden traffic spikes
- **Soak testing**: Monitors the API's performance over an extended period under sustained load
- **Security testing**: Ensures the API is secure from vulnerabilities and unauthorized access
- **Validation testing**: Verifies the API meets the expected system and business requirements as a whole
- **Usability testing**: Ensures the API is easy to understand and use by developers
- **Regression testing**: Ensures new changes or updates to the API do not break existing functionality
- **Compliance testing**: Verifies the API meets specific legal, regulatory, or industry standards
- **Exploratory testing**: Performs unscripted interactions with the API to uncover unexpected issues
- **Interoperability testing**: Verifies the API works correctly with different platforms, languages, and environments
- **Fuzz testing**: Identifies vulnerabilities by sending random, malformed, or unexpected data to the API
- **End-to-end testing**: Validates entire workflows or user scenarios involving the API
- **Data testing**: Ensures the integrity and correctness of the data processed or stored by the API
- **Mock and sandbox testing**: Tests API functionality in isolation or simulates environments without impacting live systems

Clearly, a comprehensive discussion on all these types of testing would require an entire book or more. Therefore, in this chapter, we will focus on a few specific types of testing: functional, regression, and validation tests.

Why these specific types of tests? Well, as stated earlier, the main goal is to help software engineers create better APIs. Among the various types of tests, functional, regression, and validation provide the most value in achieving this goal. Their primary benefit is the confidence they instill when making changes. One of the biggest challenges software engineers face is the unpredictable consequences of modifying code. Changes needed to address one problem might inadvertently break something else. This problem is both common and paralyzing. How can we ensure that our changes won't disrupt existing functionality? By prioritizing functional and regression testing, we can tackle this concern head-on, ensuring the API continues to meet its requirements while safeguarding against unintended consequences introduced by new changes. This is the essence of the **test-driven development (TDD)** methodology that started in the early 2000s.

However, creating and maintaining comprehensive test sets is a resource-intensive process, both in terms of time and cost. Writing effective tests requires a deep understanding of the system, careful planning, and meticulous implementation. Once written, these tests must be regularly updated and adapted to reflect changes in the system, which adds to the ongoing maintenance effort. This level of investment can be difficult for organizations to justify, especially in fast-paced environments where tight deadlines and budget constraints are common. As a result, many projects have historically deprioritized testing, often preventing development teams from dedicating the time and resources needed to build robust test suites. This short-sighted approach may save effort in the short term but often leads to higher costs later when defects surface, affecting system reliability and customer satisfaction. Also, let's be honest—writing tests can be boring, and it's a sentiment shared by many in the industry.

Then, in November of 2022, the introduction of generative AI completely transformed the creation of software tests. Generative AI has become an incredible asset in writing tests. While there's an ongoing debate about its impact on developing actual business logic—a discussion for another time—for testing purposes, it's a phenomenal tool. We will leverage generative AI extensively to write our tests and assist in their development. This innovation not only enhances the value of functional and regression testing for software engineers but also makes these tests much easier and cheaper to write.

Test format and tooling

When it comes to testing APIs, we have access to a variety of tools and formats, each serving different needs and levels of expertise. Tools such as Postman and Bruno provide user-friendly interfaces for building, managing, and running API tests. These tools are especially useful for quick exploratory testing or for creating manual test workflows. They excel in environments where visual representation and non-technical collaboration are essential. However, when we shift our focus to automated, scalable, and pipeline-ready testing, tools such as JUnit, which are integrated directly into the development ecosystem, offer significant advantages.

Tools such as Postman allow users to design API tests with minimal coding, making them accessible to a broader audience. These tools often include features such as sharing test collections, generating test documentation, and providing real-time visual results. Despite these strengths, they fall short when integrated into sophisticated CI/CD pipelines. Exporting and running tests from Postman or Bruno often require additional tooling or manual steps, and they lack the seamless compatibility with source control systems such as Git that code-based testing frameworks offer.

In contrast, JUnit, a widely used testing framework for Java, provides a code-first approach to testing APIs. By writing API tests directly in Java using JUnit, developers can ensure their tests are treated as part of the source code. This approach brings several advantages:

- **Integration into deployment pipelines**: JUnit tests are inherently compatible with CI/CD tools such as Jenkins, GitHub Actions, and GitLab CI. This allows API tests to run automatically during builds or deployments, ensuring that no changes go live without thorough testing.
- **Leverage of source code infrastructure**: Because JUnit tests are written in pure code, they benefit from the same tools and workflows used for the application code itself—version control, peer reviews, static analysis, and more. This consistency reduces complexity and ensures tests are always in sync with the application.
- **Scalability and maintainability**: Code-based tests are easier to maintain as they evolve alongside the application. Refactoring tools, IDE features, and linters help developers quickly adapt tests when APIs change, minimizing the risk of outdated or broken tests.
- **Reusability and customization**: JUnit enables developers to write reusable test helpers and utilities, providing greater flexibility for complex testing scenarios. It also supports advanced features such as parameterized tests, mock environments, and dependency injections.

Additionally, JUnit's ability to directly call real APIs—whether deployed locally or in a test environment—ensures that tests mimic real-world usage. This helps catch issues such as incorrect request formatting, unexpected responses, or performance bottlenecks, which can be missed in abstracted environments such as Postman or Bruno.

While Postman and Bruno are excellent for quick, one-off testing or for collaborating with non-developers, JUnit (and similar code-first tools) is the superior option for teams aiming to achieve robust, automated, and pipeline-integrated testing. By treating API tests as first-class citizens within the code base, developers can maximize efficiency, scalability, and confidence in their APIs.

Leveraging the strengths of automated testing is essential, yet AI is beginning to redefine testing entirely and a new approach to testing has appeared.

The evolution of software development practices has always revolved around improving efficiency, quality, and collaboration. One such recent advancement is the integration of generative AI into Agile software development, particularly within the SCRUM framework (see `https://www.scrum.org/resources/what-scrum-module`). Some teams are already experimenting with a new practice where prompts for AI-driven testing become an integral part of the development process, transforming how teams define, implement, and validate new features.

In this updated workflow, the Technical Lead, in collaboration with the Product Owner, plays a pivotal role in shaping the development and testing lifecycle. When a new story is defined, a feature or enhancement to an API is introduced, and the Technical Lead crafts detailed prompts designed for generative AI tools. These prompts serve as the blueprint for testing the story's acceptance criteria. By leveraging the domain expertise of both the Technical Lead and the Product Owner, the prompts encapsulate not only the technical requirements but also the business context and expected outcomes of the feature.

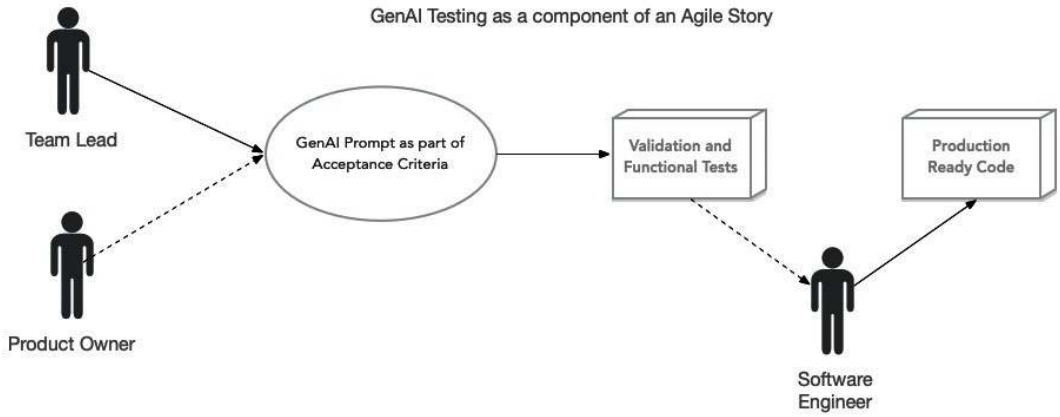


Figure 8.1 – Incorporating AI-driven prompts into Agile SCRUM workflows

For example, if a new API endpoint is being added to return a customer's transaction history, the prompt might specify the exact input parameters, the expected structure of the JSON response, and edge cases to test, such as handling invalid input or empty datasets. These prompts are more than just instructions—they're a shared artifact that bridges technical and business perspectives, ensuring alignment and clarity.

This integration of generative AI into Agile workflows offers several significant advantages:

- **Consistency**: Standardized prompts ensure a consistent approach to testing across stories, reducing variability and improving reliability
- **Efficiency**: By automating the generation of test cases, developers can focus on coding the feature itself, reducing repetitive tasks and freeing up time for innovation
- **Collaboration**: The joint creation of prompts by Technical Leads and Product Owners fosters better communication and alignment between technical and business teams
- **Quality assurance**: Prompts encourage the team to think critically about edge cases, error handling, and expected outcomes from the start, reducing bugs and rework down the line

Storing prompts in the code base

To solidify the importance of these prompts, they are stored in the project's Git repository alongside the source code. This integration elevates prompts to a first-class citizen within the development lifecycle, treating them with the same rigor as the code itself. Storing prompts in Git provides several key benefits:

- **Version control**: Prompts evolve as features mature or requirements change. Storing them in Git ensures a complete history of modifications, enabling teams to trace changes over time.
- **Collaboration**: Developers can review and refine prompts during code reviews, ensuring their accuracy and relevance before implementation begins.
- **Traceability**: Linking prompts to specific stories, commits, and branches creates a transparent and auditable process that ties testing directly to development efforts.
- **Reusability**: Over time, prompts for similar features or scenarios can be repurposed, reducing redundancy and accelerating future development cycles.

AI-generated code as acceptance criteria

Once the prompts are finalized and stored, they become an integral part of the story's acceptance criteria. The assigned software engineer uses the prompt to generate test cases or other relevant code through generative AI tools. This generated code acts as a baseline for validating the new feature. The key requirement is that the generated code must pass successfully—be *green*—before the story is marked as complete.

This approach introduces a new layer of rigor to the definition of *done* in SCRUM. It ensures that testing is not an afterthought but a parallel process that evolves alongside the feature's development. Additionally, by automating large portions of the testing process through AI, teams can deliver high-quality software faster, with fewer manual testing bottlenecks.

Integrating AI-driven testing into Agile practices demands more than tools—it requires a cultural shift.

Adopting a new Agile mindset

This practice represents a shift in mindset for Agile teams. The act of defining prompts moves testing considerations to earlier in the development process, aligning with the principles of TDD. It emphasizes the importance of testing not just as a verification step but as a collaborative, iterative process that enhances the overall quality of the software.

By embedding AI-driven testing prompts into the Agile workflow, teams not only embrace the power of generative AI but also elevate their development practices to a new level of precision and efficiency. This approach aligns perfectly with the goals of Agile: to deliver value faster, adapt to change effectively, and maintain a high standard of quality throughout the development lifecycle.

The structure of an API test environment

The goal of an API testing environment is to replicate all critical components of your application locally, facilitating comprehensive and realistic tests of your code. The following figure illustrates a typical setup for such an environment, emphasizing a self-contained system suitable for development and testing purposes.

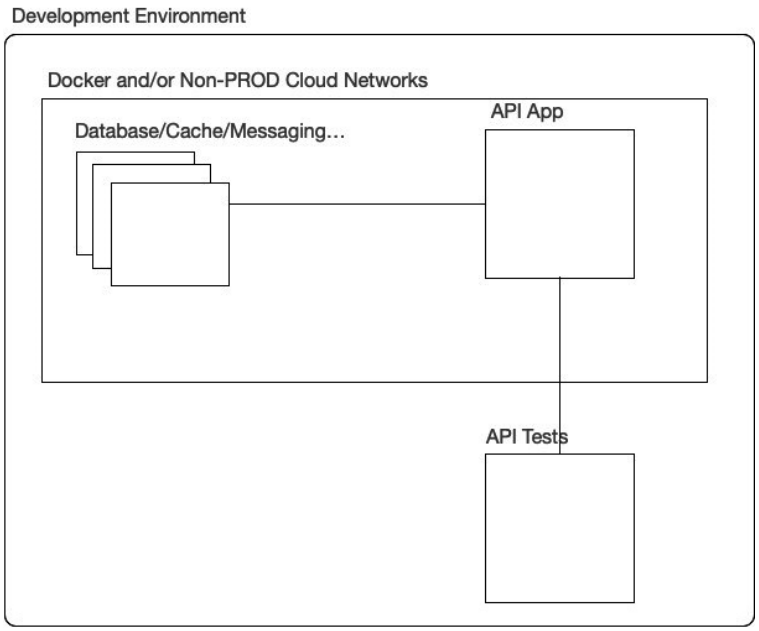


Figure 8.2 – Typical structure of a local API testing environment using Docker and cloud networks

In this diagram, **API App** represents the primary application whose APIs you are testing. This app connects directly to essential backend components such as databases, caching services, or messaging queues, all deployed within a containerized (Docker) or cloud-based, non-production network environment. The **API Tests** component directly interacts with **API App**, simulating real-world scenarios and ensuring thorough coverage and validation of API functionality.

The term "local" here is flexible: it can refer to an individual software engineer's machine, typically using Docker containers, or any non-production cloud-based environment provisioned for development purposes. Ensuring every team member has access to such standardized, isolated environments is crucial, allowing consistent and repeatable tests across different workstations and reducing variability in test outcomes.

There is an upfront effort involved in creating this local or development environment. However, making this investment early in the development lifecycle significantly enhances efficiency, reducing the risk of integration issues and accelerating overall development.

"Prompting" the LLMs

Prompt engineering is the practice of designing and refining prompts given to generative AI models to produce precise, accurate, and relevant outputs. It involves careful phrasing and iterative tuning to achieve desired responses efficiently. And now let's look at some mechanisms needed to create effective prompts for test generation.

As was said before, the use of generative AI for software testing is a perfect application of this technology. Specifically, tests are generally less sensitive in terms of intellectual property and ownership concerns. Tests are primarily tools for validation, ensuring that your main code behaves as expected. Since the focus of tests is on functionality, edge cases, and coverage, generative AI can quickly generate a wide variety of test cases, saving developers time and effort. Moreover, tests are often repetitive and detail-oriented—tasks that AI excels at. The use of generative AI here can allow developers to focus on more creative or complex aspects of their work, while the AI takes care of the grunt work. This means better productivity, faster iteration, and higher test coverage with minimal manual effort.

In contrast, when it comes to using generative AI to write the main production code, the situation becomes a little more complex. A major concern is ownership of the generated code. Depending on the terms of the AI service, the code you use might still be partially owned or influenced by the AI provider. This can create legal or compliance issues, especially in industries where intellectual property is a critical asset. Additionally, there's the concern that your code—or patterns from it—could be used to train future AI models. This could inadvertently expose proprietary algorithms, business logic, or sensitive methodologies to external entities. While this may not be a major issue for tests (which are often more generic and less confidential), it can be a significant risk for production code, where the intellectual property and security of the organization are at stake. At the time of writing, there are several legal decisions establishing that AI-generated content cannot be copyrighted *[3][4]*. Now, we will create the actual prompts for our test.

Testing the Products API

The Products API that we have been using in this book is a very good use case for this new testing strategy. Let's look at a test-generating prompt for that API.

Let's start by creating a test to validate the creation of new products with the `PUT` method:

```
Assuming the role of a Senior Software Engineer in charge of implementing
a set of APIs to support the Products resource
Generate a JUnit 5 test case for the "Create or Update Product" API
endpoint in a Spring Boot application.
The endpoint uses the HTTP PUT method at the path /api/products/
{productId}, where {productId} is a path variable validated with the
format AA99999.
The request body is a ProductInput object, and the response is a
ProductOutput object.

The test should cover two scenarios:
1 - Creating a New Product: When a new productId is provided, the API
should create a new product and return HTTP status 201 Created. Verify
that the response body contains the expected ProductOutput.
2 - Updating an Existing Product: When an existing productId is provided,
the API should update the product and return HTTP status 200 OK. Verify
that the response body contains the updated ProductOutput.

Assume a running instance of this API running on http:8080
Include at least assertions for:
- HTTP status codes (201 Created and 200 OK)
- Response headers (if applicable)
- Response body content
- Validation that the productId follows the format AA99999

Structure the test methods clearly, using descriptive names for each
scenario.

Product is defined by the following classes - assume the expected Get and
Set methods are present.
public class Product {
  private String name;
  private String sku;
```

```
  private String description;
  private BigDecimal price;
  public Product() {
    super();
  }
  public Product(String name, String sku, String description,
                 BigDecimal price) {
    this.name = name;
    this.sku = sku;
    this.description = description;
    this.price = price;
  }
}

public record ProductDescriptionInput(@NotBlank
                                      @Size(min = 10, max = 255)
                                      String description) {
}

public record ProductInput(
        @NotBlank
        @Size(min = 3, max = 255)
        @JsonProperty("name")
        String name,
        @NotBlank
        @Size(min = 10, max = 255)
        @JsonProperty("description")
        String description,
        @NotNull
        @Positive
        @JsonProperty("price")
        BigDecimal price) {

    public Product toProduct(String productId) {
        return new Product(name(),
                productId,
                description(),
```

```
                price());
    }
}
public record ProductOutput(String name, String sku, String description,
                            BigDecimal price) {

}
```

Let's look at each part of this prompt in more detail:

First, we define the LLM perspective. This is relevant to establish the role the LLM should play, and it does influence the answers. Sometimes it is clear in the resulting code when you tell the LLM to act as a senior or junior professional:

```
"Assuming the role of a Senior Software Engineer in charge of implementing
a set of APIs to support the Products resource"
```

Then, give a clear request:

```
"Generate a JUnit 5 test case for the "Create or Update Product" API
endpoint in a Spring Boot application."
"The test should cover two scenarios:… "
```

The LLM's output is heavily influenced by what you want as a result.

Next, explain the API behavior as clearly as possible. This is important because if the underlying behavior is not clear then the LLM will try to fill in the blanks, sometimes with invalid assumptions. The more details you add, the better the test cases created:

```
"The endpoint uses the HTTP PUT method at the path /api/products/
{productId}, …."
```

Part of these details is the code that defines the data being tested. Either present the data structures used or the expected HTTP responses, but some form of unambiguous definition must be present.

Finally, add any specific assertions we want the test to validate:

```
"Include at least assertions for:
- HTTP status codes (201 Created and 200 OK)
- Response headers (if applicable) …"
```

The result of this prompt is the following code and `pom.xml` dependency elements:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>io.projectreactor</groupId>
    <artifactId>reactor-test</artifactId>
    <scope>test</scope>
</dependency>

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.http.HttpStatus;
import org.springframework.test.web.reactive.server.WebTestClient;
import java.math.BigDecimal;

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_
PORT)
public class ProductsApiControllerTest {
    @Autowired
    private WebTestClient webTestClient;

    @Test
    void shouldCreateNewProduct_whenValidInputProvided() {
        // Arrange
        String productId = "AA12345";
        ProductInput productInput = new ProductInput(
                "Test Product",
                "A detailed description for the product",
                BigDecimal.valueOf(99.99)
        );

        // Act & Assert
        webTestClient.put()
```

```
                .uri("/api/products/{productId}", productId)
                .bodyValue(productInput)
                .exchange()
                .expectStatus().isCreated()
                .expectHeader().exists("Location") // Optional: Check if
Location header exists
                .expectBody(ProductOutput.class)
                .value(response -> {
                    // Assert response body content
                    assert response.name().equals(productInput.name());
                    assert response.sku().equals(productId);
                    assert response.description().equals(
                      productInput.description());
                    assert response.price().compareTo(
                      productInput.price()) == 0;
                });
    }

    @Test
    void shouldUpdateExistingProduct_whenValidInputProvided() {
        // Arrange
        String productId = "AA67890"; // Assume this product already
exists in the system
        ProductInput productInput = new ProductInput(
                "Updated Product",
                "An updated description for the product",
                BigDecimal.valueOf(149.99)
        );

        // Act & Assert
        webTestClient.put()
                .uri("/api/products/{productId}", productId)
                .bodyValue(productInput)
                .exchange()
                .expectStatus().isOk()
                .expectBody(ProductOutput.class)
                .value(response -> {
                    // Assert response body content
```

```
                    assert response.name().equals(productInput.name());
                    assert response.sku().equals(productId);
                    assert response.description().equals(
                      productInput.description());
                    assert response.price().compareTo(
                      productInput.price()) == 0;
                });
    }

    @Test
    void shouldRejectInvalidProductIdFormat() {
        // Arrange
        String invalidProductId = "INVALID123";
        ProductInput productInput = new ProductInput(
                "Invalid Product",
                "An invalid description",
                BigDecimal.valueOf(49.99)
        );

        // Act & Assert
        webTestClient.put()
                .uri("/api/products/{productId}", invalidProductId)
                .bodyValue(productInput)
                .exchange()
                .expectStatus().isBadRequest(); // Assuming the API
enforces validation on productId
    }
}
```

This code is not in itself complex or hard to write, but it is time-consuming, and most software engineers don't particularly enjoy writing it. Also, using an LLM to create the tests adds a lot of flexibility to the process. For example, if you want this test to be created in a different language—for example, in Go—we only need to change the expected results to have a new version.

```
"Generate a Go test case for the "Create or Update Product" API endpoint
in a Spring Boot application."
```

Go ahead and play around with the code generation options.

In some projects, there is even the requirement that automated tests be written in a different language than the original code. There may not be a real technical reason for such a requirement, as an HTTP client is exactly like another, but it offers some extra psychological comfort to non-technical people in the project to have this extra layer of separation.

A more complex API to test

In the previous exercise, we looked at testing an API in which we had complete access to the source code, and we were developing that API from scratch. Also, the example API was intentionally simple. But, there is another common situation that many software engineers face: An API that is already in production, with not many automated tests available, that is more complex and returns sophisticated responses. Let's look at one such scenario.

For this section, we are going to use the HAPI FHIR APIs and write some tests for them.

HAPI FHIR is a complete Java implementation of the HL7 FHIR standard for healthcare interoperability. We chose this example because the API is complex, there is a public test implementation available, and it is used extensively in the healthcare market. This implementation is used by many organizations worldwide, including large hospital complexes, healthcare organizations, and governmental health agencies. But what is interesting to us from a testing perspective is how complex these APIs are. It is not uncommon to receive a few thousand lines of JSON in the response to a search, which allows us to create more nuanced prompts and corresponding tests.

To know more about HAPI FHIR and HL7 FHIR, please refer to the following links:

`https://hapifhir.io/`

`https://hl7.org/fhir/`

Because healthcare requirements are usually unique to each organization and dependent on many local regulations, there is a high level of customization required for any HAPI FHIR implementation. It is not uncommon to find several thousand lines of custom code in a specific implementation. A strong set of tests is crucial when customizing such a complex API. You don't want to break any of the HAPI FHIR contracts.

Also, one great advantage of HAPI FHIR as a test example for us is the availability of a public test server: `https://hapi.fhir.org/baseR4`

Having prior knowledge of the HAPI FHIR APIs is not necessary to follow this section. We are interested only in looking at the API's behavior and preparing tests to validate that such behavior does not change. Here are a couple of HAPI FHIR concepts we will use:

- **PATIENT**: HAPI FHIR is a healthcare system; therefore, PATIENT is a first-class element in it. We will write some tests to validate that searching for a patient returns the elements we want.
- **RESOURCE TYPE – BUNDLE**: In HAPI FHIR, all results of a search are bundles. This is just the name used to define a set of data, but we will validate this behavior as part of our tests.

With that in mind, let's assume you are a software engineer in the HAPI FHIR support team in a large hospital, and you just received the task of customizing the HAPI FHIR API to add a new element to a `Patient` record. This new element is used only in this hospital's system. Assume that there are no test cases for the Patient API. What can you do?

Well, you could start by looking at the source code and try to find all the relevant classes and all, but this is a time-consuming task and will delay the generation of the tests. Maybe a more pragmatic approach is needed.

So, start by looking at the API responses and prepare tests so your changes will not break the current contract.

Here is a `GET` request that returns all the patients in the system, one at a time. All the data is synthetic and not related to any real individual:

```
https://hapi.fhir.org/baseR4/Patient?_format=json&_count=1
```

Let's examine in detail the response we received (you will not get the same data when running this example, but the format and structure will be the same):

```
{  "resourceType": "Bundle",
  "id": "241fc9a1-3d08-4c1c-95a9-6ac315d178a8",
  "meta": {
    "lastUpdated": "2024-11-18T13:05:22.536+00:00"
  },
  "type": "searchset",
  "link": [
    {
      "relation": "self",
      "url": "https://hapi.fhir.org/baseR4/Patient?_count=1&_format=json"
    },
```

```
    {
      "relation": "next",
      "url": "https://hapi.fhir.org/baseR4?_getpages=241fc9a1-3d08-4c1c-
95a9-6ac315d178a8&_getpagesoffset=1&_count=1&_format=json&_pretty=true&_
bundletype=searchset"
    }
  ],
  "entry": [
    {
      "fullUrl": "https://hapi.fhir.org/baseR4/Patient/596484",
      "resource": {
        "resourceType": "Patient",
        "id": "596484",
        "meta": {
          "versionId": "1",
          "lastUpdated": "2020-02-01T17:42:02.000+00:00",
          "source": "#McyYCIRVGrcmfKSf"
        },
        "text": {
          "status": "generated",
          "div": "<div xmlns…div>"
        },
        "name": [
          {
            "family": "DelGrosso",
            "given": [
              "Joe"
            ]
          }
        ],
        "telecom": [
          {
            "system": "phone",
            "value": "1(555)123-4567",
            "use": "home"
          }
        ],
        "gender": "male",
```

```
        "birthDate": "1985-09-11",
        "address": [
          {
            "use": "home",
            "line": [
              "155 Quick Rd"
            ],
            "city": "Fun City",
            "state": "NY",
            "postalCode": "10101",
            "country": "US"
          }
        ]
      },
      "search": {
        "mode": "match"
      }
    }
  ]
}
```

This result is already 70+ lines long and is a good start for our API analysis and to design our tests.

Let's look at the results. We can see that the result of a `GET` function has a `resourceType`, which in this case is `Bundle`. `Type` is a `searchset`.

There is an array of links ("link") and an array of entries ("entry") with the actual patient's data.

Good, this is a start.

Preparing your test prompt

Remember that your goal is to make changes to the code, without breaking the current state, so start by defining clearly what you want to validate with your test—for example, "Test that all `GET` searches will return a valid pagination JSON." Now, let's write a prompt to generate our first test. The results are from ChatGPT 4:

```
Assuming the role of a Senior Software Engineer working on a HAPI FHIR
customization project.
Prepare a test case that will make a call to the following URL: https://
hapi.fhir.org/baseR4/Patient?_format=json&_count=1
```

```
Make sure to validate that the response is in a JSON format and contains:
- a single 'resourceType' of 'Bundle'
- a single 'type' of 'searchset'
- a single 'id' in the format of an UUID

Use JAVA Junit 5

Here is a sample result set. The actual values of each element will vary
from each call.

{
  "resourceType": "Bundle",
  "id": "241fc9a1-3d08-4c1c-95a9-6ac315d178a8",
  "meta": {
    "lastUpdated": "2024-11-18T13:05:22.536+00:00"
….
```

Paste the whole JSON sample with the prompt.

The results are the following tests and `pom.xml` changes:

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;
import org.springframework.http.ResponseEntity;
import org.springframework.web.client.RestTemplate;
import com.fasterxml.jackson.databind.JsonNode;
import com.fasterxml.jackson.databind.ObjectMapper;
import java.util.UUID;

class HapiFhirTest {

    private static final String HAPI_FHIR_URL =
      "https://hapi.fhir.org/baseR4/Patient?_format=json&_count=1";
    @Test
    void testHapiFhirResponse() {
        // Set up RestTemplate for HTTP call
        RestTemplate restTemplate = new RestTemplate();
```

```
        // Make the HTTP GET call
        ResponseEntity<String> response = restTemplate.getForEntity(
          HAPI_FHIR_URL, String.class);

        // Assert the response is not null and has a 200 status code
        assertNotNull(response);
        assertEquals(200, response.getStatusCodeValue());

        // Assert the response body is not null
        String responseBody = response.getBody();
        assertNotNull(responseBody);

        try {
            // Parse the response body as JSON
            ObjectMapper objectMapper = new ObjectMapper();
            JsonNode jsonNode = objectMapper.readTree(responseBody);

            // Assert 'resourceType' is present and equals 'Bundle'
            assertTrue(jsonNode.has("resourceType"));
            assertEquals("Bundle", jsonNode.get("resourceType").asText());

            // Assert 'type' is present and equals 'searchset'
            assertTrue(jsonNode.has("type"));
            assertEquals("searchset", jsonNode.get("type").asText());

            // Assert 'id' is present and is a valid UUID
            assertTrue(jsonNode.has("id"));
            String id = jsonNode.get("id").asText();
            assertTrue(isValidUUID(id));

        } catch (Exception e) {
            fail("Exception occurred while processing the response: " +
                  e.getMessage());
        }
    }

    /**
```

```
     * Utility method to validate if a string is a valid UUID.
     *
     * @param uuid the string to validate
     * @return true if valid UUID, false otherwise
     */
    private boolean isValidUUID(String uuid) {
        try {
            UUID.fromString(uuid);
            return true;
        } catch (IllegalArgumentException e) {
            return false;
        }
    }
}
With the accompanying dependencies in pom.xml
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-web</artifactId>
    <version>5.3.30</version>
</dependency>
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.15.2</version>
</dependency>
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter</artifactId>
    <version>5.10.0</version>
</dependency>
```

This code runs immediately without any modifications. This was not the case with the previous generation of LLMs. If you are using an older model, such as ChatGPT 3.5 for example, you may have to manually adjust the code.

This whole test was completed in less than a minute by the LLM and it is actually fun to write and see the results so quickly—a very different experience from having to manually write tests by hand. The test can be improved, of course. For example, it could be changed to get the URL to be tested from a list or even a property file for added flexibility, but the core of the code is there and ready to be used.

Now, let's move on to a more complex type of test and see how far we can go with our validations.

Validating more complex behavior

Looking at the API's response, we can see that the "link" array has two entries on this first page: "`self`" and "`next`".

```
  "link": [
    {
      "relation": "self",
      "url": "https://hapi.fhir.org/baseR4/Patient?_count=1&_format=json"
    },
    {
      "relation": "next",
      "url": "https://hapi.fhir.org/..."
    }
  ],
```

A good test would be to validate that a response always has the `self` link. Let's expand the prompt to include this.

Here's the new prompt:

```
Assuming the role of a Senior Software Engineer working on a HAPI FHIR
customization project.
Prepare a test case that will make a call to the following URL: https://
hapi.fhir.org/baseR4/Patient?_format=json&_count=1

Make sure to validate that the response is in a JSON format and contains:
- a single 'resourceType' of 'Bundle'
- a single 'type' of 'searchset'
- a single 'id' in the format of an UUID

Validate that array "link" is present and that there is one element with
"relation = self"
```

```
Use JAVA Junit 5
…
```

This will add the following code to our test:

```
// Validate link array
assertTrue(jsonNode.has("link"), "Response does not contain 'link'");
JsonNode links = jsonNode.get("link");
assertTrue(links.isArray(), "link is not an array");

boolean hasSelfLink = false;
for (JsonNode link : links) {
    if (link.has("relation") && "self".equals(
        link.get("relation").asText())) {
        hasSelfLink = true;
        break;
    }
}
assertTrue(hasSelfLink, "No link with 'relation' = 'self'");
```

Again, the code is not very complex, but it is already showing signs of increasing difficulty. Writing this by hand would not be hard, but would take some time and attention. LLMs can create it in 30 seconds or less.

Now, let's add even more complexity. This is a pagination exercise, so let's ask the code to follow the `next` link for a few pages and validate that page. Then, validate if every entry in the result is unique, using `fullUrl` inside the `entry` array as a candidate key.

```
Assuming the role of a Senior Software Engineer working on a HAPI FHIR
customization project.
Prepare a test case that will make a call to the following URL: https://
hapi.fhir.org/baseR4/Patient?_format=json&_count=1

Make sure to validate that the response is in a JSON format and contains:
- a single 'resourceType' of 'Bundle'
- a single 'type' of 'searchset'
- an single 'id' in the format of an UUID

Validate that array "link" is present and that there is one element with
```

```
"relation = self"

Now, follow the "next" link for at most 5 pages or until the last page.
Validate each page.

In each "entry" there is a "fullUrl" attribute. Threat it as a key and
validate that it is present in every "entry".
Also, make sure that the entries are unique and there is no repetition.

Use JAVA Junit 5
```

The result is starting to look more sophisticated and interesting.

```
public class HapiFhirPaginationTest {
    @Test
    void testHapiFhirPatientPagination() throws Exception {
        String baseUrl =
          "https://hapi.fhir.org/baseR4/Patient?_format=json&_count=1";
        ObjectMapper mapper = new ObjectMapper();
        String nextUrl = baseUrl;
        int pageCount = 0;
        Set<String> uniqueFullUrls = new HashSet<>();

        while (nextUrl != null && pageCount < 5) {
            HttpURLConnection connection = createConnection(nextUrl);
            int responseCode = connection.getResponseCode();
            assertEquals(200, responseCode,
                         "Expected HTTP status code 200");

            InputStream responseStream = connection.getInputStream();
            JsonNode response = mapper.readTree(responseStream);

            validatePage(response, uniqueFullUrls);
            nextUrl = getNextLink(response);
            pageCount++;

            connection.disconnect();
        }
    }
```

```
    private HttpURLConnection createConnection(String urlString) throws
Exception {
        URL url = new URL(urlString);
        HttpURLConnection connection = (HttpURLConnection)
                                        url.openConnection();
        connection.setRequestMethod("GET");
        connection.setRequestProperty("Accept", "application/json");
        return connection;
    }

    private void validatePage(JsonNode response,
                              Set<String> uniqueFullUrls) {
        // Validate resourceType
        assertTrue(response.has("resourceType"),
                   "Response does not contain 'resourceType'");
        assertEquals("Bundle", response.get("resourceType").asText(),
                     "resourceType is not 'Bundle'");

        // Validate id is in UUID format
        assertTrue(response.has("id"), "Response does not contain 'id'");
        String id = response.get("id").asText();
        assertTrue(isValidUUID(id), "id is not a valid UUID");

        // Validate type
        assertTrue(response.has("type"),
                   "Response does not contain 'type'");
        assertEquals("searchset", response.get("type").asText(),
                     "type is not 'searchset'");

        // Validate link array
        assertTrue(response.has("link"),
                   "Response does not contain 'link'");
        JsonNode links = response.get("link");
        assertTrue(links.isArray(), "link is not an array");

        boolean hasSelfLink = false;
        for (JsonNode link : links) {
```

```
            if (link.has("relation") && "self".equals(link.get(
                                        "relation").asText())) {
                hasSelfLink = true;
                break;
            }
        }
        assertTrue(hasSelfLink, "No link with 'relation' = 'self'");

        // Validate entries
        if (response.has("entry")) {
            JsonNode entries = response.get("entry");
            for (JsonNode entry : entries) {
                assertTrue(entry.has("fullUrl"),
                           "Entry does not contain 'fullUrl'");
                String fullUrl = entry.get("fullUrl").asText();

                // Ensure fullUrl is unique
                assertTrue(uniqueFullUrls.add(fullUrl),
                           "Duplicate fullUrl detected: " + fullUrl);
            }
        }
    }

    private String getNextLink(JsonNode response) {
        JsonNode links = response.get("link");
        for (JsonNode link : links) {
            if (link.has("relation") && "next".equals(link.get(
                                        "relation").asText())) {
                return link.get("url").asText();
            }
        }
        return null;
    }

    private boolean isValidUUID(String id) {
        try {
            UUID.fromString(id);
```

```
            return true;
        } catch (IllegalArgumentException e) {
            return false;
        }
    }
}
```

The test is now a lot more complex. Look at how some helping functions are created (`isValidUUID()`, `createConnection()`). This test would take longer to write and be costlier to maintain. But with the use of LLMs, the whole test is ready to run in just a few minutes.

I hope you are seeing the pattern at this point. As the API gets more complex, the tests also evolve. The LLM prompt itself is now stored in the Git repository and is an integral part of the code base, to the point that it can be seen as part of the software documentation.

Summary

The introduction of generative AI holds immense potential to reshape the landscape of software development, especially in the realm of testing. By automating test creation, streamlining complex processes, and enhancing adaptability, generative AI offers the promise of transforming testing into a more efficient, accessible, and cost-effective part of the development lifecycle.

Through the strategies and examples discussed in this chapter, we hope you have acquired the skills and inspiration to explore and extend the testing of your APIs.

Testing is a tool for validating behavior during the development and maintenance of our APIs, but for a truly robust service, we need to monitor these behaviors in production. That is what we will be looking at in the next chapter.

Further reading

- *Test driven development*: `https://en.wikipedia.org/wiki/Test-driven_development`
- *Prompt Engineering*: `https://en.wikipedia.org/wiki/Prompt_engineering`
- *Legal Ruling: AI-Generated Art and Copyright*: `https://www.reuters.com/legal/ai-generated-art-cannot-receive-copyrights-us-court-says-2023-08-21/`
- *U.S. Copyright Office's View on AI Prompts*: `https://www.theverge.com/news/602096/copyright-office-says-ai-prompting-doesnt-deserve-copyright-protection`
- *HAPI FHIR: Open-Source Implementation of HL7 FHIR*: `https://hapifhir.io/`
- *HL7 FHIR Official Specification* `https://hl7.org/fhir/`

Part 3

Deployment and Performance

This part guides you through deploying and scaling your RESTful APIs to ensure they perform well in real-world environments. You'll learn how to monitor and trace your APIs for better observability, optimize performance with techniques such as virtual threads and load testing, explore alternative Java frameworks for API development, and implement practical deployment strategies using containers and cloud platforms.

This part will cover the following chapters:

- *Chapter 9, Monitoring and Observability*
- *Chapter 10, Scaling and Performance Optimization Techniques*
- *Chapter 11, Alternative Java Frameworks to Build RESTful APIs*
- *Chapter 12, Deploying APIs*

9

Monitoring and Observability

Monitoring and **observability** are fundamental to ensuring that RESTful services operate reliably, efficiently, and securely. In distributed systems, where requests often pass through multiple services and components, having proper observability practices in place is critical. Without them, diagnosing performance bottlenecks, identifying the root causes of errors, and optimizing service behavior becomes incredibly challenging.

In this chapter, we will guide you through the essential practices and tools needed to achieve effective observability for RESTful services. We will begin by discussing **logging** best practices, explaining how structured logging and correlation IDs can simplify troubleshooting. We will then delve into distributed tracing, demonstrating how trace information such as `traceId`, `spanId`, and `parentSpanId` helps map the lifecycle of a request as it flows across multiple services.

We will introduce **Micrometer Tracing**, a powerful observability framework integrated into Spring Boot 3.x, which automatically instruments applications to capture trace data. Additionally, we will explore OpenTelemetry, a vendor-neutral framework that extends observability by collecting and correlating logs, metrics, and traces for a holistic view of distributed systems. By the end of this chapter, you will understand how to implement logging and tracing effectively, configure Spring Boot applications for observability, and visualize your data in tools like Zipkin and Jaeger.

This chapter provides practical examples, step-by-step guidance, and best practices to ensure you can successfully monitor and debug your RESTful services.

In this chapter, we will be covering the following topics:

- Importance of logging in REST APIs
- Logging best practices for API troubleshooting

- Logging basics with SLF4J
- Implementing a central logging filter
- Implementing service tracing in distributed systems
- Implementing tracing using Micrometer
- Metrics from tracing data
- OpenTelemetry for monitoring and observability
- Best practices for end-to-end observability

Technical requirements

In this chapter, we will enhance our existing APIs—the Product API and the Order Management API—to be able to trace requests between them. To be able to follow along and use the code examples as they are printed in the book, you should have the following:

- Intermediate knowledge of the Java language and platform
- At least a basic knowledge of Spring Boot or a similar framework
- Java 21 and Maven 3.9.0 installed
- Docker 27.3.1 or higher installed

You can access the code for this chapter on GitHub at `https://github.com/PacktPublishing/Mastering-RESTful-Web-Services-with-Java/tree/main/chapter9`.

Importance of logging in REST APIs

Logging is the foundational element of observability in any system. Logs are essentially records of events that happen while your application runs. They are the first source of information when trying to troubleshoot an issue in any environment, especially in production.

Logs act as the system's memory, providing insights into what went wrong or how certain processes behaved. For a REST API, logs can show the path of each request, which is crucial for understanding failures or slow performance.

For instance, suppose your API returns a `500 Internal Server Error` to a user. Without logs, you would have no way of knowing what caused the error. However, with logs, you could see that a database query failed because the server ran out of connections, helping you fix the problem.

In the next sections, we are going to cover an effective logging design that will empower our applications significantly when troubleshooting is needed.

Common logging pitfalls

Despite the importance of logging, many developers struggle with using logs effectively. Common mistakes include:

- **Over-logging**: Logging too much information can make it hard to find key details.
- **Under-logging**: Insufficient logging might leave out key data needed for troubleshooting.
- **Logging sensitive information**: Mistakenly logging things like user passwords or credit card numbers, which should never happen due to security and compliance concerns (e.g., GDPR or PCI DSS).

A balance needs to be struck between logging enough to troubleshoot issues and not overwhelming the system with unnecessary data.

Effective log design

Designing logs properly is crucial for them to be useful. Each log entry should include relevant metadata:

- **Timestamp**: When the log entry was created.
- **Log level**: Severity of the log (`INFO`, `DEBUG`, `ERROR`).
- **Service name**: The name of the service generating the log.
- **Correlation ID**: A unique ID that allows you to track a request through multiple services (more on this below).

For example, a well-structured log entry in JSON might look like this:

```
{
  "timestamp": "2024-10-23T10:20:30Z",
  "level": "ERROR",
  "service": "user-service",
  "correlationId": "abc123",
  "message": "Database connection timed out"
}
```

Correlation IDs

In microservices, a request might pass through multiple services. To track the entire request path, it is essential to attach a correlation ID to each request. The correlation ID is a unique identifier that stays with the request as it moves through the system. This allows you to correlate logs from different services to see how a single request was handled end to end. For example, a user request to retrieve profile information might go through an API gateway, then hit the authentication service, and finally query the user database. With a correlation ID, you can trace each step of the process across all services involved. Spring makes it easy to generate and propagate correlation IDs. You can generate a correlation ID at the start of a request and pass it along with HTTP headers between services. This will help you diagnose where issues occur in a chain of services.

Next, we will learn about the best logging practices, understand the different log levels and their structure, and what should and should not be logged to avoid security issues.

Logging best practices for API troubleshooting

Effective logging is essential for troubleshooting, especially when working with distributed systems or cloud-native architectures. Here are some best practices to ensure your logging strategy is robust.

Choosing the right log level

Logs should be written at the appropriate log level, which indicates the severity of the event:

- `TRACE`: Used for very fine-grained details, primarily for debugging low-level operations like recording the internal state of loops, method entry/exit points, or interactions between components in detail. Should be turned off in production due to the huge amount of logs that are generated at this level.
- `DEBUG`: Used for low-level information that helps in debugging issues, such as details of an HTTP request. Logs provide detailed information that helps during development or debugging, focusing on application-specific logic or operations. Also, this level should be turned off in production due to the huge amount of logs that are generated at this level.
- `INFO`: Used for general application flow information, such as when a service starts up or shuts down. These logs are less verbose than `DEBUG` or `TRACE` and are typically enabled in production.

- `WARN`: Indicates something unusual but not necessarily an error. For example, a service might temporarily run out of resources but recover. These logs are a warning for potential future issues. For example:
- `ERROR`: Used when something has gone wrong, like an exception being thrown or a critical failure in a database connection. These logs often indicate that the system requires attention or intervention.
- `FATAL`: Indicates a critical error that causes the application or service to crash or become unusable. These logs are extremely rare and signify the most severe issues that require immediate attention. Note that this level is not present universally in libraries like SLF4J or Logback and is often represented by the `ERROR` level; however, it is present in Log4J and Log4J2 logging libraries, which will not be covered by this chapter.

In a REST API, a failed user login attempt due to incorrect credentials might be logged at the `WARN` level:

```
{
  "timestamp": "2024-10-23T11:15:30Z",
  "level": "WARN",
  "service": "auth-service",
  "message": "Failed login attempt for user john.doe@example.com",
  "error": "AuthenticationException"
}
```

Meanwhile, a database outage that causes the entire service to fail should be logged at the `ERROR` level or as `FATAL` if available:

```
{
  "timestamp": "2024-10-23T11:20:30Z",
  "level": "ERROR",
  "service": "user-service",
  "message": "Failed to connect to database",
  "error": "TimeoutException" }
```

Choosing the right level ensures that you can quickly filter out non-critical logs when troubleshooting.

Structured logging

Structured logging refers to logging in a consistent, machine-readable format, such as JSON. This allows logs to be easily parsed and queried by logging tools (like ELK Stack or Splunk), making it easier to filter, aggregate, and analyze logs.

Rather than logging a simple message like this:

```
User john.doe@example.com failed to log in
```

You should log the event in a structured format:

```
{
  "timestamp": "2024-10-23T12:00:00Z",
  "level": "WARN",
  "service": "auth-service",
  "user": "john.doe@example.com",
  "event": "login-failure",
  "reason": "incorrect-password"
}
```

Now, you can easily search for all login failures or group them by the `user` field.

Avoiding sensitive data in logs

Sensitive information such as passwords, credit card numbers, or personal identifiers should never be logged. If this data is accidentally exposed in logs, it can lead to serious security breaches.

For example, if a login attempt fails, it is okay to log the username, but never log the password. A log message like this:

```
{
  "timestamp": "2024-10-23T12:05:00Z",
  "level": "ERROR",
  "message": "Login failed for user john.doe@example.com with password
'secretpassword'"
}
```

is dangerous. Instead, log something like this:

```
{
  "timestamp": "2024-10-23T12:05:00Z",
  "level": "ERROR",
  "message": "Login failed for user john.doe@example.com"
}
```

You can use Jackson's `@JsonIgnore` or `@JsonProperty(access = Access.WRITE_ONLY)` annotations to prevent sensitive data from being serialized into logs.

Jackson is a widely used Java library for processing JSON data. It provides powerful capabilities for serializing Java objects into JSON and deserializing JSON into Java objects, making it a crucial tool in RESTful web services where data is often exchanged in JSON format.

In a Spring Boot application, Jackson is the default JSON processor and is commonly used to automatically transform request and response payloads, making API interactions seamless.

It allows developers to customize JSON output using annotations like `@JsonIgnore`, `@JsonProperty`, and `@JsonInclude`, ensuring that only the necessary fields are exposed while sensitive or unnecessary data is excluded.

This is particularly useful when logging requests or response objects, as it ensures that sensitive information (like passwords or credit card details) does not get exposed in log entries.

When to use each annotation?

- `@JsonIgnore`: Use when you want to prevent a field from ever being included in serialized output, such as responses and logs, and deserialized input from requests.
- `@JsonProperty(access = Access.WRITE_ONLY)`: Use when you need to accept the field as input but want to exclude it from all serialized output, making it suitable for fields that should remain private (e.g., passwords) during logging or API responses.

Let us go through an example to demonstrate how to use these annotations in a RESTful service.

Suppose you have a `User` class with fields like `username`, `email`, and `password`. When logging this `User` object, we want to ensure that the password field is not included in the serialized output.

Completely exclude the field from serialization/ deserialization

The `@JsonIgnore` annotation completely omits a field from deserialization and serialization, meaning it will not be included in the input or the output JSON at all.

When you use `@JsonIgnore` on a field, it is completely ignored by Jackson both during serialization (when converting an object to JSON) and deserialization (when converting JSON to an object). This means that if you mark a field with `@JsonIgnore`, Jackson will neither include it in the output JSON nor allow it as an input in the JSON request body.

This is useful when you want to ensure that sensitive information (e.g., passwords or tokens) is never exposed in any serialized output, including logs.

```
import com.fasterxml.jackson.annotation.JsonIgnore;

public class User {
         private String username;
    private String email;
         @JsonIgnore  // This will prevent the password from being
                     //serialized or deserialized into JSON
    private String password;
    // Constructors, getters, setters omitted for brevity
}
```

With this setup, if you log the User object using Jackson for serialization, the password field will be omitted:

```
import com.fasterxml.jackson.databind.ObjectMapper;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class UserService {

    private static final Logger logger = LoggerFactory.getLogger(
        UserService.class);
    private static final ObjectMapper objectMapper = new ObjectMapper();
```

```
    public void logUserDetails(User user) {
        try {
            String userJson = objectMapper.writeValueAsString(user);
            logger.info("User details: {}", userJson);
        } catch (Exception e) {
            logger.error("Error serializing user details", e);
        }
    }
}
```

Output log (password is excluded):

```
{  "username": "john.doe",
  "email": "john.doe@example.com" }
```

In this example, the `password` field is completely omitted from the log output because of the `@JsonIgnore` annotation.

Now that you know how to exclude fields from serialization and deserialization, let us see how to allow a field to be deserialized from JSON input but excluded from serialization, such as in logs or API responses.

Allowing data input deserialization but excluding it from output serialization

The `@JsonProperty(access = Access.WRITE_ONLY)` annotation is useful when you want a field to be deserialized (e.g., when receiving input from a user) but not serialized (e.g., when logging or sending data as a response). This is common for fields like passwords in user registration forms:

```
import com.fasterxml.jackson.annotation.JsonProperty;
import com.fasterxml.jackson.annotation.JsonProperty.Access;

public class User {

    private String username;
    private String email;
         @JsonProperty(access = Access.WRITE_ONLY)  // Allows input but
                                                    //prevents serialization into JSON
```

```
    private String password;
    // Constructors, getters, setters omitted for brevity
}
```

With `@JsonProperty(access = Access.WRITE_ONLY)`, you can still accept the `password` field in incoming JSON requests, but it will be excluded from any serialized output, including logs.

Example usage:

```
import com.fasterxml.jackson.databind.ObjectMapper;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/users")
public class UserController {

    private static final Logger logger = LoggerFactory.getLogger(
        UserController.class);
    private static final ObjectMapper objectMapper = new ObjectMapper();

    @PostMapping("/register")
    public String registerUser(@RequestBody User user) {
        // Log user details without exposing password
        try {
            String userJson = objectMapper.writeValueAsString(user);
            logger.info("Received user registration request: {}",
                        userJson);
        } catch (Exception e) {
            logger.error("Error serializing user details", e);
        }
                // Proceed with user registration logic
        return "User registered successfully!";
    }
}
```

When a new user registration request comes in, the `password` field will be available for processing, but it will not be logged.

Incoming request:

```
{
  "username": "john.doe",
  "email": "john.doe@example.com",
  "password": "mysecretpassword"
}
```

Output log (password is excluded from the log):

```
{
  "username": "john.doe",
  "email": "john.doe@example.com"
}
```

With `@JsonProperty(access = Access.WRITE_ONLY)`, the password is received by the application but is excluded from logs or other serialized JSON output, ensuring that sensitive data is protected.

By using these annotations strategically, you can control sensitive data exposure in logs, which is an essential part of security best practices for RESTful APIs.

Now, let's get back to the last best practice for logging

Capturing contextual information

To make logs more useful, include contextual information about each request, such as:

- HTTP method (`GET`, `POST`, etc.)
- Endpoint (e.g., `/api/v1/users/123`)
- Response status (`200`, `404`, `500`, etc.)
- Request headers and payloads (but be careful to exclude sensitive data)

This information will allow you to better understand what happened during an API request.

For example, a request log might look like this:

```
{   "timestamp": "2024-10-23T12:10:00Z",
  "level": "INFO",   "service": "user-service",
  "method": "GET",
  "endpoint": "/api/v1/users/123",
  "status": 200 }
```

By capturing such details, you can correlate issues with specific requests and quickly pinpoint where things went wrong.

Next, we will dive into one of the most famous logging libraries in the market, SLF4J, and how to create logs from our application.

Logging basics with SLF4J

This section introduces how to add logging points into your own Java code using SLF4J. This is crucial since you need to add loggers throughout your entire application to be able to generate logs from its flow that will help you troubleshoot it when it is deployed, especially in production.

In Spring Boot, SLF4J (Simple Logging Facade for Java) is commonly used as a logging API that can work with different logging frameworks (such as Logback, Log4j2, etc.). Spring Boot uses SLF4J by default and integrates it seamlessly, so all we need to do is inject the logger and start logging messages.

Also, if you are not using Spring Boot, you can just add the SLF4J dependency to your Maven POM dependencies file or Gradle dependencies file to use it if the framework that you are using does not already include it.

Let us start with a simple example of how to log a Spring Boot application using SLF4J. Here is a User Creation service that uses SLF4J to log messages at different log levels (`INFO`, `WARN`, `ERROR`):

```
@Service
public class UserService {
    // Initialize the logger
    private static final Logger logger = LoggerFactory.getLogger(
        UserService.class);

    public void createUser(String username) {
        logger.info("Creating a new user with username: {}", username);

        try {
            // Simulate some business logic
            if (username == null) {
                throw new IllegalArgumentException(
                    "Username cannot be null");
            }
            // Simulate a successful operation
```

```
            logger.info("User {} created successfully", username);
        } catch (Exception e) {
            // Log the exception as an error
            logger.error("Error creating user: {}", e.getMessage(), e);
        }
    }
}
```

In this example:

- We log informational messages with `logger.info()`.
- We log an error with `logger.error()`, including the exception stack trace.
- The `LoggerFactory.getLogger(UserService.class)` creates a logger specifically for the `UserService` class.

For successful user creation, we would have a log output like this:

```
2024-10-23T12:00:00.123 INFO  [UserService] Creating a new user with
username: john_doe
2024-10-23T12:00:00.456 INFO  [UserService] User john_doe created
successfully
```

Or, for a failed user creation, for example, trying to create a null user, the output would be:

```
2024-10-23T12:01:00.123 INFO  [UserService] Creating a new user with
username: null
2024-10-23T12:01:00.125 ERROR [UserService] Error creating user: Username
cannot be null
java.lang.IllegalArgumentException: Username cannot be null
    at com.example.service.UserService.createUser(UserService.java:12)...
```

Using SLF4J with different log levels helps organize and filter log messages, making it easier to troubleshoot and debug.

In the next section, let us look at how we can automate some of the logging in our application to reduce the burden of having tons of logging everywhere in your code, by implementing a central logging filter.

Implementing a central logging filter

To improve observability in a RESTful application, we can implement a central logging component that logs all incoming HTTP requests and responses. A filter is an effective choice for this as it allows you to intercept requests before they reach the controller layer, enabling you to log key requests and response details in one place.

In this example, we will implement a `RequestLoggingFilter` that logs incoming requests, following best practices such as structured logging, adding correlation IDs, and avoiding sensitive information. This filter will log essential request metadata, such as HTTP method, URL, status code, and processing time, in a structured JSON format:

```
@Component
public class RequestLoggingFilter extends OncePerRequestFilter {
    private static final Logger logger = LoggerFactory.getLogger(
        RequestLoggingFilter.class);
    public static final String X_CORRELATION_ID_HEADER =
        "X-Correlation-Id";

    @Override
    protected void doFilterInternal(HttpServletRequest request,
        HttpServletResponse response, FilterChain filterChain)
            throws ServletException, IOException {

        // Generate or retrieve a unique correlation ID
        String correlationId = request.getHeader(X_CORRELATION_ID_HEADER);

        if (correlationId == null) {
            correlationId = UUID.randomUUID().toString();
        }
        MDC.put("correlationId", correlationId);

        // Capture the request start time
        long startTime = System.currentTimeMillis();

        try {
            // Proceed with the request
            filterChain.doFilter(request, response);
```

```
        } finally {
            // Capture the request end time
            long duration = System.currentTimeMillis() - startTime;

            // Log the structured request information
            logRequestDetails(request, response, duration);

            // Clean up MDC after the request
            MDC.clear();
        }
    }

    private void logRequestDetails(HttpServletRequest request,
        HttpServletResponse response, long duration) {
        // Extracting useful metadata for structured logging
        String httpMethod = request.getMethod();
        String requestUri = request.getRequestURI();
        String queryString = request.getQueryString() !=
            null ? "?" + request.getQueryString() : "";
        String correlationId = MDC.get("correlationId");
        int statusCode = response.getStatus();

        // Structured JSON log example
        try {
            String logEntryJson = new ObjectMapper().writeValueAsString(
                    createLogEntry(correlationId, httpMethod,
                    requestUri + queryString, statusCode, duration)
            );
            logger.info(logEntryJson);
        } catch (JsonProcessingException e) {
            logger.error("Failed to convert log entry to JSON", e);
        }
    }

    private Map<String, Object> createLogEntry(
        String correlationId, String method, String url, int status,
```

```
            long duration) {
            Map<String, Object> logEntry = new HashMap<>();
            logEntry.put("timestamp", Instant.now().toString());
            logEntry.put("level", "INFO");
            logEntry.put("correlationId", correlationId);
            logEntry.put("method", method);
            logEntry.put("url", url);
            logEntry.put("status", status);
            logEntry.put("duration", duration + "ms");

            return logEntry;
        }
    }
```

So, what is this filter doing?

- **Choosing the right log level**: The filter logs complete requests at the `INFO` level, which is appropriate for general application flow information. If a request encounters an error, it could be logged at `ERROR` in other components, such as exception handlers.
- **Structured logging**: The filter uses structured logging to log information in JSON format, including fields like `correlationId`, `method`, `url`, `status`, and `duration`. Structured logging allows for easier parsing, searching, and aggregating in centralized logging tools.
- **Avoiding sensitive data**: The filter avoids logging the request body directly, which could contain sensitive information like passwords. If needed, further filtering can exclude or mask sensitive data in headers or query parameters.
- **Capturing contextual information**: The filter captures relevant metadata for each request, including the HTTP method, URL, status code, and duration. This provides valuable context for debugging and performance analysis.
- **Using correlation IDs**: The filter generates a correlation ID (if one is not already present) and stores it in the **Mapped Diagnostic Context (MDC)**. This ensures that the correlation ID is added to all subsequent logs within the request's lifecycle, enabling end-to-end tracking across services.

Mapped Diagnostic Context (MDC)

Mapped Diagnostic Context (MDC) is a feature in logging frameworks like SLF4J (with Logback) and Log4j that allows developers to store and retrieve contextual information per thread. This context information is automatically included in log messages, making it easier to track related logs across different parts of an application.

With this logging filter in place, here is an example of what a log entry might look like:

```
{
    "duration": "157ms",
    "method": "POST",
    "level": "INFO",
    "correlationId": "71ef4140-f3a6-488f-ba30-b2a31ac507df",
    "url": "/orders",
    "timestamp": "2024-12-09T18:08:00.719019511Z",
    "status": 201
}
```

But to have the `correlationId` spread to other services and enable the power of tracking the request across multiple services, we need to update the header with the proper newly generated value from `correlationId`, adding it before the request is sent. We will do it using `BeanPostProcessor`:

```
@Component
public class ProductsApiBeanPostProcessor implements BeanPostProcessor {

    @Override
    public Object postProcessBeforeInitialization(Object bean,
        String beanName) throws BeansException {
        if (bean instanceof ProductsApi) {
            ProductsApi productsApi = (ProductsApi) bean;
            ApiClient apiClient = productsApi.getApiClient();
            String correlationId = MDC.get("correlationId");
            if (correlationId != null) {
                apiClient.addDefaultHeader("X-Correlation-Id",
                    correlationId);
            }
            productsApi.setApiClient(apiClient);
```

```
        }
        return bean;
    }
}
```

As an example, in the Order Management API, before calling the `productsApi` into the `ProductsQueryUseCaseImpl` implementation, we will make a call to the `beanPostProcessor` to have the `ApiClient` bean updated with the `correlationId` set in the header from the request.

The `ApiClient` is the REST client that was generated by the OpenAPI plugin using the Product API specification and will be used for every call to the Product API from the Order Management API. Here is the updated version of the class:

```
@Service
public class ProductsQueryUseCaseImpl implements ProductsQueryUseCase {

    Logger logger = LoggerFactory.getLogger(
        ProductsQueryUseCaseImpl.class);

    private final ProductsApi productsApi;
    private final ProductsApiBeanPostProcessor beanPostProcessor;

    public ProductsQueryUseCaseImpl(ProductsApi productsApi,
        ProductsApiBeanPostProcessor beanPostProcessor) {
        this.productsApi = productsApi;
        this.beanPostProcessor = beanPostProcessor;
    }

    @Override
    public ProductDetails getProductById(String productId) {
        try {
            beanPostProcessor.postProcessBeforeInitialization(productsApi,
                "productsApi");
            ProductOutputDto product = productsApi.getProductById(
                productId);
            return new ProductDetails(product.getSku(),
                product.getPrice());
        } catch (Exception ex) {
            logger.error("Error getting product with id {}", productId,
```

```
                            ex);
                throw new EntityNotFoundException(
                    "Product not found with id " + productId);
            }
        }
    }
```

This ensures that the `correlationId` is propagated with every service request. The filter in the called service will read the `correlationId` and include it in the logs, enabling you to uniquely track requests across the services.

Additional functionality can be added to the filter to capture even more detailed logging information:

- **Log-specific headers**: Capture headers like `User-Agent` or `Authorization`, but exclude or mask sensitive details.
- **Conditional logging for error responses**: Modify the filter to log `4xx` and `5xx` responses at `WARN` or `ERROR` levels for easier error tracking.
- **Error handling**: Combine this filter with a global exception handler to capture and log unhandled exceptions, leveraging the correlation ID to tie error logs to their originating requests.

This logging filter implements best practices and creates consistent, structured logs across the application, making it easier to monitor, troubleshoot, and analyze incoming API requests.

And to effectively track requests across multiple services, implementing tracing is essential. It helps maintain a clear trace and simplifies troubleshooting in distributed systems. In the next section, we will explore how to achieve this.

Implementing service tracing in distributed systems

In distributed systems, where a request might span multiple services, **distributed tracing** provides visibility into how a request moves through various components.

Let's begin by understanding what we mean by distributed tracing.

What is distributed tracing?

Distributed tracing allows you to follow the lifecycle of a request as it flows from one service to another. This helps you see where delays or errors occur. In tracing terminology, each step in the request's journey is called a span, and a trace is the entire set of spans associated with a request.

For example, imagine a request comes into your system to create a new user. This request might touch on the following services:

1. API gateway
2. Authentication service
3. User service (to create the user in the database)
4. Notification service (to send a welcome email)

Each of these steps is a span, and all the spans together form a trace.

Distributed tracing tools, such as Zipkin or Jaeger, can visualize the trace and highlight which services or steps are causing delays.

Next, we will understand how each trace is identified uniquely by adding a trace ID to each request.

Using trace IDs for end-to-end request tracking

Just like correlation IDs help in logs, trace IDs are unique identifiers attached to each request, allowing for end-to-end request tracking" you to track that request across multiple services. The difference is that trace IDs are automatically managed by tracing systems and include timing information.

In Spring, the Micrometer Tracing library automatically generates trace IDs for each request and propagates them across service boundaries. These IDs are included in the logs and tracing systems, allowing you to correlate logs and traces for detailed troubleshooting.

In a Spring Boot application, Micrometer Tracing generates the following log message:

```
{
  "timestamp": "2024-10-23T13:00:00Z",
  "traceId": "b8e3fbe5cd34fbe5",
  "spanId": "af18fbe5cd34ab23",
  "level": "INFO",
  "service": "user-service",
  "message": "Created new user with ID 123"
}
```

`traceId` helps you connect this event to other related events across different services.

Now, let us learn how to implement the tracing feature with Micrometer.

Implementing tracing using Micrometer

With the release of Spring Boot 3.x, Spring Cloud Sleuth has been replaced by Micrometer Tracing for tracing support. Micrometer Tracing is fully compatible with Spring Boot 3.x and offers a more modern, flexible way to implement distributed tracing in your applications.

In this section, we will walk through how to implement distributed tracing in a Spring Boot application using Micrometer Tracing, enabling you to track requests across services and get detailed insights into their performance.

Setting up Micrometer Tracing in Spring Boot

To implement tracing in a Spring Boot application using Micrometer Tracing, follow these steps:

1. **Add Micrometer Tracing dependencies**:

   Micrometer Tracing is part of the Micrometer ecosystem, and it integrates easily with Spring Boot 3.x. To enable Micrometer Tracing in your project, add the necessary dependencies to your `pom.xml` (if using Maven):

   ```
   <dependency>
       <groupId>io.micrometer</groupId>
       <artifactId>micrometer-observation</artifactId>
   </dependency>
   <dependency>
       <groupId>io.micrometer</groupId>
       <artifactId>micrometer-tracing-bridge-brave</artifactId>
   </dependency>
   <dependency>
       <groupId>org.springframework.boot</groupId>
       <artifactId>spring-boot-starter-actuator</artifactId>
   </dependency>
   ```

   Here is what these dependencies are used for:

   - `micrometer-observation`: Provides the core Observation API that serves as a facade for metrics, logging, and tracing. It allows you to instrument code once and get multiple observability benefits, focusing on what you want to observe rather than how to implement it.

- `micrometer-tracing-bridge-brave`: Bridges the Micrometer Observation API to Brave, enabling distributed tracing capabilities. This dependency is responsible for creating and propagating trace and span IDs across service boundaries, which is what adds the trace context to your logs.
- `spring-boot-starter-actuator`: Provides production-ready features to help monitor and manage your application. It includes endpoints for health checks, metrics, and other operational data. This starter automatically configures the observability infrastructure when combined with the Micrometer dependencies.

Together, these dependencies enable comprehensive observability with metrics, tracing, and health monitoring in a Spring Boot microservices architecture.

2. **Configure Micrometer Tracing**:

   Once you've added the dependencies, Micrometer Tracing is automatically configured in Spring Boot. By default, Micrometer will instrument your HTTP requests, generating trace IDs and span IDs for each incoming request. These IDs are propagated across service boundaries.

   To enable tracing fully, you may need to configure how traces are exported. For example, to export traces to Zipkin, add the following configuration to your `application.yml`:

```
management:
  tracing:
    enabled: true
    sampling:
      probability: 1.0 # Enables 100% sampling of traces
    zipkin:
      enabled: true
      endpoint: http://localhost:9411/api/v2/spans
logging:
  pattern:
    level: '%5p [${spring.application.name:},%X{traceId:-
},%X{spanId:-}]'
```

   In this configuration, we have the following parameters:

   - `management.tracing.enabled=true`: Enables tracing for the application.
   - `management.tracing.sampling.probability=1.0`: Ensures that all requests are traced (for production, you might want to adjust this for performance reasons).

- `management.tracing.zipkin.enabled=true`: Enables exporting traces to Zipkin.
- `management.tracing.zipkin.endpoint`: Specifies the URL of your Zipkin server for trace collection.
- `logging.pattern.level`: Specifies a pattern on which each log will be presented. Here, we are setting it to log as follows: application name, traceId, spanId. Note that the format for getting the values in the logs for the trace ID and the span Id can differ from library to library.

3. **Configuring RestClient for tracing**

   To guarantee that the generated trace is propagated through the called services, you need to configure your HTTP client to propagate the trace context. This is done by configuring `RestClient` to use the one that is created by Spring, using the dependencies and configurations done in steps 1 and 2.

   Since we are using `OpenAPI` in this book to generate the client that queries the Products API from the Order Management API, you need to override the generated `RestClient` from OpenAPI with the one from instantiated in Spring.

   To do that, you need to properly set the beans in the configuration. In the Order Management API, these configurations are done on the `ProductsApiConfiguration` class, under the `adapter.outbound.rest` package:

```
@Configuration
public class ProductsApiConfiguration {

    @Bean
    public ProductsApi getProductsApi(ApiClient apiClient) {
        return new ProductsApi(apiClient);
    }

    @Bean
    public ApiClient getApiClient(RestClient restClient) {
        return new ApiClient(restClient);
    }

    @Bean
    public RestClient getRestClient(
        ObservationRegistry observationRegistry,
```

```
        RestClientBuilderConfigurer configurer) {
        return configurer.configure(RestClient.builder())
                .requestFactory(customClientHttpRequestFactory())
                .observationRegistry(observationRegistry)
                .build();
    }

    private ClientHttpRequestFactory
customClientHttpRequestFactory() {
        Duration duration = Duration.ofSeconds(6);
        ClientHttpRequestFactorySettings settings =
ClientHttpRequestFactorySettings.DEFAULTS
                .withConnectTimeout(duration)
                .withReadTimeout(duration);

        return ClientHttpRequestFactories.get(settings);
    }

    @Bean
    public SpanHandler logSpanHandler(){
        return new LogSpanHandler();
    }

    static final class LogSpanHandler extends SpanHandler {
        final Logger logger = Logger.getLogger(
            ProductsApiConfiguration.class.getName());

        LogSpanHandler() {
        }

        public boolean end(TraceContext context, MutableSpan span,
            SpanHandler.Cause cause) {
            if (!this.logger.isLoggable(Level.INFO)) {
                return false;
            } else {
                this.logger.info(span.toString());
                return true;
            }
```

```
        }

        public String toString() {
            return "LogSpanHandler{name=" + this.logger.getName() +
"}";
        }
    }
}
```

This code configures the `RestClient` bean with a custom `ClientHttpRequestFactory` and an `ObservationRegistry` from the imported `micrometer-observation` dependency.

Then, it adds the configured `RestClient` bean into the `ApiClient` bean, followed by the configured `ApiClient` bean into the `ProductsApi` bean. We use the `ProductsApi` bean to call the external Products API from the Order Management API.

This is how we ensure that the `traceId` and `spanId` values are generated and that `traceId` is properly propagated over all the services that we call from the Order Management API.

But also, after that, we are configuring a custom `SpanHandler` with a `LogSpanHandler`. That is used to log useful information from the trace context into the application logs. Information like the duration, the request origin, its timestamp, and the `traceId` give us various data to monitor and troubleshoot the application in production. In the next section, you will see a real example logged by this `SpanHandler`.

What are beans?

In Spring, beans are simply Java objects that are created, managed, and configured by the Spring **IoC (Inversion of Control)** container. Think of them as the building blocks of your application. When you define a class as a bean, you're telling Spring to take responsibility for instantiating it, handling its dependencies, and managing its lifecycle. You can define beans using annotations like `@Component`, `@Service`, or `@Bean`, or through XML configuration. Once registered, these beans can be automatically "wired" together, meaning Spring will inject dependencies between them without you having to manually create and connect objects. This approach makes your code more modular, easier to test, and less coupled.

Viewing trace data

Once your application is instrumented with Micrometer Tracing, you can view the trace data in a distributed tracing tool like Zipkin or Jaeger. These tools allow you to visualize traces and spans, helping you diagnose performance bottlenecks or failures across services.

Just to understand the difference between traces and spans:

- The trace would include multiple spans representing the various services and operations involved in a request.
- Each span includes timing data, enabling you to identify slow services or problematic operations.

When Micrometer Tracing is implemented in a Spring Boot application, the logs will include additional fields such as `traceId` and `spanId`. These fields help you correlate logs across services in a distributed system. The `traceId` remains the same for the entire lifecycle of a request across different services, while each service or operation within a service gets its own `spanId`.

Here is an example of how the logs will look with Micrometer Tracing enabled and properly configured, calling the Products API from the Order Management API. It follows the logging pattern that we defined in the `applications.yml` file:

```
2025-03-19T16:45:39.207-03:00  INFO [order-management-api,67db1edfd85f4
2d21368a69936519fd1,1368a69936519fd1] 24477 --- [order-management-api]
[nio-8090-exec-1] [67db1edfd85f42d21368a69936519fd1-1368a69936519fd1]
c.p.o.a.o.rest.ProductsQueryUseCaseImpl  : Getting product with id AA00001
```

These are the elements contained in this log:

- `2025-03-19T16:45:39.207-03:00`: **Timestamp** – When this log entry was created
- `INFO`: **Log level** – Indicates an informational message (not an error)
- `[order-management-api,67db1edfd85f42d21368a69936519fd1,1368a69936519fd1]`: Defined pattern logging, containing:
    - **Service name** – Identifies which microservice generated the log
    - **Trace Id** – Unique identifier tracking the request across all services
    - **Span Id** – Identifies this specific operation within the trace
- `24477`: **Process ID** – The operating system's identifier for this application instance
- `---`: **Separator** – Visual divider in the log format

- `[order-management-api]`: **Application name** – Repeats the service name for readability
- `[nio-8090-exec-1]`: **Thread name** – The specific execution thread handling this request
- `[67db1edfd85f42d21368a69936519fd1-1368a69936519fd1]`: **Correlation ID** – Combined `traceId-spanId` for easy request tracking
- `c.p.o.a.o.rest.ProductsQueryUseCaseImpl`: **Logger name** – The class that generated this log (shortened)
- `Getting product with id AA00001`: **Log message** – Simple text description of the operation being performed, showing the product ID being requested

This is how Micrometer Tracing enables the logs in the application to make it possible to track requests between applications.

But if you want to have even more details in your logs, you will get them from the `LogSpanHandler` that we configured along with the beans in the configuration section.

Let us look at a specific log generated by the `LogSpanHandler`:

```
2025-03-19T16:33:01.897-03:00  INFO [order-management-api,67db1beb2f77e
de88c04d7187d10b32c,8c04d7187d10b32c] 22620 --- [order-management-api]
[nio-8090-exec-1] [67db1beb2f77ede88c04d7187d10b32c-8c04d7187d10b32c]
c.p.o.a.o.rest.ProductsApiConfiguration  :
{
    "traceId": "67db1beb2f77ede88c04d7187d10b32c",
    "parentId": "8c04d7187d10b32c",
    "id": "b362a6315dcb8bd9",
    "kind": "CLIENT",
    "name": "http get",
    "timestamp": 1742412781886860,
    "duration": 9204,
    "localEndpoint": {
        "serviceName": "order-management-api",
        "ipv4": "192.168.96.1"
    },
    "tags": {
        "application": "order-management-api",
        "client.name": "localhost",
        "exception": "none",
        "http.url": "http://localhost:8080/api/products/AA00001",
```

```
        "method": "GET",
        "outcome": "SUCCESS",
        "status": "200",
        "uri": "/api/products/{productId}"
    }
}
```

This specific log is the one generated by the `LogSpanHandler` configuration that adds all the information from the trace context.

The beginning of this log follows the same structure defined in the logging pattern shown previously, but what differs here is the information contained in its body, which is in JSON format.

Let us understand each of the elements inside of this JSON body generated by the `LogSpanHandler` and how they can help us in the observability of our applications:

- `traceId`: **Distributed trace identifier** – Links all spans across services for this request
- `parentId`: **Parent span ID** – This field identifies the parent span from which the current span originated. If the current service or operation was triggered by another service, the parent span ID helps trace the hierarchy of calls between services.
- `id`: **Span ID** – Unique identifier for this specific operation
- `kind`: **Span type** – `"CLIENT"` means the outbound request to another service
- `name`: **Operation name** – Describes what action was performed
- `timestamp`: **Start time** – When this operation began (in microseconds)
- `duration`: **Execution time** – How long the operation took (6.246ms)
- `localEndpoint`: **Service information** – Details about the originating service
- `tags`: Contextual metadata containing
    - **Application name** – Service identifier
    - **Client name** – Target server hostname
    - **Exception** – Error status (none means successful)
    - **HTTP URL** – Full URL that was called
    - **Method** – HTTP verb used (`GET`)
    - **Outcome** – Result category (`SUCCESS`)
    - **Status** – HTTP response code (`200 = OK`)
    - **URI** – Request path pattern with path variables

If you noticed, we talked about `parentSpanId` in this example. Let us understand better how this relates to `spanId` and how this can be useful in monitoring the performance of distributed systems.

Understanding parentSpanId and spanId

In distributed tracing, every request generates a trace, which consists of multiple spans. A span represents a single unit of work, such as a service call, database query, or a specific business process within a service. Each span includes a unique identifier, called the `spanId`, and a `parentSpanId` that links it to the span from which it originated. This parent-child relationship helps to visualize how requests propagate through different services in a distributed system.

In sum, a `spanId` is a unique identifier for the current operation or service, and `parentSpanId` is the `spanId` of the calling operation or service. This field links spans together, showing which service called another.

Using these IDs, tracing tools like Zipkin or Jaeger can display a complete timeline of the trace, revealing the structure and timing of each request across services.

To visualize the relationship between `spanId` and `parentSpanId`, let's walk through an example trace for a user registration request in an e-commerce application, where each service involved in the trace has its own `spanId` and, if applicable, a `parentSpanId`.

1. An API gateway receives the initial request and generates a `traceId` and `spanId`.
    - The `User` service handles the user registration, with a span linked to the API gateway's `spanId` as its `parentSpanId`.
2. A `Notification` service sends a welcome email to the user. This service's span has the `spanId` of the `User` service as its `parentSpanId`.

Below is a simplified visualization of what this trace might look like in a tool like Jaeger:

Span Name	Service	spanId	parentSpanId	Start Time	Duration
`register_user`	API Gateway	`span1`	-	0ms	15ms
`create_user_record`	User	`span2`	`span1`	5ms	40ms
`send_email`	Notification	`span3`	`span2`	25ms	30ms

In this visualization:

- The `register_user` span from the API gateway is the root of the trace. It has no `parentSpanId` because it starts the trace.
- The `User Service` span (`create_user_record`) is a child of the `API Gateway` span, so it references `span1` as its `parentSpanId`.
- The `Notification` service span (`send_email`) is a child of the `User Service` span and has `span2` as its `parentSpanId`, indicating it was triggered by the user creation process.

The tracing tool displays the parent-child hierarchy as a timeline to visualize request propagation. Below is a diagram that matches this hierarchy, showing how each service relates in time:

```
0ms        15ms         40ms           70ms
|-----------|------------|--------------|
| API       | User       | Notification |
| Gateway   | Service    | Service      |
| register_ | create_    | send_        |
| user      | user       | email        |
```

In a trace visualization tool:

- The `API Gateway` span (`span1`) initiates the request.
- `User Service` (`span2`) begins shortly after `API Gateway`, and it takes more time as it performs operations like database insertion, for example.
- `Notification Service` (`span3`) starts after `User Service` completes the user creation. The duration of each span indicates how long each operation took.

Parent-child hierarchy insights

This trace hierarchy is useful for:

- **Identifying bottlenecks**: If `User Service` took unusually long, it would appear as a longer bar in the timeline, prompting an investigation.
- **Tracing errors**: If an error occurred in `Notification Service`, you could see it in the trace and quickly trace it back to the originating request from `User Service`.
- **Understanding dependencies**: By looking at the parent-child structure, you can see how each service depends on others and the sequence of operations.

This visualization and the `spanId` and `parentSpanId` relationship allow software engineers, architects, and system analysts to understand the flow and timeline of each request across multiple services, helping optimize performance, troubleshoot issues, and gain insights into the system's behavior.

Logs across multiple services

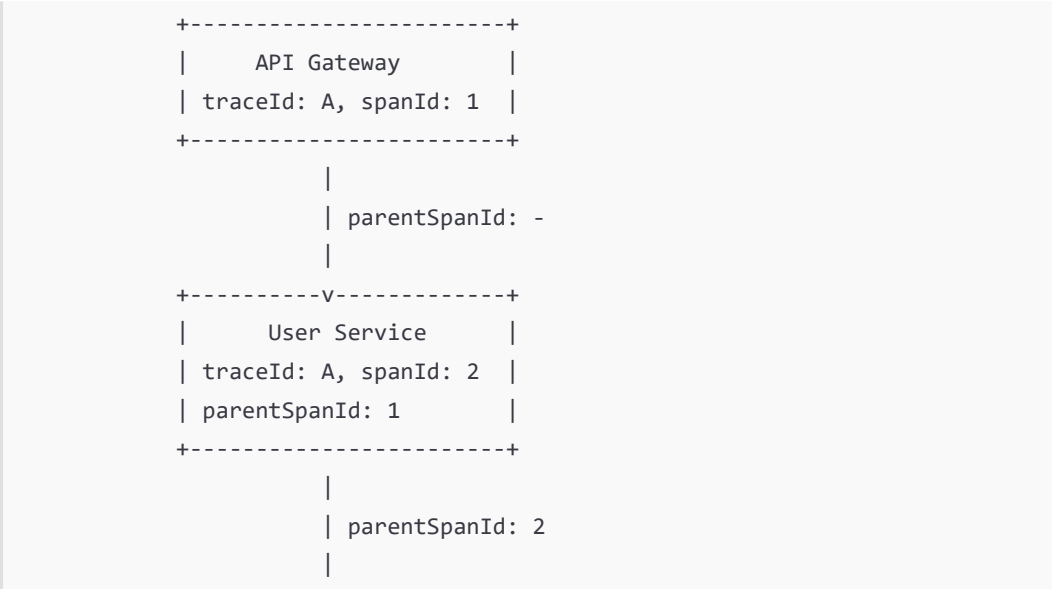
When a request flows through multiple services in a distributed system, each service logs its part of the request independently. By correlating these logs through unique identifiers like `traceId` and `spanId`, we can connect individual logs across services to form a complete picture of the request's journey. This end-to-end visibility is crucial for understanding how services interact, identifying bottlenecks, and troubleshooting errors.

In this example, a User Registration request passes through three services:

1. **API Gateway**: Receives the initial request and routes it to the appropriate backend service.
2. **User Service**: Processes the registration by creating a user record in the database.
3. **Notification Service**: Sends a welcome email to the user upon successful registration.

Each service logs its part of the request using the same `traceId` to correlate logs. The `spanId` is unique within each service, while the `parentSpanId` links it back to the calling service.

The diagram below shows how the request moves through each service, with corresponding logs identified by numbers that correlate to the example logs below:



```
          +----------v-------------+
          | Notification Service   |
          | traceId: A, spanId: 3  |
          | parentSpanId: 2        |
          +------------------------+
```

Each log entry shown below is marked with a number corresponding to the steps in the diagram above. By following `traceId`, `spanId`, and `parentSpanId`, we can see how each service is connected within the trace, enabling us to reconstruct the request's journey.

Next, let us see examples of logs on each service this trace is going through.

API Gateway log

When the API gateway receives the request, it generates a new `traceId` (A) and its own `spanId` (1).

```
{
  "timestamp": "2024-10-23T10:00:00Z",
  "level": "INFO",
  "service": "API Gateway",
  "traceId": "A",
  "spanId": "1",
  "parentSpanId": null,
  "message": "Received request for user registration"
}
```

User Service log

The User Service processes the registration, using the `traceId (A)` to connect it to the original request. The `User Service` log entry has a unique `spanId (2)` and references the API Gateway's `spanId (1)` as its `parentSpanId`.

```
{
  "timestamp": "2024-10-23T10:00:05Z",
  "level": "INFO",
  "service": "User Service",
  "traceId": "A",
  "spanId": "2",
  "parentSpanId": "1",
  "message": "Processing user registration"
}
```

Notification Service log

After the User Service completes the user creation, it triggers the Notification Service to send a welcome email. The `Notification Service` log entry includes the `traceId` (A) to maintain continuity, generates its own `spanId` (3), and uses the User Service's `spanId` (2) as its `parentSpanId`.

```
{
  "timestamp": "2024-10-23T10:00:10Z",
  "level": "INFO",
  "service": "Notification Service",
  "traceId": "A",
  "spanId": "3",
  "parentSpanId": "2",
  "message": "Sending welcome email"
}
```

In this example, the `traceId` (A) remains the same across all services, linking the logs together to represent the entire request flow. Each log's `spanId` and `parentSpanId` establish a parent-child relationship, showing how each service is connected in the sequence. Here's how these logs work together:

- **Log 1 (API Gateway)**: Initiates the request with `traceId` (A) and `spanId` (1).
- **Log 2 (User Service)**: Continues the request, referencing the API Gateway's `spanId` (1) with its `parentSpanId` and creating a new `spanId` (2) for itself.
- **Log 3 (Notification Service)**: Completes the flow by linking back to the User Service's `spanId` (2) and creating its own `spanId` (3).

Using the combination of `traceId`, `spanId`, and `parentSpanId`, we can follow the lifecycle of the user registration request as it moves from service to service, providing a clear and structured view of the request's journey.

The logs are useful in the following ways:

- **End-to-end traceability**: By searching for logs with the same `traceId`, you can trace a request across different services (API Gateway, User Service, and Notification Service) and see how the request was handled at each step.
- **Service dependencies**: The `parentSpanId` helps you understand how services are connected. In the example above, the Notification service was called by the User service, which was triggered by the API Gateway. The logs show the hierarchy of calls.

- **Performance insights**: Comparing the timestamps across spans can give you insights into performance bottlenecks. For instance, you can measure how much time each service took to handle the request by comparing the timestamps.

Micrometer Tracing enriches your logs with essential trace and span data that allows you to track requests across distributed systems. This traceability simplifies troubleshooting and helps you visualize the flow of requests, making it easier to detect performance issues or service failures. By integrating this data with tools like Zipkin or Jaeger, you can also visualize traces in real time, further enhancing your observability strategy.

Visualizing traces with Zipkin

To be able to run a local Zipkin instance on your machine, create the following `docker-compose.yml` file, which will automatically download and configure your local Zipkin instance with the following content:

```
services:
  zipkin:
    image: ghcr.io/openzipkin/zipkin-slim:${TAG:-latest}
    container_name: zipkin
    environment:
      - STORAGE_TYPE=mem
      - MYSQL_HOST=mysql
    ports:
      - 9411:9411
```

This file defines from where we are getting the Docker image and the version, that is the latest available. Also, we will run this at memory, using the default port `9411`.

To run this container, you need Docker installed on your system. Once it is installed, open a console in the same directory where this file is saved and run the following command:

```
docker compose up
```

Once the application is running, go to your local running Zipkin's UI (`http://localhost:9411`) to view the traces. You should see a graphical representation of the trace, showing the `traceId`, `spanId`, and the parent-child relationships across services.

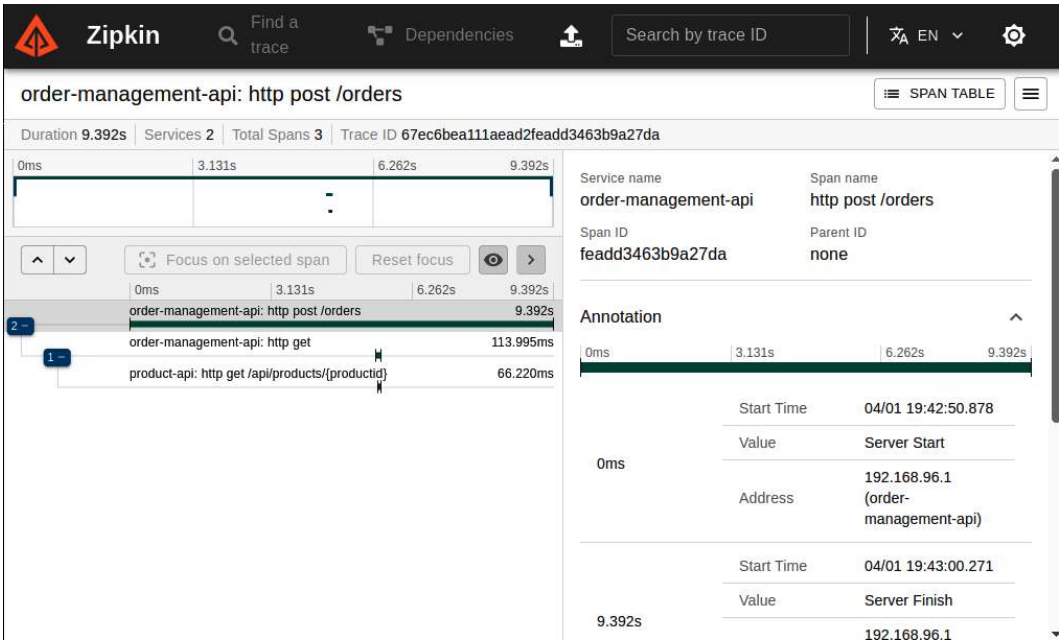


Figure 9.1 – Complete trace journey in Zipkin dashboard with Micrometer

Adding custom spans

In addition to the automatic tracing of HTTP requests, you may want to create custom spans to trace specific operations within your services. For example, you can trace important business logic or database queries.

To create custom spans, inject the `Tracer` into your service and use it to manually create and manage spans:

```
import io.micrometer.tracing.Tracer;
import io.micrometer.tracing.Span;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class UserService {

    @Autowired
    private Tracer tracer;
```

```
    public void createUser(String userId) {
        Span newUserSpan = tracer.nextSpan().name("createUser").start();
        try (Tracer.SpanInScope ws = tracer.withSpan(newUserSpan.start()))
{
            // Business logic for creating a user
            // Simulate a process that takes time, e.g., a database call
            Thread.sleep(1000);
            System.out.println("User created with ID: " + userId);
        } catch (Exception e) {
            newUserSpan.error(e);
        } finally {
            newUserSpan.end();
        }
    }
}
```

In this example, we use `Tracer` to create a custom span called `createUser`, which tracks the execution of the user creation process. The span is manually started with `start()` and completed with `end()`. We also ensure that any exceptions are captured in the span by calling `newUserSpan.error(e)`.

Next, let us understand how we can extract metrics from the tracing data and how this can help us monitor the whole application environment behavior.

Metrics from tracing data

Metrics for RESTful web services are essential for evaluating and optimizing the performance of these services. These metrics provide insights into how efficiently APIs handle requests, process data, and deliver responses.

This helps in ensuring that RESTful web services operate smoothly, providing a better user experience and meeting business objectives.

With Micrometer Tracing in place, you can extract meaningful metrics from your trace data. Metrics give you quantitative insights into your API's performance and health.

Metrics like latency, throughput, and error rates are key to understanding how your REST API performs under load. These metrics help detect slow services, overloaded endpoints, or frequent errors that need to be addressed.

Types of metrics to monitor

Some common metrics to track in REST APIs include:

- **Request duration**: How long does it take for the API to respond?
- **Request count**: The number of requests served over a period.
- **Success/failure rate**: Percentage of successful vs. failed requests.
- **HTTP error codes**: Count of 4xx and 5xx responses.

You can configure Micrometer to track these metrics automatically. For example, to track request duration, add the following configuration:

```
management.metrics.web.server.request.autotime.enabled=true
```

Metrics examples:

- **Latency**: Average time it takes for a request to complete (e.g., 200ms).
- **Throughput**: The system processes an average of 150 requests per second during peak hours.
- **Error Rate**: Percentage of requests that fail (e.g., 5% of requests return a 500 error).

Metrics can help identify performance bottlenecks. For instance, if one of your API endpoints consistently has a higher latency than others, it might indicate a need to optimize database queries, improve caching, or refactor code.

For example, if the `/api/v1/users` endpoint shows an average response time of 500ms, but other endpoints respond in under 100ms, you can use tracing data to find out where the delay occurs (e.g., in a database query or a third-party API call).

Viewing metrics with Micrometer

Micrometer integrates with Prometheus and Grafana to visualize your metrics in real-time dashboards. This allows you to create custom views and alerts based on the performance of your API.

For example, in Grafana, you can create a dashboard that visualizes the latency of your API endpoints over time using time series graphs. These graphs help you spot trends and optimize performance by highlighting periods of increased latency, which might indicate bottlenecks or resource constraints. For instance, you can use a line chart to display how the average response time of a specific endpoint changes over time, making it easier to identify patterns or anomalies.

Additionally, Grafana supports a variety of other visualizations that can be used to display metrics such as request counts, success/failure rates, and HTTP error codes. For example, you can use bar charts to compare the number of successful versus failed requests over a given period, or pie charts to show the distribution of different HTTP status codes returned by your API.

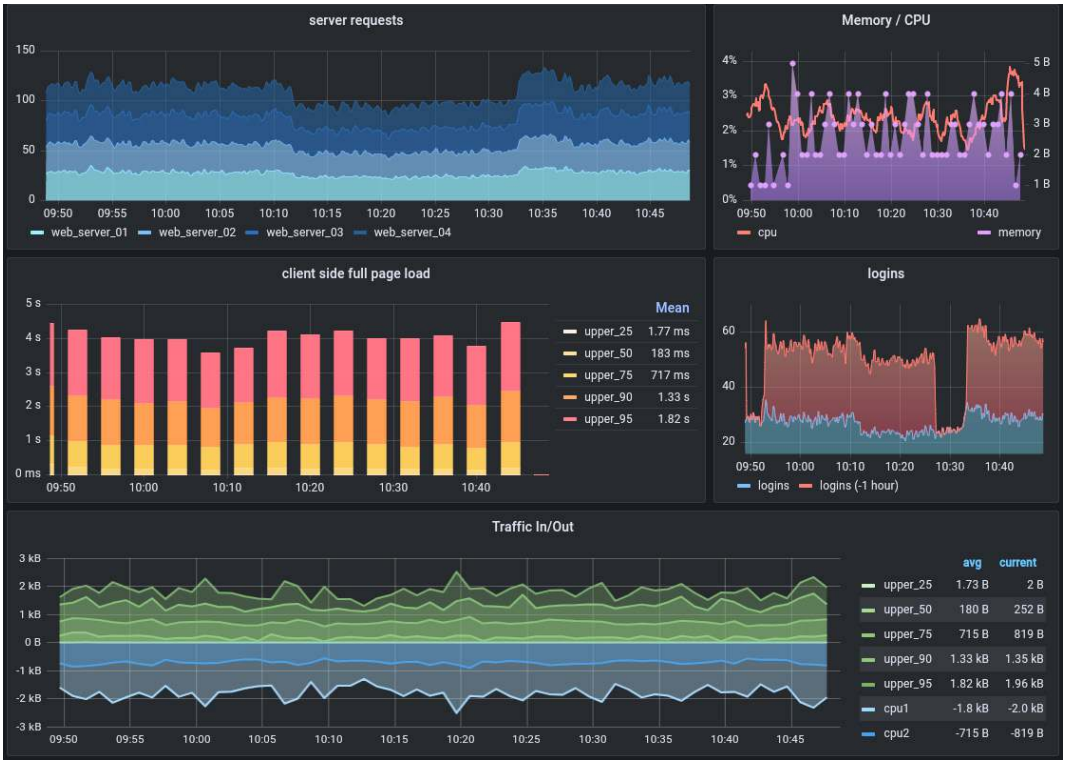


Figure 9.2 – Metrics displayed in a Grafana dashboard

Now that we have mastered Micrometer, let us look at another option for monitoring and observability, which is also open source and widely used in the market, OpenTelemetry.

OpenTelemetry for monitoring and observability

OpenTelemetry is an open-source, vendor-neutral observability framework that provides tools to collect telemetry data (logs, metrics, and traces) from your applications. It is a comprehensive standard designed to provide deep insights into distributed systems and is widely supported across languages and platforms.

OpenTelemetry unifies logs, metrics, and traces into one framework, providing a standardized way to instrument your services. It works with a variety of backends (like Prometheus, Jaeger, Zipkin, and Grafana) and supports distributed tracing across microservices.

OpenTelemetry helps you track the complete lifecycle of a request as it moves through multiple services, providing valuable insights into service performance, latency, and bottlenecks.

OpenTelemetry consists of the following components:

- **Traces**: Monitor the journey of requests across multiple services.
- **Metrics**: Collect quantitative data on service performance, such as response times and error rates.
- **Logs**: Record discrete events within the system, such as errors or warnings.

Using OpenTelemetry in Spring Boot

OpenTelemetry provides a standardized way to collect, process, and export telemetry data (logs, metrics, and traces) from your application. In a Spring Boot application, OpenTelemetry can be integrated to automatically capture tracing data across your services. Once integrated, this data can be exported to observability tools like Jaeger, Zipkin, or Grafana to visualize and monitor the flow of requests in real time.

In this section, we'll go through the steps to set up OpenTelemetry in a Spring Boot application, validate that tracing is working, and review sample logs showing the output after implementing OpenTelemetry.

1. **Add OpenTelemetry dependencies**: To integrate OpenTelemetry with Spring Boot, you'll need the OpenTelemetry SDK along with specific instrumentation dependencies for Spring and HTTP clients. Add the following dependencies to your `pom.xml`:

   Under the dependency management tag:

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>io.opentelemetry.instrumentation</groupId>
            <artifactId>opentelemetry-instrumentation-bom</
artifactId>
            <version>2.14.0</version>
            <type>pom</type>
            <scope>import</scope>
```

```
        </dependency>
    </dependencies>
</dependencyManagement>
```

After that, proceed to add the following dependencies under the dependencies tag:

```
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-tracing-bridge-otel</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<dependency>
    <groupId>io.opentelemetry.instrumentation</groupId>
    <artifactId>opentelemetry-spring-boot-starter</artifactId>
</dependency>
<dependency>
    <groupId>io.opentelemetry</groupId>
    <artifactId>opentelemetry-exporter-otlp</artifactId>
</dependency>
```

Let us understand what each of these dependencies are used for:

- `opentelemetry-instrumentation-bom`:
    - A Bill of Materials (BOM) that ensures version alignment across all OpenTelemetry dependencies.
    - Helps manage compatible versions between OpenTelemetry components and their transitive dependencies.
    - Must be imported before other BOMs (like spring-boot-dependencies) when using Maven like we are doing here.
- `micrometer-tracing-bridge-otel`:
    - Bridges Micrometer's Observation API to OpenTelemetry's tracing system.
    - Facilitates the propagation of trace context and spans between Micrometer and OpenTelemetry.

    - Essential component for enabling distributed tracing with OpenTelemetry in Spring Boot applications.

- `opentelemetry-spring-boot-starter`:
    - Provides auto-configuration for OpenTelemetry in Spring Boot applications
    - Includes built-in instrumentation for many Spring Boot features
    - Simplifies the process of instrumenting a Spring Boot application with minimal configuration
    - Particularly useful for Spring Boot Native image applications or when seeking reduced startup overhead compared to the Java agent approach
- `opentelemetry-exporter-otlp`:
    - Implements the OpenTelemetry Protocol (OTLP) exporter for sending telemetry data
    - Allows applications to export collected tracing data to OpenTelemetry Collectors or other backends
    - Supports standardized telemetry data delivery between observability tools
    - Can be configured to use either HTTP or gRPC transport protocols
- `spring-boot-starter-actuator`:
    - Adds production-grade monitoring and management features to Spring Boot applications.
    - Provides dependency management and auto-configuration for Micrometer (metrics and tracing)
    - Required foundation for both metrics and tracing capabilities in Spring Boot. This is mandatory for any tracing, either only with Micrometer or along with OpenTelemetry
- Exposes endpoints for application health, metrics, and other operational data

2. **Configuring OpenTelemetry into the application**: Let us have the following properties in the `application.yml`.

    - First, we are going to define the URL where OpenTelemetry will send the traces, and in this case, this is the path and exposed port from Jaeger that we are going to run from the `docker-compose` file that you will define next.

    ```
    tracing:
      url: http://localhost:4318/v1/traces
    ```

    - Under the `otel` tag, we are setting OpenTelemetry to not export logs or metrics, only traces. It will also work without these configurations but will throw multiple exceptions in the application log because Jaeger only reads traces, not logs or metrics.
    - If you are going to use another backend tool like Grafana that consumes the logs and the metrics, instead of setting it to none, you should add the proper configuration for the backend that you are going to use.
    - Since OpenTelemetry is compatible with a wide variety of backends, you should refer to the documentation to see how to configure it for the logs and metrics backend that you will be using.

    ```
    otel:
      logs:
        exporter: none
      metrics:
        exporter: none
    ```

    - Next, under the management tag, you have the same configuration as shown in the Micrometer section. The configuration needs to contain tracing enabled and the sampling probability at 1.0 in order to generate as many traces as possible for our testing purposes. In production environments, you should configure that with a smaller value to avoid unneeded tracing.

    ```
    management:
      endpoints:
        web:
          exposure:
            exclude: "*"
      tracing:
    ```

```
    enabled: true
    sampling:
      probability: 1.0
```

- Finally, be aware that the logging pattern with OpenTelemetry changes. Using pure Micrometer with Brave, as shown in the section Implementing tracing using Micrometer, registers the trace and the span as `traceId` and `spanId` in the MDC (Mapped Diagnostic Context), that is, from where the pattern gets its values. But OpenTelemetry registers them as `trace_id` and `span_id`. This is a slight change but if you do not take this into consideration, you will not see the tracing in your application logs.

```
logging:
  pattern:
    level: '%5p [${spring.application.name:},%X{trace_id:-
},%X{span_id:-}]'
```

3. **Configuring OpenTelemetry exporter**: Create an OpenTelemetry configuration class in your Spring Boot application to export the traces to the defined tracing URL. This setup is generally handled automatically when using `opentelemetry-spring-boot-starter`, but you can add further customization to initialize it as a bean in Spring, like we are doing here.

   This will export the traces directly to be collected by the backend, which in our example will be Jaeger this time. To achieve that, we are going to use the OtlpHttpSpanExporter class from the OpenTelemetry library imported earlier:

```
import io.opentelemetry.exporter.otlp.http.trace.
OtlpHttpSpanExporter;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class OtlpConfiguration {

    @Bean
    OtlpHttpSpanExporter otlpHttpSpanExporter(@Value(
        "${tracing.url}") String url) {
```

```
        return OtlpHttpSpanExporter.builder()
            .setEndpoint(url)
            .build();
    }
}
```

With this configuration, the `OtlpHttpSpanExporter` will export the traces directly into the defined tracing URL from the `applications.yml`, so Jaeger can read our traces directly. By default, OpenTelemetry will automatically instrument HTTP and Spring MVC requests.

1. **Validate tracing in the application**: To validate that OpenTelemetry tracing is working, we can:
    - Check traces in the exported tool: Start your Spring Boot application and use Jaeger (or the chosen backend) to view the traces. Each incoming request should appear in the tracing tool as a new trace with a unique `traceId`.
    - Observe logs with trace and span IDs: After setting up OpenTelemetry, logs should contain `traceId` and `spanId`, allowing you to correlate log entries across services.
2. **Verify logs and trace visualization**: After implementing OpenTelemetry, you should see enhanced logs with traceId and spanId for each request. Additionally, the tracing backend (Zipkin, in this case) will provide a visual representation of the trace.

Logs with OpenTelemetry Tracing

Below is an example log output from the `order-management-api` calling the `product-api` after implementing OpenTelemetry. Notice the `traceId` and `spanId` added to each log entry, following the defined logging pattern into the `application.yml` file:

```
2025-04-01T19:10:56.485-03:00  INFO [order-management-api,e809af87a330f
1ab03ccfa395e5d5864,d01a9ecf8f4e570b] 10441 --- [order-management-api]
[io-8090-exec-10] [                                                 ]
c.p.o.a.o.rest.ProductsQueryUseCaseImpl  : Getting product with id AA00001
```

It follows the same structure described in the section Viewing trace data, but here these traces and spans are being generated by OpenTelemetry.

We can also see the logs on the `product-api` side, which receives the `traceId` from the `order-management-api` and generates its own `spanId`:

```
2025-04-01T19:10:56.488-03:00  INFO [product-api,e809af87a330f1ab03ccfa3
95e5d5864,38cb8ed93e944e69] 10609 --- [product-api] [io-8080-exec-10] [
] c.p.p.a.o.d.ProductsQueryUseCaseImpl     : Getting product by id:
AA00001
```

With these IDs, each log entry can be correlated to a specific request and its journey across multiple services.

Next, we will learn how to run a local Jaeger instance to see the whole tracing in action.

Visualization example with Jaeger

To be able to run a local Jaeger instance on your machine, create the following `docker-compose.yml` file, which will automatically download and configure your local Jaeger instance with the following content:

```
services:
  jaeger:
    image: jaegertracing/jaeger:${JAEGER_VERSION:-latest}
    container_name: jaeger
    environment:
      - COLLECTOR_OTLP_ENABLED=true
    ports:
      - 4318:4318
      - 16686:16686
```

This file defines to get the latest Docker image for Jaeger. Also, here we are allocating port `4318` for tracing and `16686` for the Jaeger UI interface.

The property `COLLECTOR_OTLP_ENABLED=true` is optional on Jaeger v2 since its default is always true and mandatory on Jaeger v1. At the time of this writing, you should get above v2 while running this `docker-compose.yml` file.

To run this container, you need Docker installed on your system. Once it is installed, open a console in the same directory as where this file is saved and run the following command:

```
docker compose up
```

Once the application is running, go to your local running Jaeger's UI (`http://localhost:16686`) to view the traces. You should see a graphical representation of the trace, showing the `traceId`, `spanId`, and the parent-child relationships across services, and even showing deeper details like database `INSERT` and durations, giving a broad view of the trace.

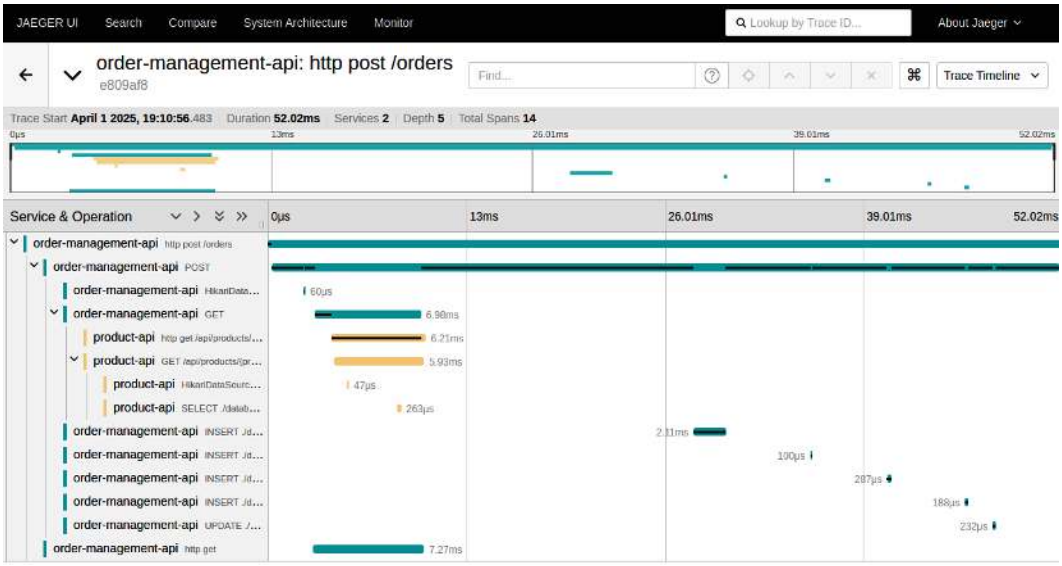


Figure 9.3 – Complete trace journey in Jaeger dashboard with OpenTelemetry

In the case of any error, it will highlight what happened and where it happened:

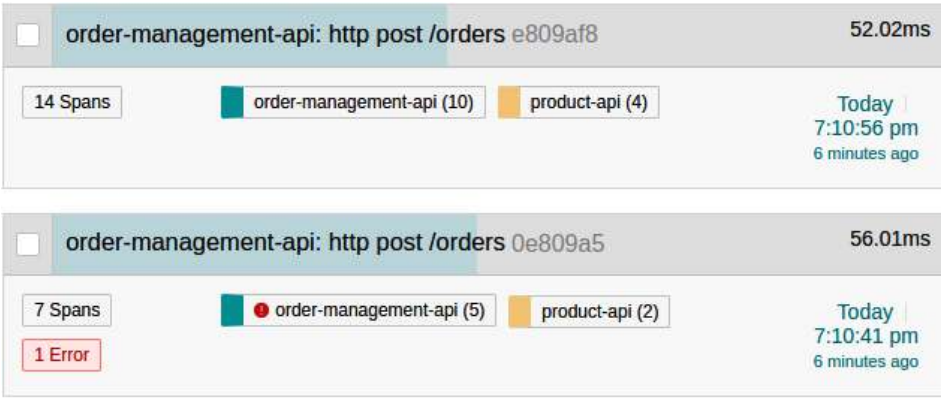


Figure 9.4 – Successful and error traces in Jaeger dashboard

Feel free to refer to this chapter repository to get the working code version and replicate the same behavior on your local machine.

By examining both the logs in the application and the trace visualization in Jaeger, you can validate that OpenTelemetry is successfully capturing traces, correlating logs with tracing data, and providing a full view of the request's journey through your distributed system. This setup not only helps troubleshoot issues but also provides insights into optimizing performance across services with the goal of monitoring distributed services.

Let us take a look at how to create custom spans with OpenTelemetry.

Creating custom spans

While OpenTelemetry automatically instruments HTTP requests, you can also create custom spans to monitor specific operations. Here's an example in which a span is manually created for a user registration process:

```
import io.opentelemetry.api.trace.Span;
import io.opentelemetry.api.trace.Tracer;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class UserService {

    @Autowired
    private Tracer tracer;

    public void registerUser(String username) {
        Span span = tracer.spanBuilder("UserService.registerUser")
                .setAttribute("username", username)
                .startSpan();

        try {
            // Simulate registration logic
            Thread.sleep(100);
            System.out.println("User registered: " + username);
        } catch (InterruptedException e) {
            span.recordException(e);
        } finally {
            span.end();
```

```
        }
    }
}
```

In this example, a custom span named "`UserService.registerUser`" is created, and a username attribute is added. The span is started with `.startSpan()` and ended with `.end()`.

Now that you have mastered the usage of OpenTelemetry as well, let us focus on some of the best practices for observability that you should be aware of.

Best practices for end-to-end observability

Observability is an ongoing process that involves logs, metrics, and traces working together to provide full visibility into your REST API's performance and behavior.

Preparing your environment to be completely covered by observability and monitoring is not an easy task, and you must consider the following topics:

Combining logs, metrics, and traces

Logs, metrics, and traces complement each other to give you a holistic view of your application:

- **Logs**: Provide detailed information about specific events.
- **Metrics**: Offer a quantitative summary of performance over time.
- **Traces**: Show the lifecycle of individual requests across services.

By using these tools together, you can quickly diagnose issues and optimize performance. For example, you might use tracing to find a slow request, logs to determine why it's slow, and metrics to track how often the issue occurs.

Alarms and notifications

With observability tools in place, you can set up alarms to notify your team when something goes wrong. For example, you can configure Prometheus to send alerts when the error rate exceeds a certain threshold or when response times spike.

In the context of RESTful microservices, AWS CloudWatch offers comprehensive monitoring capabilities that transform raw operational data into readable, near-real-time metrics stored for up to 15 months.

For example, when implementing API Gateway as the front door to your microservices architecture, CloudWatch can monitor key performance indicators such as IntegrationLatency to measure backend responsiveness, overall latency to assess API call efficiency, and cache performance metrics to optimize resource utilization. API Gateway logging, which feeds into CloudWatch Logs, provides valuable visibility into consumer access behaviors, allowing teams to understand common customer locations, analyze request patterns that might impact database partitioning, identify abnormal behavior that could indicate security concerns, and optimize configurations by tracking errors, latency, and cache performance. This monitoring framework creates a secure, easily maintainable environment that scales with growing business needs while providing actionable intelligence to continuously improve service delivery.

Continuous improvement

Observability is an ongoing process. As your system evolves, regularly review and refine your logging, tracing, and metrics collection to ensure you are capturing the most useful data. Use tools like Prometheus, Grafana, Zipkin, and OpenTelemetry to continually monitor and improve your system's performance.

Summary

In this chapter, we explored the core components and best practices for achieving effective monitoring and observability in RESTful services. Beginning with logging, we discussed the importance of structured logging for API troubleshooting, log levels to indicate severity, and correlation IDs to link requests across services. By implementing these logging practices centrally, such as with a filter in Spring Boot, we ensured consistent and secure logging across the application.

We then introduced distributed tracing, explaining how `traceId`, `spanId`, and `parentSpanId` create a parent-child relationship among services, allowing developers to track the journey of requests through a system.

Micrometer Tracing was covered as a key tool in Spring Boot 3.x for enabling and managing distributed tracing. It automatically instruments Spring Boot applications, capturing trace and span information for each request.

Micrometer Tracing integrates with multiple exporters, including Prometheus, Zipkin, and Jaeger, to send trace data to external observability platforms. With its configurable sampling and tagging, Micrometer Tracing provides granular visibility into each service, enabling efficient troubleshooting and performance optimization.

Building on tracing, we explored OpenTelemetry as a vendor-neutral observability framework that collects and correlates traces, metrics, and logs in distributed systems.

OpenTelemetry integrates smoothly with Spring Boot to provide out-of-the-box tracing for HTTP and Spring MVC requests, with added flexibility to create custom spans. We covered how to configure OpenTelemetry, validate its functionality through logging and visualization in tools like Zipkin, and observe end-to-end traces across services.

By combining logging, tracing, and OpenTelemetry with tools like Zipkin or Jaeger for visualization, we can gain a comprehensive view of each request across services.

This chapter provided foundational strategies for implementing robust observability, allowing for effective monitoring, faster troubleshooting, and insights to optimize the performance of RESTful APIs in complex, distributed environments.

In the next chapter, you will learn about scaling and performance optimization techniques, to be able to make the best out of your applications.

10

Scaling and Performance Optimization Techniques

Chapter 9 introduced skills such as quantitative measurements to observe how a system connected via APIs behaves at runtime.

That chapter provides a good basis for this chapter on **performance** and **scalability**, two non-functional requirements that are strongly concerned with time, size, and other quantitative aspects of software systems and the data they process.

We will start by explaining what developers need to know about performance and scalability in general. Then, we will dive into specifics, describing the most common strategies and techniques for improving performance and scalability. We will demonstrate some of these techniques using our Product and Order Management APIs.

We highlight the potential of Java **virtual threads** for increasing application throughput by improving CPU usage efficiency in the context of API development.

We also show that to support performance and scalability, your API should be prepared to work with specialized infrastructure components.

Lastly, we will show how load testing helps you avoid unpleasant surprises once your application starts to receive the production load. Using performance testing, developers can get information that is vital to target their optimization efforts.

By the end of this chapter, you will know how to prevent performance and scalability issues starting from the analysis phase, through appropriate API design, to subsequent optimizations triggered by the findings from load tests.

The following topics are covered in this chapter:

- Understanding performance and scalability in API development
- Applying performance optimization strategies
- Increasing the throughput with virtual threads
- Using infrastructure support
- Designing and executing effective load tests

Technical requirements

To demonstrate some of the techniques described in this chapter, we will use the example code of the Product and Order Management APIs that were developed in previous chapters. The changes that we will make to the code for this chapter can be found in the repository at `https://github.com/PacktPublishing/Mastering-RESTful-Web-Services-with-Java/tree/main/chapter10`.

Understanding performance and scalability in API development

Performance in computing refers to how efficiently a system or application executes tasks under a given workload. Efficiency has two aspects:

- **Speed of processing**: This is measured mostly in terms of response time (latency) and throughput (how many operations or how much data the system can handle per unit of time).
- **Consumption of resources**: This refers to the amount of resources required, such as CPU, memory, and network bandwidth utilization, to do the work.

Statistics are an integral part of performance measurement because it makes sense to measure performance when the system processes a large number of different requests involving different amounts of data. Therefore, the time and other resources required to fulfill a request inevitably fluctuate and depend on a number of factors, making the actual measured values of the performance characteristics virtually random in nature.

It is usually sufficient to use average values to calculate throughput. For response times, the average value is less useful because it does not capture well how the speed of response is perceived by users. The maximum value is often of particular interest because it can indicate potential issues, such as the occurrence of timeout errors. Moreover, response times exceeding a reasonable limit discourage human users from continuing to use the application.

Even more useful than the extreme values that occur very rarely are the percentile values. If we rank all response times from shortest to longest, then, for example, the 95th percentile is the time at which 95% of the measured times are shorter. In other words, there is only a 5% probability that a randomly selected request will take longer.

Scalability refers to the ability of a system to handle an increasing load (increased number of users, higher data volumes, or additional transactions) while maintaining the performance within the limits expected by users. You can see that we cannot speak about scalability without considering performance.

Scalability is one of the main motivating factors behind moving from monoliths to distributed (microservices) architectures. We can assume that our example Product API will need to handle significantly more requests than the Order Management API because not every product being browsed will be bought. We can scale the deployment of the Product API to use more instances of the service than the Order Management API.

With RESTful APIs, the tasks whose response time and throughput we are interested in are the HTTP requests. The time and computing resources needed to execute an HTTP request are composed of two parts:

- The processing needed by the API technology itself, such as serialization and deserialization of the data, network transmission, and protocol overhead
- The processing inside the service providing the API (such as executing algorithms, accessing databases, and downstream API calls), which is often influenced by the API request and response payload design, such as the amount of data processed in one request, the possibility of performing concurrent processing and data streaming, and the cacheability of the responses.

When designing APIs, you usually focus on the functional requirements because what the application should do is at the front of the users' minds. How the application is expected to perform and cope with increasing traffic is often considered obvious or implied.

It is also hard to predict the actual load, the load progression over time, and which parts of the system will be the most impacted. Donald Knuth's famous saying "*Premature optimization is the root of all evil*" warns us that we should not try to optimize everything.

On the other hand, performance and scalability considerations should be part of the design process, and you often can prevent a lot of problems with just a few simple adjustments to the APIs. Ideally, you should design the APIs to be as simple as possible to fulfill the functional requirements, while keeping the options open for future extensions and optimizations.

When in doubt, let the REST principles guide you towards a more flexible and standard solution that will likely support the performance and scalability requirements:

- Using URLs based on a resource structure reflecting the business domain
- Using correct HTTP methods and providing respective idempotency support for `GET`, `PUT`, and `DELETE`
- Using correct HTTP headers and status codes
- Using clearly defined parameters to let the client choose what data and operations are really needed
- Using appropriate `Content-Type` and `Content-Encoding` for the data (especially with large binary documents)

Having explained what performance and scalability mean for API development in general, we can move on to describe the strategies that, when applied appropriately, can have the greatest impact on the performance and scalability of your applications integrated using APIs.

Applying performance optimization and scalability improvement strategies

If you detect an existing or imminent performance or scalability problem with your API, you should be armed with the tools to solve it. Let's take a closer look at some of the strategies and techniques for optimizing performance in the following sections.

Knowing the performance requirements

As mentioned in the previous section, the performance requirements are often implied and not clearly specified by users and customers.

For some applications, performance may not be critical, but if we take the example of most e-commerce sites, an application with a response time reaching a few seconds is equivalent to an application that does not work at all because it immediately discourages potential customers from buying.

It is therefore necessary to include at least some rough performance expectations for the system to be developed. If you can be more rigorous, you should ask the customer to define **service-level agreements** (**SLAs**) specifying maximum response times or the number of requests processed for each operation. The SLAs may vary for peak and off-peak times. Remember to capture size limits for large data objects, large numbers of concurrent users, and their geographical distribution.

The collected performance requirements can be used to design load tests. The load tests will show whether the system can handle the expected load and, if not, which parts are the bottlenecks. Information about the geographical distribution of users and the quality of their network connections should be used to add simulated network latencies in the load test environment.

To design the system correctly, it is important to know whether the system should be optimized for response time (systems with human interaction) or throughput (batch processing). To achieve better (perceived) response times, it may make sense to split the API requests so that a request performs the essential operation only, delivering the essential data only. This way the user experience is improved, while the non-essential request(s) can start later and/or take longer.

In contrast to that, grouping multiple operations or data items in one request may improve the throughput by reducing the API overhead. However, you must ensure that a request does not become so big that it fails due to a timeout.

In many cases, you cannot estimate or simulate the load and environment attributes precisely enough, so you must react to performance problems that start occurring in production. To support that, your APIs should be ready to evolve without breaking the existing clients, which is the topic of *Chapter 5*.

Providing only what is really needed

Security, another non-functional requirement, is often regarded as going against performance, but in many instances, their solutions can in fact overlap. Limiting the data that we transfer in APIs is good for security and can be good for performance, too, because time and resources are not wasted on items that the API client does not need.

You should identify expensive items that are large or take a long time to get. If an expensive item is not used by all requests, you can do either of the following:

- Add a parameter to the API endpoint to only return the expensive item when the client asks for it
- Create a separate endpoint (resource) for the expensive item

For example, if our Product API stored photos of the product, we would not have to return the photos in every request for a product. Instead, we can return a list of image IDs and define a sub-resource of the product resource, returning an image by its image ID.

A common mistake is to design API resources to be a copy of the database entities. Some developers might even be tempted to use the clear antipattern of using the JPA entity classes in APIs directly without defining separate API DTOs.

Special attention should be paid to fields that define relationships between JPA entities. Exposing the whole graphs of objects resulting from following the relationships without due justification often leads to bloated APIs, creating a ticking time bomb with performance (besides security and maintainability) problems.

For example, suppose that at some point we extend the database schema of our Product API service with additional database tables and JPA entities containing information about stock availability, location, historical prices, and so on. These additional database tables could be linked to the product table using foreign keys, and the JPA model could include relationships that would map to the respective SQL joins. It would be a mistake to include all the additional information automatically in the Product API resource JSON representation, except in the unlikely scenario where business analysis confirms that all possible use cases for reading a product will always need the full set of detailed data. Instead, to support the use cases that require access to the related entities, the API could send the extra data only when a parameter is set or via a separate endpoint.

Maintaining statelessness

Statelessness (as explained in *Chapter 1*) is one of the key principles of REST architecture. All the input data needed to perform an operation should be in the URL path and the request body. The request processing must not depend on some implicit session data.

Statelessness is important for performance and scalability because it allows horizontal scaling of the service providing the API. A load balancer (briefly explained in the upcoming section on infrastructure components) can route any request to be processed by any of the service instances because the instance does not have to remember the session state (history of the previous requests belonging to the same user journey). Avoiding the session state can also help in reducing memory consumption.

Of course, the application may need to support a session-like user experience. For example, you may want to let the user create an order product by product, so the application needs to remember an incomplete order, commonly represented by a shopping cart on most e-commerce sites. To fulfill this requirement, you have two main options:

- Maintain the session state on the client (web browser) using JavaScript code, local storage, and so on.
- Define the shopping cart as a first-class entity of the backend service. This means creating a RESTful resource identified by URL for the new entity and using standard HTTP methods to manipulate it. This makes the data stored in the backend explicit, as opposed to an implicit session without a clear structure that would be harder to manage.

Limiting large collections

When your API transfers a collection of items (usually represented by a JSON list), you should try to find out the expected number of items in the collection.

If the number of items frequently exceeds a reasonable limit, you should consider implementing a pagination solution for the collection (see *Chapter 6*). The number of items in a collection and the number of items per page that calls for the introduction of pagination depend on how expensive the API operation is. For small items that are quick to retrieve, the number could be higher.

You should also think about how large a set it makes sense to work with on the client side at the same time. If the items are displayed on a screen, the user can usually not see more than a few items at the same time. You can base your pagination policy on that number.

Besides pagination, you can limit the number of items returned by ensuring that queries are sufficiently specific. For instance, in a substring-based search, you can enforce a rule that the search string must have a minimum length of three characters, which helps narrow down the results and improve efficiency.

Or, before the full data retrieval, you can first execute a cheaper database query returning only the count of the matching items, and if it exceeds a certain limit, you can stop there and return a response asking the client to provide a more specific query. Cheaper in the context of performance optimization means taking less time and fewer resources.

When deciding about the collection size limits, you should also think about the nature of the data sources your API operation needs to access. In distributed (microservices) architectures, you usually need to call another API to fulfill an API request.

If you need to make separate API calls to get some details for each item of a collection, you can try to shorten the response time by making several such API calls in parallel. However, you should limit the number of parallel API invocations to avoid overwhelming the API. The *Increasing the throughput with virtual threads* section later in this chapter may be useful for implementing concurrent API calls.

Another option is to check whether the API your service needs to call supports (or can be made to support) bulk operations, such as getting details for a list of IDs in one request. But you must be reasonable about the size of the request because requests that are too large may take too long and hit timeouts at various points along their path from the client to the server and back.

- For example, the Order Management API calls the Product API for each product to get its price. We could optimize this integration so that the Product API provides a bulk "get multiple products" endpoint. The API would take a list of product IDs as its input and return the prices for all the products at once.

Optimizing large objects

We have already mentioned that it is not wise to try to optimize everything. The Pareto principle says that the vast majority (around 80%) of (performance) problems are caused by a small minority (around 20%) of the items comprising your API.

A relatively common case where a small part of an API is responsible for most of the bytes transferred and time spent is when you want to transfer large blocks of data whose internal structure is not important to the API, such as photos, videos, or documents, which are usually encoded using binary formats. These items have a significant impact on the network transfer volume, memory consumption, and response times.

It is a good practice to avoid including large (binary) objects in structured (JSON) payloads. Instead, you can define separate resources (endpoints) for the large objects. This has the following advantages:

- The payload can use an encoding that is suitable for the large object format. Trying to embed binary data in JSON usually leads to the use of inefficient encodings such as BASE64.
- If the text-based encoding of large data objects cannot be avoided, the efficiency of the network transfer can be improved by combining it with compression, such as gzip.
- Large objects can be cached independently of other data. This makes sense given the size of the objects and because it is likely that the large objects change less frequently than the structured data.

Where possible, you should also try to limit the size of large objects. When displaying images on a client device, the performance can be improved by scaling down the images to a lower resolution on the server.

You can limit the size of large objects at the very beginning. If your application allows users to upload files, you should consider constraining the uploaded file size.

We have implemented a separate endpoint for product photos in our example Product API in the *Uploading and downloading files via the REST API* section in *Chapter 6*.

In that example code, we used a method of the `MultipartFile` class to get the file content as a byte array. We also return a byte array in the method used to download the file. This approach allows our code to stay simple, but it has the drawback that the whole file is stored in the heap memory at once, although we do not do any processing that requires having the whole file in memory.

We could optimize the memory usage by using `InputStream`/`OutputStream` instead of the byte array. Or we could use a reactive framework such as WebFlux (more on that in *Chapter 11*). However, we would have to go all the way, that is, also persisting the content using methods supporting streaming access.

Caching

Caching may be the most popular performance improvement strategy. However, caching is not a silver bullet, and designing a correct caching setup is not easy, so before adopting this strategy, you should make sure you have considered the option of improving performance by applying a proper API design, as described in the previous sections.

Caching involves a trade-off: it provides shorter response times at the cost of additional memory to store cached data and the risk of potential inconsistency if the cached data becomes stale.

Caching is closely tied to the challenge of cache invalidation—determining when data items in the cache should be removed. This may occur either because the cached data has become stale or to free up memory when it is unlikely that the cached items will be accessed again. Cache invalidation is considered one of the hardest problems in computing. The two most common ways to detect items that need to be removed from the cache are as follows:

- *Least recently used*, based on the assumption that a resource that was last used a long time ago is unlikely to be used again
- *Least frequently used*, based on the assumption that resources that have been used frequently are likely to be used again

To speed up the operations of an API, we can consider caching on the consumer (client) side of the API (caching the HTTP responses) and on the provider (server) side of the API (caching data needed to perform the operation and return a result).

Caching on the provider (server) side can take many forms: it can be data from a database, results of expensive computations, or semi-processed results of downstream API calls. Caching on the service provider side is not specific to just API implementations, and therefore, it is beyond the scope of this book. For detailed guidance, you can refer to Mastering Spring Boot 3.0 by Ahmet Meric which covers broader caching strategies within Spring-based applications. For database access using JPA/Hibernate, it is advisable to understand the concept of first- and second-level cache. For general caching at the Spring component level, you should know about Spring Cache (`https://docs.spring.io/spring-boot/reference/io/caching.html`) and the `@Cacheable` annotation.

To support the cacheability of data, the operations producing the data should be stateless, meaning we should get the same output data if the inputs are the same. This allows us to share the cached data across multiple instances of a service or multiple services using distributed caches such as Hazelcast or Redis. This way, data put in the cache by one instance of a service can be reused by another instance needing the same data.

Caching on the client side

Now, let's focus on the caching on the client side. As the client does not have access to the implementation details of the service, it must use some hints to decide for which API responses and for how long a cached response can be used instead of calling the API again. Remember that client-side caching need not be done in the end client; it can also be done by a proxy server sitting between the server and the end client.

Before we get into the details of cache control, it is important to verify the structure and granularity of the resources (endpoints) the API consists of. Consider placing data items that rarely change (they are good candidates for caching) in a separate resource from that of other items that change frequently. Only then can the caching be aligned well with the modification patterns of the data.

Requests using the `GET` method are considered cacheable by default. This rule is used by most web browsers; therefore, users sometimes need to force the browser to reload the web page (e.g., pressing *Ctrl* + *F5* on the Windows operating system) in case the page content has changed faster than the browser expected.

Conversely, the `PUT`, `PATCH`, and `DELETE` methods used to modify data on the server cannot be implemented without contacting the server, so they are not suitable for caching.

The `POST` method can be used to modify data on the server, such as creating a new order in our example Order Management API. It can also be used for read-only operations, usually for complex queries when you want to use a request body instead of putting the operation inputs in the URL or request headers.

A more detailed caching control, irrespective of the HTTP method used, can be achieved with standard HTTP response headers:

- `Cache-Control`: Supports a detailed specification of who (proxies or the end client) can cache the response with the same URL and when (for how long, based on a validation request)
- `ETag`: Short for "entity tag," a value that can be used to check whether a resource has changed or not (in combination with the `If-None-Match` request header in the subsequent request to the same URL)

Old headers (`Expires`, `Last-Modified`, `Pragma`), still supported for backward compatibility, are already superseded by the `Cache-Control` header.

Example — caching product photos

We will demonstrate client-side caching using the standard HTTP headers on the product photo download endpoint created in *Chapter 6*.

Photos are relatively large data objects that are not expected to change very frequently, hence, they are a good candidate for caching.

First, we will use the `Cache-Control` header to specify the time the client can store the photo in the cache. Inside the `ProductsApiController` class, we will change the `downloadProductPhoto` method body to include the header, specifying 20 seconds as the time to cache the image:

```
return ResponseEntity.ok()
    .contentType(
        MediaType.valueOf(photo.getPhotoContentType()))
    .cacheControl(
        CacheControl.maxAge(20, TimeUnit.SECONDS))
    .body(photo.getPhoto());
```

Now, we can start the Product API application and add a test product and its image by calling the PUT endpoints:

```
curl -v -X PUT http://localhost:8080/api/products/AK12345 -d
'{"name":"testprod235","description":"test description","price":123.45}'
-H 'Content-Type: application/json'
curl -v -F "file=@/home/mv/Documents/image.jpeg" -X PUT http://
localhost:8080/api/products/AK12345/photo
```

Please note that the caching header tells the client it can store the resource in its cache, but the client is not required to do so. Simple clients may ignore the caching headers completely.

We will use a web browser because it understands and supports the caching headers; however, it does not load the `main` resource (the one whose URL is entered in the address bar) from the cache. This is why we will create an HTML file named `refer_img.html` that we will use as the `main` resource, and inside the HTML file we will refer to the image we want to download or cache:

```
<img src="http://localhost:8080/api/products/AK12345/photo"/>
```

Now, we can open a new browser tab and turn on the developer tools (by pressing *F12* on Windows/Linux or *Option* + ⌘ + *I* on Mac). Within the developer tools, we switch to the **Network** tab.

We will open the `refer_img.html` file in the browser by dragging and dropping the file in the browser window. When the HTML file is opened for the first time, the browser must make the HTTP request to download the image:

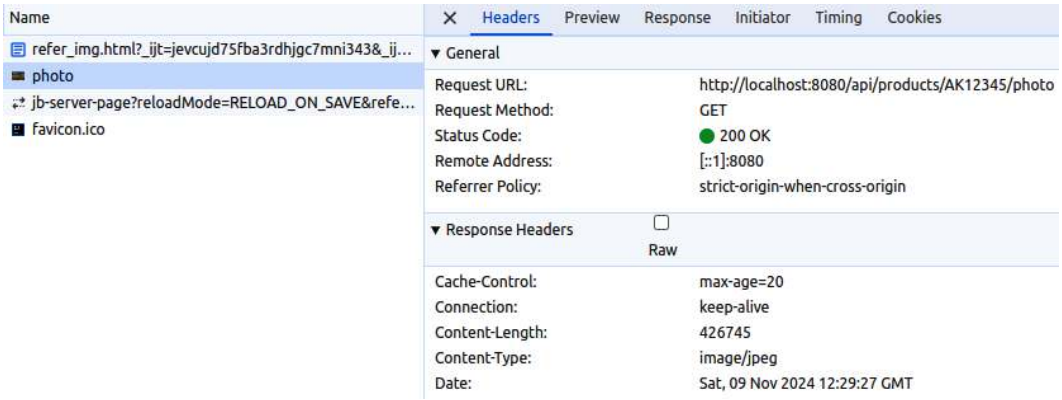


Figure 10.1 – Downloading a new resource that is not available in the cache yet

We can see the response headers, including **Cache-Control: max-age=20**.

If we reload the page within 20 seconds, we should see the following:

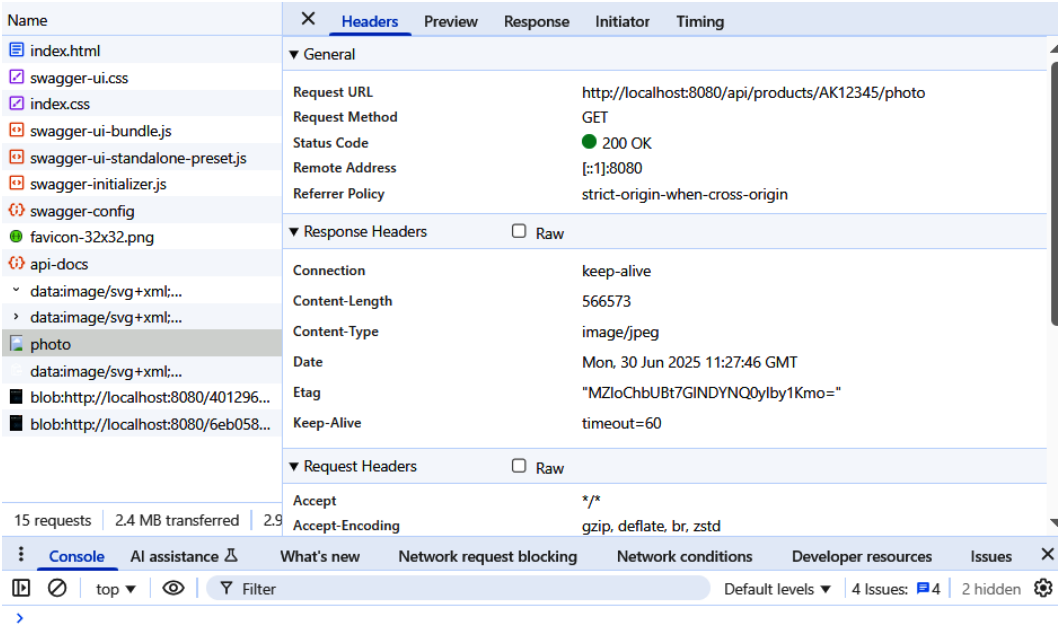


Figure 10.2 – Browser using the cached resource

The text next to **Status Code** and the warning in the **Request Headers** section indicate that this time, no real network transmission has been done, and the cached image is displayed instead. In the **Response Headers** section, we can see that the access time (the **Date** header) is the same as before because the headers are cached as well.

If we reload the page after more than 20 seconds, the image is freshly downloaded from the server, and we will see a new time as the value of the **Date** header.

Let's make the caching more sophisticated with the `ETag` header. To make the value of the header change if and only if the image changes, we will compute the entity tag as a hash of the image bytes. The following method computes the hash using the SHA-1 algorithm and converts it to a printable string using Base64 encoding:

```
private String getHashString(byte[] bytes) {
    try {
        MessageDigest md =
            MessageDigest.getInstance("SHA-1");
```

```
            return Base64.getEncoder().encodeToString(
                md.digest(bytes));
        } catch (NoSuchAlgorithmException e) {
            throw new RuntimeException(e);
        }
    }
```

Now, we can change the controller method to use the `ETag` header:

```
        return ResponseEntity.ok()
            .contentType(MediaType.valueOf(
                photo.getPhotoContentType()))
            .eTag(getHashString(photo.getPhoto()))
            .body(photo.getPhoto());
```

After restarting the application, we can start testing again. The first load of the page will download the image normally:

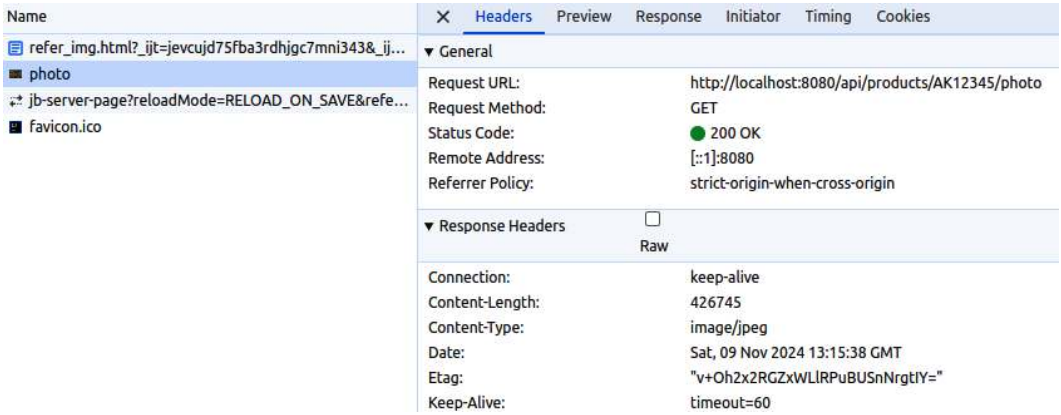


Figure 10.3 – Browser downloading the resource with a new ETag

The **Response Headers** section contains the **Etag** header with the computed hash string.

After that, no matter how long we wait, when reloading the page we will see the following:

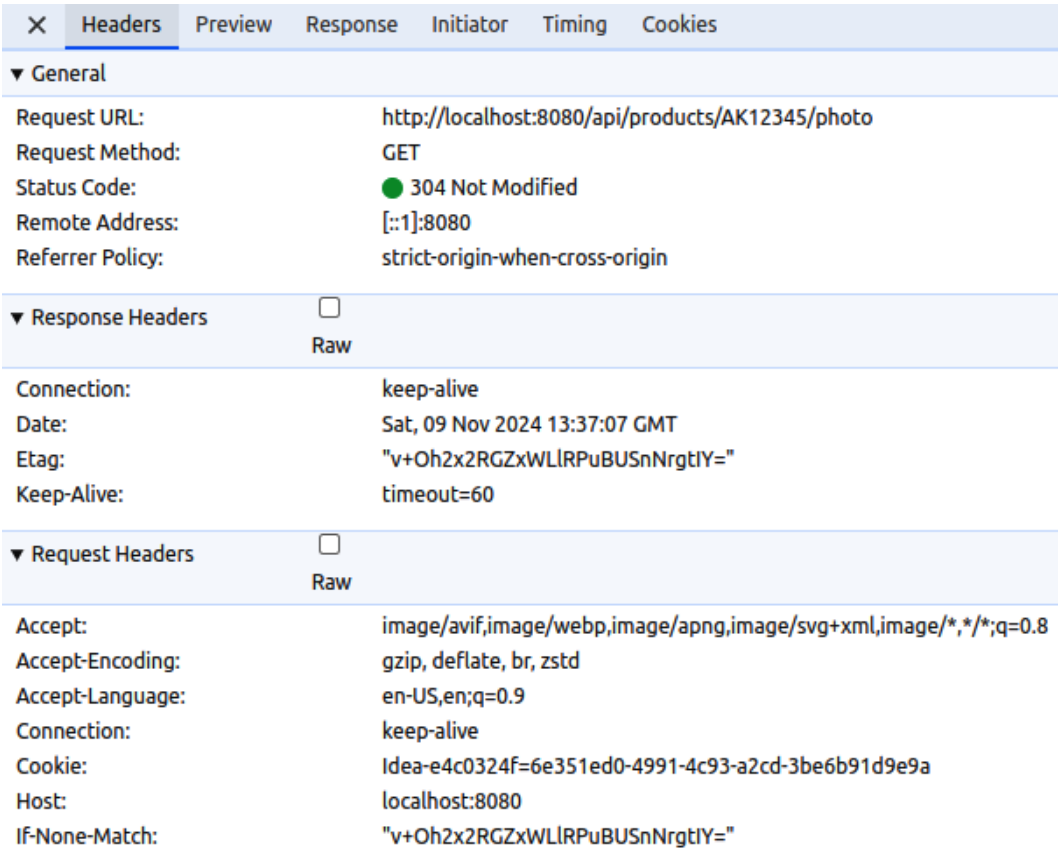


Figure 10.4 – Server responding with "304 Not Modified" when the ETag value is matched

The **Date** header is fresh, meaning the browser did send a real request to the server. However, the **Status code** is **304 – Not Modified**, and no content is sent back.

Figure 10.5 – No content is returned with the "304 Not Modified" response

This is because the browser, when making the request, sent the `If-None-Match` request header containing the `ETag` value it got in the previous request. Spring Framework automatically compares the value with the newly computed one, and if they match, it returns the `304` status in the response.

Let's check what happens when we upload a new image:

```
curl -v -F "file=@/home/mv/Documents/image2.jpeg" -X PUT http://
localhost:8080/api/products/AK12345/photo
```

On the next reload of the page, we get this:

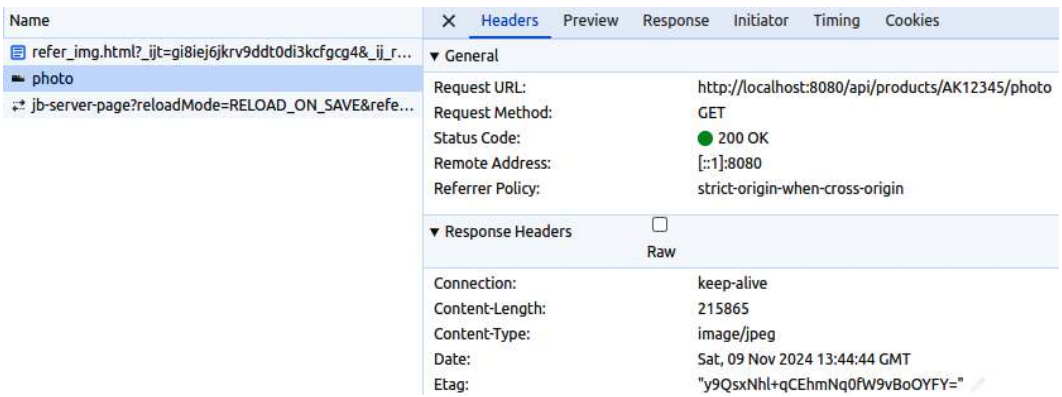


Figure 10.6 – Server responding with "200 OK" and the full content when ETag not matched

As the **Etag** value has changed, the full content is returned, and the status code is **200 OK**.

Command Query Responsibility Segregation (CQRS)

The **Command Query Responsibility Segregation** (**CQRS**) pattern is designed to separate read and write operations, improving performance and scalability in applications. By separating data modification (commands) from data retrieval (queries), each operation can be optimized independently. This separation is particularly useful in applications where read and write operations require different scalability strategies.

For instance, an application with high read traffic can scale read operations without affecting write performance. This prevents slowdowns during read operations when resource-intensive write processes occur. In our example from *Chapter 2*, we applied CQRS principles in a single application, but in practice, CQRS typically involves deploying two separate applications: one handling reads and the other handling writes.

By segregating read and write responsibilities, CQRS improves application performance by eliminating dependency between these operations, enabling independent scaling. To ensure both applications share the same API endpoint, a gateway can be used to redirect traffic to the correct application based on the operation being performed.

Echoing request data is not required

When implementing a POST or PUT endpoint, a commonly used convention is to return the created or updated entity in the HTTP response body. We even use this pattern in the design and code examples in our book. This pattern is usually connected to the reuse of data models: we already have a model for the entity that is used in the request and in the GET response, so why not use it also for the POST and PUT responses?

There is nothing in the HTTP protocol or REST architecture telling us to echo back what we got in a POST or PUT request. On the contrary, there are some arguments against it:

- Waste of network bandwidth and an increase in response time, especially for large entities.
- The client already has the data because it has just sent them in the request.
- It can lead to incorrect assumptions on the client side that the returned data are always up to date, but that might not be true if some other request on the same entity is performed in parallel.

Even if there is some additional information generated on the server, why do we automatically assume that the client needs the modified parts or the whole entity data? It breaks the single responsibility principle: a POST or PUT is expected to do a create/update. If the client needs to read the data, it can send a GET request. Different models for the read and write operations are also a natural consequence of applying the CQRS principle explained earlier in this chapter.

Asynchronous processing

HTTP, the heart of RESTful web services, is built around the synchronous request-response style of communication, making it easy to understand and implement. When talking about asynchronous APIs, we usually think of systems based on message brokers using protocols other than HTTP. Messaging-based communication is one of the main alternatives to RESTful web services mentioned in *Chapter 1*. However, HTTP can also support asynchronous processing.

If the operation to be exposed via a REST API involves processing that takes a long time, waiting for the processing to finish before returning a response blocks the API client makes the API seem unresponsive and may lead to time-outs. The responsiveness of such an API can be improved by returning a response immediately after reading the request and saving the input data from it in reliable storage.

The standard response status code in this case is `202 - Accepted`. The long-running processing can continue asynchronously without blocking the client. A separate API endpoint can be provided for the client to poll the status of the processing and receive its results when finished.

For example, suppose we wanted to improve the photo upload endpoint of our example Product API so that it scales the photo to standard dimensions or performs other graphics enhancements on it. This kind of processing can be time-consuming. In such cases, we can decide to change the upload endpoint to an asynchronous one. We would store the unprocessed photo and return a `202` status code. The photo enhancement would be done asynchronously and a `GET` endpoint would be available for checking whether the photo is ready for use.

After explaining how to ensure performance and scalability using general approaches at the level of analysis, design, and the HTTP standard, in the next section, we will move more to the level of implementation and focus on one feature of current versions of the Java platform that is particularly relevant to the performance and scalability of applications communicating using APIs.

Increasing the throughput with virtual threads

Virtual threads are a Java feature (final since Java 21) related to the performance of concurrent processing, a topic highly relevant to API implementation. All server-side applications are concurrent because they must handle concurrent incoming requests. However, with Java server-side frameworks such as Spring Boot, the code can focus on just one request using the so-called thread-per-request model. More advanced applications may need multiple threads per request. We will explore both usages of concurrency in the following subsections.

Garbage collector for threads

Before going into more exact and detailed descriptions, let's use a simple metaphor to explain the point of virtual threads even to developers who may find threads and concurrency difficult to understand. All Java developers must understand the concept of garbage collection, which is an abstraction that the Java Virtual Machine provides to liberate developers from the responsibility for allocating and freeing memory.

Of course, you can still get an `OutOfMemoryError`, but if your application uses the objects in a typical way, at some point objects that are not used anymore are not referenced from any other object, so the runtime can detect those objects automatically, and thus the memory they occupy can be freed and reused. This abstraction, used by programmers since the very first version of Java, simplifies the code and prevents many hard-to-detect errors related to direct memory access and allocation known from programming in C or other languages without a garbage collector.

Similarly to what a garbage collector does with memory, virtual threads provide the illusion of an (almost) infinite number of threads. What really happens is, in a typical application using one thread per incoming network request (more on that in the next section), the JVM, together with the standard libraries, can automatically detect when the platform thread would be blocked due to waiting (e.g., for I/O). To prevent this, the virtual thread state is moved to the heap, and the platform thread is reused to work on some other request.

For most applications, relying on virtual threads to efficiently utilize platform threads, rather than using platform threads directly, will likely depend on the garbage collector to clean up memory objects at the end of each request.

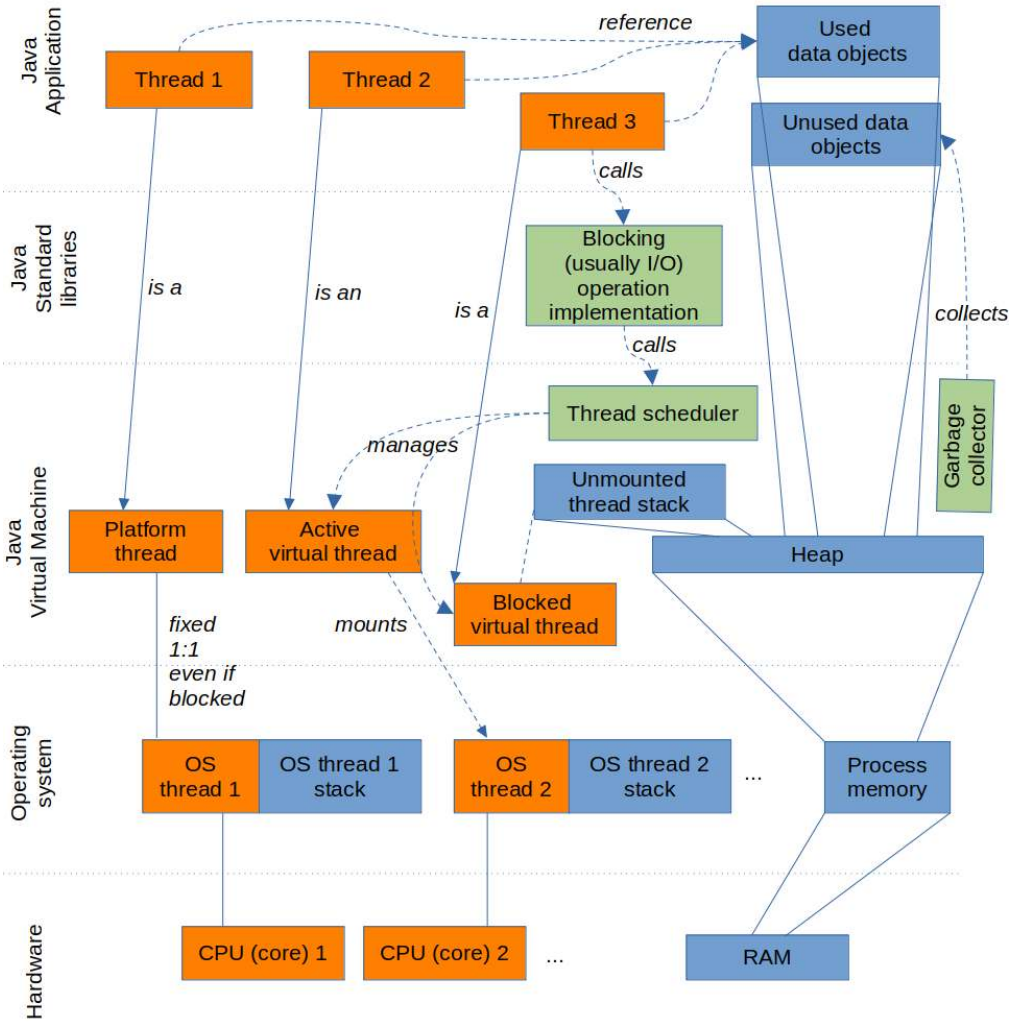


Figure 10.7 – Illustration of how virtual threads work, split into layers

Thread-per-request model

Our example APIs are implemented using Spring Web, a framework using the thread-per-request model. This means that for every HTTP request, the web server used by the framework takes a dedicated Java thread and, within that thread, it invokes the controller method matching the request path and method.

The advantage of the thread-per-request model is that it allows the classic imperative programming style with code blocks containing statements executed in the order they appear in the source code. This code style is easy to read and understand. It is also easy to debug because when the execution stays within one thread, the call stack displayed in debugging tools is complete, letting the developer track the method invocations nicely at all code levels.

Unfortunately, the thread-per-request model without virtual threads has limited scalability because the threads provided by the operating system (so-called platform threads) are an expensive resource. Every platform thread occupies a considerable amount of memory, so the number of platform threads a JVM can use is limited. Web servers using platform threads must take a thread that is needed to handle an incoming request from a limited thread pool. If there are no threads left in the pool, the request waits until a thread is available.

The controller method and any methods invoked from the controller method (let's call it handling code) hold the dedicated thread until the controller method returns or throws, transferring the execution back to the Spring Web framework. It is quite common that the handling code contains blocking operations. A blocking operation prevents the CPU from performing useful work because it halts execution while waiting for required data.

To illustrate this, let's look at the execution of the "create order" operation of the Order Management API.

Let's first look at `OrderManagementApiController.java`:

```
    public ResponseEntity<OrderResponseDto> ordersPost(
        OrderRequestBodyDto orderRequestBody) {
            // ^ involves deserialization from JSON (CPU usage)
            final var order = ordersCommandUseCase.createOrder(
                orderMapper.toOrderRequest(orderRequestBody));
            return ResponseEntity.status(HttpStatus.CREATED)
            .body(orderMapper.toOrderResponse(order));
            // ^ involves serialization to JSON (CPU usage)
        }
```

Next, let's see `OrdersCommandUseCaseImpl.java`:

```
    public Order createOrder(OrderRequest orderRequest) {
        updateProductPrice(orderRequest.products());
            // ^ involves I/O
        Order order = orderMapper.toOrder(orderRequest);
        return ordersRepository.save(// involves I/O
            OrderEntity.fromOrder(order, null));
        }
    private void updateProductPrice(List<Product> products) {
        products.forEach(orderProduct -> {
            ProductDetails cataloguProduct =
            productsQueryUseCase.getProductById(
                // ^ involves I/O
                orderProduct.getProductSKU());
            orderProduct.setPrice(
                cataloguProduct.price());
        });
    }
```

Blocking operations (commonly caused by the need for network I/O communication) can have a disastrous performance impact because, in many applications, they take orders of magnitude longer than the surrounding non-blocking (CPU-heavy) operations. Communication is an essential part of the processing within distributed applications (microservices). It is important to understand that the time spent waiting for data to be transferred over a remote API (or from a complex database query) is typically hundreds of milliseconds to seconds. In contrast, the CPU processing time needed to perform the business logic of most enterprise applications once all the data is available (even if we consider more complex operations such as JSON parsing and serialization) is orders of magnitude shorter – it is counted in microseconds.

This means that with a thread-per-request model, the CPU that is blocked by the input/output operations cannot be used to serve other requests; in a service under heavy load, more than 99% of the time, the CPU will be idle.

One way to address this problem is to use a reactive programming framework (more on that in *Chapter 11*), but that means abandoning the thread-per-request model with its advantages described previously: code that is easy to understand and debug.

Virtual threads are clever because they let developers keep the thread-per-request model while allowing efficient use of the platform threads. When a virtual thread needs to use the CPU, it is mounted on a platform thread (also called a *carrier thread*). When a virtual thread hits a blocking operation, its state of execution is stored in the heap memory and its carrier thread is released so that it can be used to execute a different virtual thread. When the blocking operation finishes, the virtual thread can mount any free carrier thread (that can be different from the one it used before getting blocked), restore its state from the heap, and continue its execution.

Virtual threads, in contrast to platform threads, are cheap, so an application can use millions of them. An HTTP server and an application framework can provide a new virtual thread for every incoming request, no matter how many requests are already waiting for an I/O operation.

To use virtual threads to process incoming requests, you just need to configure the framework to turn on the feature. With Spring Boot 3.2 or later and Java 21 or later, you can use the following configuration property:

```
spring.threads.virtual.enabled=true
```

You can continue using the code style containing the blocking operations, knowing that the blocked threads will be virtual ones, and the CPU will be used efficiently.

Besides handling many incoming requests using the thread-per-request model, an application may need to explicitly perform multiple operations in parallel to handle just one incoming request.

Parallel processing within one request

In the context of integration-heavy applications, a common case is optimizing the latency when handling the request requires multiple independent remote APIs to be called.

Imagine a price quote API that calls the APIs of multiple vendors, compares their prices, and responds with the best one.

Java provides a useful abstraction in the form of the `ExecutorService` class. Each operation that should be executed in parallel is a task that you submit to `ExecutorService`.

When the `ExecutorService` class is created, you specify a strategy it should use to manage the threads that execute the tasks. Without virtual threads, the typical strategy would be to use a thread pool of a fixed size. The thread pool allows a certain number of parallel tasks, but the number is limited because the number of platform threads is limited.

With virtual threads, we can let `ExecutorService` use a new virtual thread for every task without a thread pool and without having to specify the size of that pool:

```
try (ExecutorService es = Executors.newVirtualThreadPerTaskExecutor()) {
    es.submit(()-> { /* task 1 */ });
    es.submit(()-> { /* task 2 */ });
    // ...
};
```

We have briefly explained that, in a nutshell, virtual threads enhance CPU efficiency by minimizing idle threads and ensuring more time is spent performing useful operations. The immediate effect of that is an increase in throughput (the amount of work done per unit of time). However, throughput and response time are usually connected. The increased throughput means that a given CPU power can handle a higher load without some of the requests having to wait for a thread to become available. Consequently, the response time can be improved as well.

So far, we have discussed ways to improve performance and scalability that can be applied mostly within individual services. However, the infrastructure components that reside between the individual components connected by APIs are usually also involved in achieving satisfactory performance and scalability of the entire distributed system (the application). The following section will briefly introduce you to these infrastructure components.

Virtual thread pinning

There are situations when a virtual thread performing a blocking operation cannot release the platform thread for other virtual threads. Using the official terminology for this situation, we say that the virtual thread cannot unmount from its carrier because the virtual thread is pinned to the carrier thread.

As of Java 21, thread pinning can occur in two cases.

The first case of thread pinning is when the virtual thread is performing a native code. This is inevitable because the JVM cannot take away the platform thread from the non-Java code safely.

The second case is when the virtual thread runs code inside a synchronized block or method. This is a limitation of the implementation of the support for virtual threads within JVM. If your application code or the code of some of its dependencies (web server or libraries) uses Java synchronized constructs, it may limit the performance of virtual threads compared to the expectation. In some cases, it can even lead to deadlocks where all the available platform threads are blocked by pinned threads waiting on a synchronized lock.

This means you should not turn on virtual threads blindly. It is recommended to run performance tests on your application with and without virtual threads. You should be aware of your application dependencies and try to use the latest versions of them. Remember that Spring Boot uses the embedded Tomcat as the web server (servlet container) by default, but it also supports switching to other web servers.

The pinning of virtual threads caused by the synchronized constructs is eliminated in Java 24. So, even if the Java version you use supports all the features your application needs, it is always a good idea to upgrade to a new Java version as soon as possible to get fixes and improvements to the implementation of the Java features you use. New versions of Java/OpenJDK also come with improvements in tooling that can help to diagnose and fix various performance problems, possibly including virtual thread pinning.

Using infrastructure support

Ideally, you want to design and implement your APIs so they can work well without depending on a specific infrastructure setup. However, software developers should be aware that components external to the services they develop can be used to enhance the performance and scalability of the system.

There are several types of infrastructure components:

- **Load balancer**: A component distributes the requests among multiple service instances providing the same API. A load balancer is inevitable for horizontal scaling.
- **Proxy**: A component that is an HTTP server and client at the same time. It receives a request and fulfills it by making another request to the actual service providing the API. Proxies let us add various functionalities between the client and the server, of which caching is the most important one for performance:
    - A proxy that receives requests with public URLs and turns them into requests within a private network is called a **reverse proxy**.
    - Proxies combining different functionalities supporting APIs are also called **API gateways**.
- **Content delivery network (CDN)**: This is a network of proxies that are geographically distributed so that a client can access a URL via a nearby proxy, shortening the response times.

The components mentioned here often combine performance and security concerns. One problem where the overlap is obvious is the deflecting of **denial of service** (**DoS**) and **distributed denial of service** (**DDoS**) attacks. It involves throttling, which is limiting excessive traffic from one source to ensure that the service remains available to all its clients.

It follows from this that the architecture in which an API-providing service is embedded may contain multiple components that are often beyond the control of the API developers. This is another reason the best possible approach to API design is to adhere to the standards defined by HTTP and other internet standards (RFCs). This will ensure that your API works well with the surrounding infrastructure.

We have gone through several possible performance and scalability issues, along with ways to address them. However, in the last section of this chapter, it is essential to show how to examine the performance of a particular service under a particular workload, because only in this way can the real problems be detected and targeted for resolution.

Designing and executing effective load tests

Chapter 8 explained various the tests that can be used to ensure the correctness of APIs, meaning that these tests check that the APIs and services behind them process the input data as `expected` and respond with the expected output data. They focus on the exact values of the detailed attributes of the input and output data and their semantics to verify the functional requirements of the application. To do so, they try different test data combinations to cover the most common use cases as well as negative scenarios and edge cases.

In the context of performance and scalability, we test the application from a different angle. Instead of categorizing test cases by functional requirements and checking the exact data values, we usually try a smaller set of use cases characterized by the expected load they put on the application.

However, the number of repetitions of identical or very similar requests sent to the application will be significantly higher. The timing aspect is very important, so the requests are made at a frequency that simulates real traffic, including load fluctuations. Multiple simultaneous requests are sent to simulate many users using the application at the same time.

These differences in load testing in comparison to the other types of test imply that although it is possible to use the same tools for both, doing load testing using tools that specialize in it is easier, more resource-efficient, and provides results that are more accurate and presented in an appropriate form.

Example — load-testing the Order Management API

We will load-test our example Order Management API using the tool called Gatling (its documentation is available at `https://docs.gatling.io/`). Gatling integrates with Java very well, so we can include the load test in our Java source code and run it using a Maven plugin without having to install a separate program.

First, we will add the Gatling dependencies (with the `test` scope) to the `pom.xml` file:

```
        <dependency>
            <groupId>io.gatling</groupId>
            <artifactId>gatling-app</artifactId>
            <version>3.7.2</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>io.gatling.highcharts</groupId>
            <artifactId>gatling-charts-highcharts</artifactId>
            <version>3.7.2</version>
            <scope>test</scope>
        </dependency>
```

Then, we can start writing the test class, also known as a Gatling simulation. Our simulation will try to create many new orders via the Order Management API:

```
public class CreateOrderSimulation extends Simulation {
```

The top level of the test is in the class constructor:

```
  public CreateOrderSimulation() {
      setUp(POST_SCENARIO_BUILDER.injectOpen(
          rampUsersPerSec(10).to(300)
          .during(Duration.ofSeconds(10)),
          constantUsersPerSec(300)
          .during(Duration.ofSeconds(80)))
      .protocols(HTTP_PROTOCOL_BUILDER))
      .assertions(
          global().responseTime().max().lte(5000),
          global().successfulRequests().percent().gt(90d));
  }
```

The code refers to the `HTTP_PROTOCOL_BUILDER` and `POST_SCENARIO_BUILDER` constants, which describe the requests that should be sent to the tested API. The `injectOpen` method specifies the timing and parallelization of the requests:

- During the first 10 seconds of the simulation, the rate of users (requests) grows from 10 per second to 300 per second.
- During the following 80 seconds, the rate of requests is constant at 300 per second.

At the end, we can see the test assertions that expect the maximum response time of 5,000 milliseconds and that more than 90% of the requests should be successful.

To make the performance differences more visible, we will modify the code of the Product API (the API the Order Management API depends on), adding an artificial delay of 1 second in the `getProductById` method of the `ProductsQueryUseCaseImpl` class:

```
        try {
            TimeUnit.SECONDS.sleep(1);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
```

Now, we can start the Product API application. Our test orders contain the product ID AK21101, so we need to create it using the Product API. We can use the Swagger UI, the `curl` command, or, if we use IntelliJ IDEA, we can use a simple text file with the `.http` extension to send an HTTP request:

```
PUT http://localhost:8080/api/products/AK21101
Content-Type: application/json

{
    "name": "Keyboard",
    "description": "Ergonomic Keyboard",
    "price": 60
}
```

Next, we start the Product Management API with these two configuration properties disabled by commenting them out using the # character:

```
    hikari:
#       maximum-pool-size: 500
threads:
    virtual:
#       enabled: true
```

The load test is started by running the following Maven command:

```
mvn gatling:test
```

As the test is running, we can observe that after the load reaches a certain point, many requests fail (the count of `KO`, meaning `not OK`, is growing).

The whole test fails because neither the response time nor the percentage of successful requests is fulfilled.

The detailed graphical visualization of the test can be found in the `target/gatling` directory:

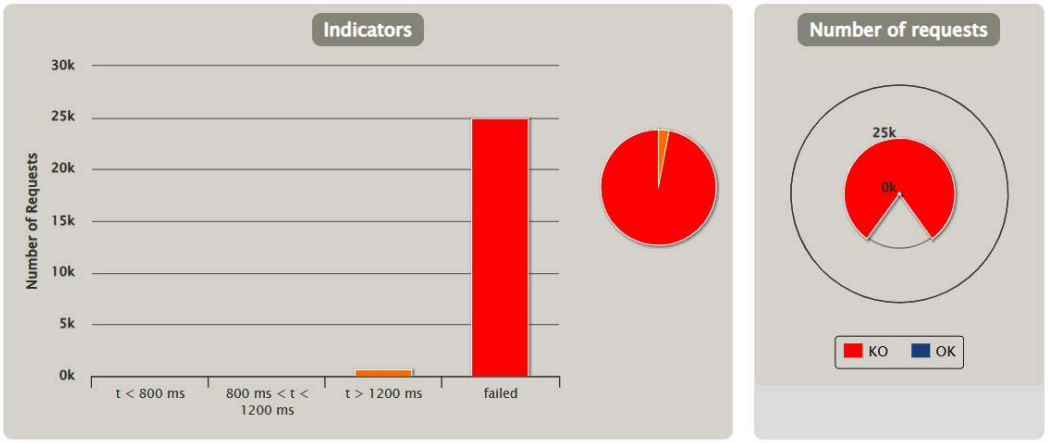


Figure 10.8 – Most of the requests failed as the service was not able to cope with the load

Now, let's try to improve the performance by enabling virtual threads in the Order Management API:

```
threads:
    virtual:
        enabled: true
```

After restarting the application and the Gatling test, we get different results.

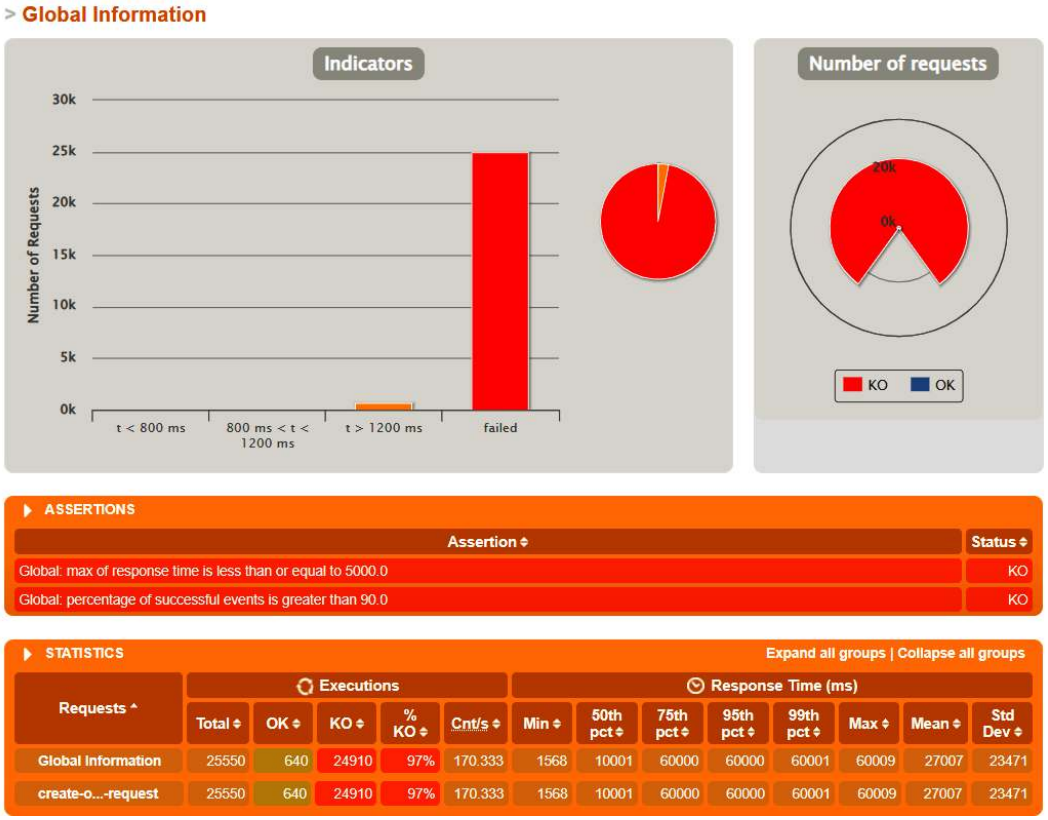


Figure 10.9 – Enabling virtual threads revealed a bottleneck in the database connection pool

The number of successful responses increased slightly, but still, most of them failed.

Looking at the console log output of the Order Management API application, we find many repetitions of the following error:

```
java.sql.SQLTransientConnectionException: HikariPool-1 - Connection is not
available, request timed out after 30000ms.
```

By enabling virtual threads, we removed the limiting caused by a thread pool on the number of requests our application was able to process in parallel. However, another part of the application has now become the bottleneck, namely the database connection pool.

The default limit for the Hikari connection pool is 10 database connections, and that is exhausted very quickly given the 1-second delay of the Product API. The Order Management API must wait until it can finish the database transaction and release the connection. To get rid of the bottleneck, we set the maximum number of connections to `500`:

```
hikari:
    maximum-pool-size: 500
```

Note that this is just a simple example to make the impact of virtual threads very visible. In a real application, such a high number of connections could cause a problem with the database, and we would probably have to use a different solution, such as starting the transaction only after we get the price from the Product API or fixing the long response time of the Product API.

After this change, we can restart the Order Management API and the Gatling test again.

Now, finally, our test passes, and the results are very satisfactory:

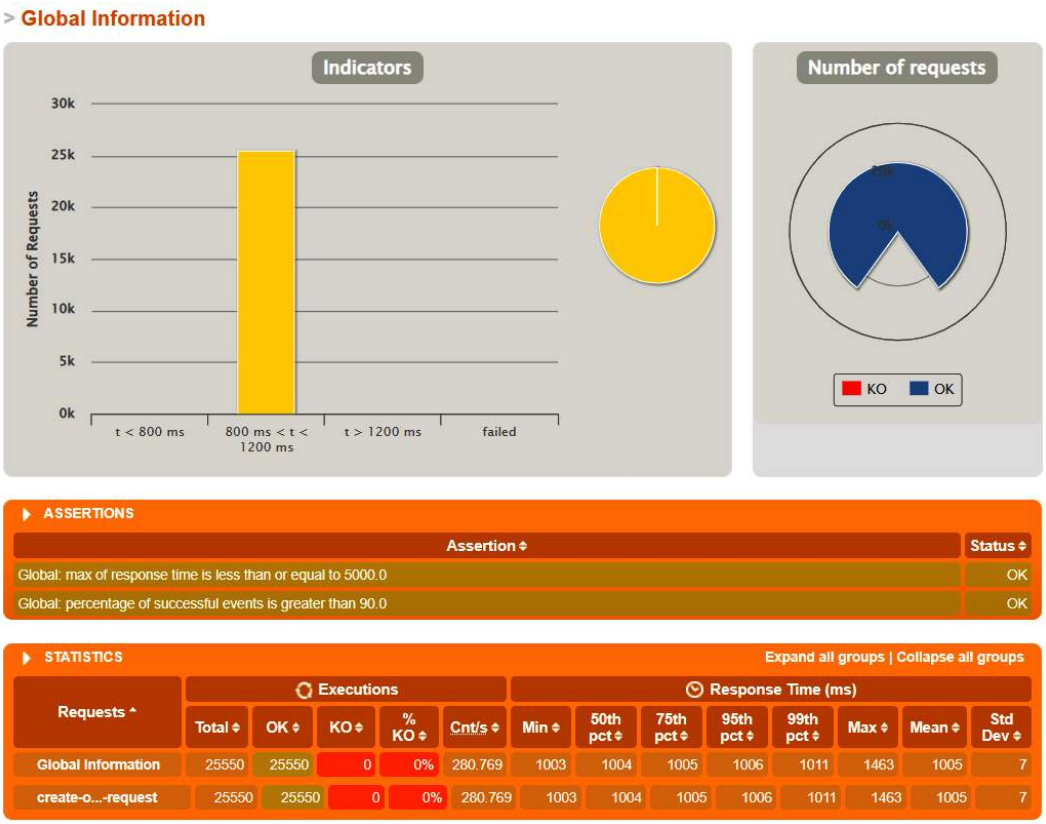


Figure 10.10 – Service using virtual threads can handle the high frequency of requests

All requests were OK, and all the response times were very close to 1 second from the Product API.

Summary

In this chapter, we learned the specifics of performance and scalability in the context of RESTful APIs. We explored various design approaches and techniques to improve the two non-functional requirements. We saw an example of using caching headers to reduce the volume of network traffic between the client and the server. We delved deeper into virtual threads, a highly performance-relevant feature of new Java versions. Finally, we verified the achievement of the expected performance improvements using a load test. In the next chapter, we will show how the principles described in this book can be applied to back-end Java frameworks beyond Spring Boot, and how you can make your application more future-proof and vendor-neutral by using community-driven standards.

11

Alternative Java Frameworks to Build RESTful APIs

Until this chapter, we have used the Spring Boot framework to demonstrate the API implementation in real code. By using the very popular Spring Boot, we are trying to the book as readable and useful for as many readers as possible.

Now it is time to explain why and how other Java frameworks can be used to implement RESTful web services.

You will see that the same design principles and a similar code structure are still applicable, so you can use the concepts from the other chapters with any framework you may want to use for your specific situation.

First, we will explain how standards such as Jakarta EE and MicroProfile can help you not get lost in the sea of frameworks and implementation stacks you could use to build your application. We will briefly mention the possibility of using reactive programming, available with most current frameworks.

The current choice of implementation technologies is the result of the history of innovative approaches to aid developers in creating applications that fulfill the current architectural and deployment preferences. We will walk you from Java EE through Spring to frameworks such as Quarkus and the MicroProfile specification.

To provide you with a detailed view, we will demonstrate the similarities and differences in our example API implementation with the Quarkus and Helidon frameworks compared to the Spring Boot implementation from the previous chapters.

This makes for the following list of sections:

- Understanding the benefits of standards
- Choosing imperative or reactive
- Java EE and Jakarta EE
- Spring Framework and Spring Boot
- New microservice frameworks and MicroProfile
- Example implementation with Quarkus
- Example implementation with Helidon

Technical requirements

In this chapter, we will demonstrate the use of frameworks other than Spring Boot (used in previous chapters). However, the necessary dependencies will be downloaded during the project build automatically based on the Maven `pom.xml` file, so the prerequisites stay the same:

- Intermediate knowledge of the Java language and platform
- At least basic knowledge of Spring Boot or a similar framework
- Java 21 and Maven 3.9.x installed (alternatively, you can use the `mvnw` command included in the example code, which will download the correct Maven version)

You can access the code for this chapter on GitHub at `https://github.com/PacktPublishing/Mastering-RESTful-Web-Services-with-Java/tree/main/chapter11`.

Understanding the benefits of standards

Java is a versatile language with a large ecosystem of both open source and commercial frameworks and libraries. Hence, you have more options to choose from than you will ever be able to fully evaluate.

You could even try to implement your own HTTP server or client and a custom framework, although it is not recommended other than for educational purposes due to the extreme costs of catching the details needed to make it production-ready and to maintain it.

Choosing a framework or any component your software will depend on is difficult. It involves predicting both the future features your software will need and the future evolution and support of the framework (dependency).

For example, an application based on the Java Enterprise Edition standard needs to be deployed to an application server. Later, a need to deploy new application versions frequently, cloud deployment, and scalability requirements may require a faster startup time, smaller container size, and removing the dependency on the application server. Or, the version of the application server the operations team has experience with may become outdated and it will limit the version of Java that can be used.

A common approach is to choose between the frameworks with large communities of users and developers. But we also want new innovative frameworks to enter the market, although their communities will naturally start small.

Fortunately, vendor-neutral standards and specifications have been created to prevent the Java community from becoming fragmented by mutually incompatible frameworks. By choosing a framework that conforms to a standard, you retain the option to switch to a different implementation of the same standard. It will be much easier for developers to start working on your code even without experience specific to the framework provided they are familiar with the standard.

Therefore, in the following sections, we focus not only on the individual frameworks but also on the standards (specifications) they support.

Choosing imperative or reactive

In the previous chapters, we used the imperative programming style using the Spring Web framework.

The **imperative programming style** models an API operation using a simple Java method that takes data from the HTTP request as arguments and returns a result object that will be used to generate the HTTP response. The method body consists of statements that are executed sequentially in the order they appear in the code using one Java thread.

In the following code snippet, we can see the statements `statement1`, `statement2`, and so on, which will be executed one after another in one thread, with access to the same scope of variables and the same call stack.

```
@GetMapping // or other similar annotation
ResultType operationName(InputType1 input1,
                         InputType2 input2...) {
    statement1;
    statement2;
```

```
    ...
    return ...;
}
```

A similar code snippet shows the same structure when the result contains multiple entities. This time, the result type is a well-known Java collection (`List`).

```
@GetMapping // or other similar annotation
List<ResultType> collectionReturningOperationName(
                 InputType1 input1, InputType2 input2...) {
    statement1;
    statement2;
    ...
    return ...;
}
```

The style is called *imperative* because the program is understood as a sequence of commands, telling the computer explicitly what steps to perform in which order. In this model, data is just parameters for the commands.

Besides the "classic" imperative style, with most modern frameworks you also have the option of using the reactive style. For example, with the Spring framework family, you can use *Spring WebFlux*.

The **reactive programming style** is built around the idea of asynchronous processing of streams of data. The program execution is understood as driven by the flow of data. The program reacts to the incoming data items using a pipeline of functions (data transformations).

The processing steps of a single request using the reactive style are executed asynchronously and they usually jump between different Java threads.

Several libraries were created to support reactive programming in Java, among them *RxJava* and *Project Reactor*. All of them share common features based on the Observable design pattern. Their interfaces were standardized in the Reactive Streams specification (`https://www.reactive-streams.org/`). Starting from Java version 9, the interfaces from the Reactive Streams specification have been part of the standard Java API as interfaces embedded in the `java.util.concurrent.Flow` class.

Now let us compare the preceding example to similar code using Spring WebFlux:

```
@GetMapping // or other similar annotation
Mono<ResultType> operationName(InputType1 input1, InputType2 input2...) {
    return reactiveDataSource()
        .map (item -> transformation 1)
        .map (transformation 2...);
}

@GetMapping // or other similar annotation
Flux<ResultType> collectionReturningOperationName(InputType1 input1,
                                                  InputType2 input2...) {
    return reactiveDataSource()
        .map (item -> transformation 1)
        .map (transformation 2...);
}
```

With the reactive style, the handler method returns a special reactive type. For WebFlux (using the Project Reactor library as the reactive streams implementation), the result type will be `Mono` (for at most one data item) or `Flux` (a reactive analogy of a collection that can contain any number of items).

The reactive handler method is invoked and returns before even the first request starts being processed. It is only used to declaratively build the pipeline of asynchronous transformations that should be performed for each request. The actual request processing is then controlled by the reactive framework, depending on the availability of data to process.

By breaking up the processing into separate asynchronous steps (transformation functions), the reactive programming style lets the framework use the computing resources just in time when the data is available.

For the reactive framework to be efficient, the pipeline steps (that would be the functions that are the arguments of the map functions in the preceding examples) are expected to be non-blocking: They should not contain statements that synchronously wait for an input/output to finish. Using the reactive programming style on the incoming API request layer (the `Controller` class in WebFlux) does not make for a correct reactive implementation. Instead, a reactive program should be reactive across all its layers, including database access and downstream API calls.

The reactive programming paradigm enables the development of extremely scalable services and systems. Among other features, reactive streams enhance the communication between the code parts that implement individual processing steps:

- Failures are expected and can also be processed reactively; reactive streams have a separate error channel for that. This is different from ordinary Java streams where failures (Java exceptions) break the whole pipeline, not allowing the processing to continue without explicit catch clauses.
- Data consumers can control the rate of data sent to them by the producers using so-called back pressure.

On the other hand, with reactive programming, we pay the tax of having to fragment the implementation code into small non-blocking functions capable of running on different threads. Reactive programming increases complexity due to its asynchronous and multi-threaded nature, as well as by requiring the discipline to avoid blocking input/output. It puts an increased cognitive load on developers who need to be able to translate between the pipeline declarations and the runtime execution. Request processing spanning multiple threads makes debugging more difficult since the steps leading to a particular execution point are not aligned with the stack trace of the current thread. Therefore, it is advisable to make sure the application being developed needs the level of scalability that justifies the increased complexity.

If the only problem you need to solve is the efficient use of threads so that CPU cores are not idle waiting for blocking operations, it can be done with the currently available versions of Java without incurring the cost of introducing reactive programming. It was explained in the in the section, *Increasing the throughput with virtual threads*, in *Chapter 10*.

To summarize the criteria for choosing the imperative versus the reactive option of the backend framework, try to answer the following questions. The more your answers converge to "yes," the more likely it makes sense to use the reactive implementation:

- Is scalability near the top of the features you optimize for?
- Is the team ready to accept the increased cognitive load of reactive programming?
- Do you need advanced stream control features such as backpressure?
- Is it *not* possible to use virtual threads?
- Can all the layers (database access, downstream API calls, etc.) of your application be reactive?

In the following sections, we will take a look at various Java server-side implementation frameworks, grouped by the related standards, and a little bit of historical context. Before we move on to the next section, here is a table that is providing a chronological overview of major Java standards and their associated server-side framework implementations.

Standards	**year**	**Implementations**
Java 2 EE	1999	
	2000	
	2001	
	2002	
	2003	
	2004	Spring Framework 1.0
	2005	
	2006	
	2007	
	2008	
JAX-RS in Java EE 6	2009	
	2010	
	2011	
	2012	
	2013	
	2014	Spring Boot 1.0
	2015	
MicroProfile 1.0	2016	
Java EE 8	2017	
	2018	Micronaut 1.0
	2019	Helidon 1.0, Quarkus 1.0
Jakarta EE 9	2020	
	2021	
Jakarta EE 10	2022	
	2023	
MicroProfile 7.0	2024	
Jakarta EE 11	2025	

Table 11.1 – Timeline of standards and frameworks

Java EE and Jakarta EE

The Java language and platform can be used to develop various types of applications; however, the area where Java has become the most successful and popular is server-side (backend) applications.

Java Enterprise Edition (Java EE) is a set of standard APIs that extend the **Java Standard Edition** (Java SE, commonly referred to as just "Java" and implemented by various products that build on the OpenJDK project) to support the development of server-side applications.

From Java 2 Platform, Enterprise Edition to Jakarta EE

Java EE started with the name Java 2 Platform, Enterprise Edition (J2EE), along with the release of version 2 of Java (SE) by Sun Microsystems. The Java trademark and Java EE passed to Oracle, which acquired Sun Microsystems in 2010. The last version of Java EE is Java EE 8, released in 2017.

After Java EE 8, the Java EE code and documentation were donated to the Eclipse Foundation, and the specification, starting from version 9, is named *Jakarta EE* (`https://jakarta.ee/`). The respective Java packages were renamed from `javax.*` to `jakarta.*`.

Types of JEE containers

An implementation of the Jakarta EE specification is called a *Jakarta EE container* (also known as an application server), to which Jakarta EE applications can be deployed in the form of **Web Archives (WAR)** or **Enterprise Application Archives (EAR)**. There are various commercial and open-source implementations.

An application server can be a full Jakarta EE container (for example, WebSphere, OpenLiberty, JBoss, GlassFish, Payara, or TomEE) or it can be a *servlet (web) container* only supporting a subset of the Jakarta EE APIs (for example, Jetty or Apache Tomcat). The *Servlet API* is a standard way for Jakarta EE applications to serve HTTP requests.

From the Servlet API to declarative endpoint handler methods

The **Servlet API** models how an HTTP server sees the communication: The main objects are the HTTP request and response, encapsulating all the details such as the HTTP method, URI path, headers, and request parameters. The handler method must explicitly (imperatively) implement the logic to read the items of the request it is interested in and set them in the response. The payloads are accessed as Java I/O streams. The handler method has to convert between the streams of bytes and structured data objects (model classes).

Another Jakarta EE API specification, **Jakarta RESTful Web Services (JAX-RS)**, builds on the Servlet API to make implementation of RESTful web services easier by providing a declarative programming model where the developer only implements handler methods annotated with their respective resource paths and HTTP methods. The JAX-RS implementation automatically dispatches the HTTP requests to the matching handler methods. It also deserializes the request bodies and serializes the responses. JAX-RS in Jakarta EE is like Spring Web MVC in the Spring ecosystem.

As the handler classes provide REST resources, the convention is to call them "resources," – for example, a class providing methods operating on products could be named `ProductResource`. The resource path is specified with the `@Path` annotation and there are annotations for specifying the HTTP method: `@GET`, `@POST`, `@PUT`, and so on. You can see that a resource class is like a controller class in Spring Web.

Spring Framework and Spring Boot

Early versions of Java EE (most notably **Enterprise Java Beans**) were infamous for being hard to use for developers and requiring a lot of boilerplate code. JAX-RS (mentioned above) and CDI (the dependency injection standard) only came to Java EE with version 6, released in 2009.

That situation motivated the creation of the **Spring Framework** (`https://spring.io/`), with version 1.0 released in 2004. Among other features and modules, it provided support for dependency injection, aspect-oriented programming, and Spring Web MVC, the module enabling web service implementation with controller classes.

Spring Framework with **Spring Web MVC** (commonly referred to as just **Spring Web**) and many other modules, thanks to improvements in developer experience, has become very popular and is the most used framework for developing server-side Java applications currently.

The requirement to deploy applications to a separate software product, the Java EE container, was perceived as an unnecessary additional step by developers. Upgrades of application servers required organizational coordination that caused them to be complex and slow, impeding technical modernization, including the upgrades to new Java (SE) versions.

To prevent CPU and memory usage or specific extensions of one application from influencing other applications, there is a common practice to use a separate application server instance for each application.

Spring Boot, first released in 2014, brought the possibility to develop standalone Spring applications that can be run directly without a separate application server. The application can be packaged as a simple **Java Archive (JAR)** that includes an embedded web server (Tomcat by default). Simplified deployment, together with other features such as easy configuration and sensible defaults, made Spring Boot a good fit for developing microservices.

Although Spring Framework and Spring Boot have become a de facto standard in backend Java development, it is important to understand that Spring Web and many other Spring modules are just a layer on top of Java/Jakarta EE. When using Spring Web, you use not only Spring-specific abstractions but also Java/Jakarta EE features directly: Servlet API, Java Bean Validation API, and so on. By using a particular version of Spring, you automatically depend on a particular version of Java/Jakarta EE as well.

New microservice frameworks and MicroProfile

Jakarta EE and Spring Boot, thanks to their long history, are the most mature and most well-known server-side Java application frameworks, providing reliability and stability to many developers. However, around the years 2018-2019, new frameworks such as *Micronaut*, *Helidon*, and *Quarkus* gradually appeared, which, by not being limited by backward compatibility, could choose innovative approaches to application development and better meet the current challenges of microservices and cloud deployment.

Highly scalable cloud-native applications need to be able to do the following:

- Start new service instances quickly to adapt to higher load (upscaling) or to restart after failures
- Use as little CPU, memory, and networking resources as possible to optimize cloud service costs

Such applications should have the following characteristics:

- A small application executable code size to minimize the time delay and network traffic needed to transfer the application to the cloud nodes where new service instances should run
- A short service startup time (the time the service spends preparing before being able to serve requests)
- A small and stable memory consumption (even if a service needs more memory during startup only, it will need a larger and more expensive cloud computing instance)

The new microservice frameworks address these challenges by bringing the following features:

- Limiting the number and complexity of library dependencies
- Replacing dynamic processing during application startup (application configuration, dependency injection, and aspect-oriented programming) with static code generation during application build
- Avoiding the use of reflection, dynamic class loading, or dynamic (proxy) class creation

The resulting applications are well-suited for ahead-of-time compilation to platform-native code (using **GraalVM** – see `https://www.graalvm.org/`)

On the other hand, the emergence of new frameworks brought the problems of how to avoid the fragmentation of the Java backend developer community and how to overcome the barrier of adoption of the frameworks because of a lack of developers familiar with their programming model.

The Micronaut framework tries to help with the transition from Spring and Spring Boot by providing similar abstractions (controllers), supporting many Spring annotations, and other compatibility features to make the transition from Spring smooth.

All the new frameworks try to some extent to exploit the potential of the experience and the broad community around the Java/Jakarta EE standards. Due to their lightweight nature, they choose to implement only selected parts of the Jakarta EE specification. In contrast, they do implement additional features required by distributed cloud-native applications, such as declarative REST clients, observability, fault tolerance, and so on, that were not standardized within Jakarta EE.

Many of the above-mentioned features were standardized in the **MicroProfile** specification (`https://microprofile.io/`) whose version 1.0 was created in 2016. Shortly after its creation, the MicroProfile project joined the Eclipse Foundation. The latest MicroProfile version available at the time of writing this book is 7.0. It is aligned with version 10.0 of Jakarta EE.

The subset of Jakarta EE that overlaps with MicroProfile is named the **Core Profile**. The parts of the Core Profile that are the most interesting in the context of this book are Jakarta RESTful Web Services, Jakarta JSON Binding, and Jakarta JSON Processing. The Core Profile also contains the lightweight dependency injection framework called Jakarta CDI Lite, Jakarta Interceptors, and obviously Jakarta Annotations as a common dependency.

MicroProfile 7.0 extends the Core Profile with the following modules, which are very relevant for API and microservice implementation:

- **OpenAPI**: Generating OpenAPI specification from code for the code-first approach, we showed its Spring alternative in *Chapter 3*
- **REST Client**: For consuming REST APIs using the code-first approach (we used the specification-first approach with client code generation in *Chapter 4*)
- **Fault tolerance**: Also known as resilience, discussed in *Chapter 6*
- **JWT** authentication: An important part of API security, discussed in *Chapter 7*
- **Telemetry**: Providing observability, we showed this aspect using Spring in *Chapter 9*
- **Health:** Used by orchestrators such as Kubernetes to probe a service instance for its healthand readiness to serve requests, a functionality provided by the Actuator in Spring Boot, we will show this using Spring in *Chapter 12*
- **Config**: Flexible configuration options, a common requirement for cloud-native services

After listing the abstract concepts that the MicroProfile specification covers, in the following sections, we will show small examples of API implementations using two frameworks that fully implement the MicroProfile specification: *Quarkus* and *Helidon*.

Example implementation with Quarkus

Quarkus (`https://quarkus.io/`) is a framework providing optimized startup time and memory usage. Quarkus makes deployment to Kubernetes and the cloud easy in many ways. Despite being relatively new, thanks to its great developer experience, it has been adopted quickly in many enterprise application development projects.

Among other features, Quarkus boasts its *development mode (dev mode)*, which detects changes in Java source files or resource files (e.g., configuration properties), recompiles them automatically, and applies them (if possible) to the running application (performs a so-called hot deployment) without a need for an explicit restart. With the dev mode, the developer gets fast feedback without losing focus on the code due to lengthy application rebuilds and restarts.

We will start our Quarkus example by implementing the Product API, as we did in *Chapter 2* with Spring Boot.

Exposing the Product API

We will use the same endpoint design as in *Chapter 2*:

- `GET /products` to list all products
- `PUT /products/{id}` to create or update a product
- `DELETE /products/{id}` to remove a product
- `PATCH /products/{id}` to update product description only
- `GET /products/{id}` to get the data of one product.

We will use the same Clean Architecture structure, so for exposing the REST API we will focus on the `*.adapter.inbound.rest` package. The main class in the package is the resource class.

The resource class

In accordance with the Jakarta REST terminology, the class handling the REST resource endpoints is named `ProductResource`. The base URL path for the resource endpoints is specified using the `@Path` annotation:

```
@Path("/api/products")
public class ProductResource {
```

Each endpoint is implemented by a handler method. Let us start with the `createOrUpdateProduct` operation:

```
    @PUT
    @Path("{productId}")
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    public Response createOrUpdateProduct(
        @PathParam("productId") @ValidSku String productId,
        @Valid ProductInput productInput) {
```

There are multiple annotations on the method and its arguments:

- `@PUT`: The HTTP method
- `@Path`: The endpoint relative path
- `@Consumes`: The expected request content type
- `@Produces`: The response content type
- `@PathParam`: The argument bound to the URL path parameter

The built-in `@Valid` and custom `@ValidSku` bean validation annotations are used the same way as with Spring Boot.

The `Response` method return type, like `ResponseEntity` in Spring Boot, lets the method body decide on the HTTP response code and additional response headers dynamically.

The method to list all products is quite simple; the only annotation needed is the HTTP method, and the method returns the body of the HTTP response directly:

```
    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<ProductOutput> getProducts() {
```

The method to get one product by its ID is similar, but in addition, it needs the `productId` path parameter:

```
    @GET
    @Produces(MediaType.APPLICATION_JSON)
    @Path("{productId}")
    public ProductOutput getProductById(
        @PathParam("productId") @ValidSku String productId) {
```

Here's the method to update a product description:

```
    @PATCH
    @Path("{productId}")
    public ProductOutput editProduct(
        @PathParam("productId") @ValidSku String productId,
        @Valid ProductDescriptionInput input) {
```

And finally, the method to remove a product:

```
    @DELETE
    @Path("{productId}")
    public Response deleteProduct(
        @PathParam("productId") @ValidSku String productId) {
```

Keeping within the standard

Please note that, in the resource class, we deliberately decided to only use the annotations from packages starting with the `jakarta.` prefix so that our code stays within the MicroProfile specification. We could have used other annotations supported by Quarkus (and recommended in some tutorials) – for example, `org.jboss.resteasy.RestPath` instead of `jakarta.ws.rs.PathParam` – and we could have skipped specifying the parameter name (`"productId"`) as it would be inferred from the method argument name. However, that would make the code Quarkus-specific. For the reasons stated earlier in this chapter, it makes sense to use standardized APIs even if it means sacrificing a little bit of the convenience some specific implementation may offer.

The rest of the package (DTOs, mappers, and the ValidSku annotation) are framework-independent, so the code is the same as for Spring Boot.

To handle exceptions by mapping them to the correct HTTP responses, MicroProfile supports the ExceptionMapper interface, described in the following section.

Exception mappers

For our example API, we want to define mappings for two exception types:

- `EntityNotFoundException`, meaning a product with the given ID is not found
- `ValidationException`, meaning some input does not comply with the bean validation annotations

Both mapper classes are alike, so it will suffice if we show the first of them:

```
package com.packt.productapi.adapter.exception;

import com.packt.productapi.adapter.inbound.rest.dto.ProblemDetail;
import jakarta.ws.rs.core.Response;
import jakarta.ws.rs.ext.ExceptionMapper;
import jakarta.ws.rs.ext.Provider;

@Provider
public class NotFoundExceptionMapper implements
ExceptionMapper<EntityNotFoundException> {
```

```
    @Override
    public Response toResponse(EntityNotFoundException
                               entityNotFoundException) {
        return Response.status(Response.Status.NOT_FOUND)
                .entity(new ProblemDetail(
                        Response.Status.NOT_FOUND.getReasonPhrase(),
                        Response.Status.NOT_FOUND.getStatusCode(),
                        entityNotFoundException.getMessage()))
                .build();
    }
}
```

It can be seen that the `ProblemDetail` DTO is defined in our code as it is not provided by Jakarta REST.

In the following section, we will briefly mention the remaining parts of our application and test the REST API.

Completing and testing the Quarkus application

Classes in the `domain` and `usecase` packages, decoupled from the communication layers using Clean Architecture, need not change when we switch from Spring Boot to Quarkus.

To create a working Quarkus application, we need to implement persistence. We use Jakarta Persistence to map the database tables to entity classes. However, to be able to access the database using the Repository pattern (as we did with Spring Data), we have to use Panache, which is a module specific to Quarkus.

The repository pattern for persistence is not standardized within MicroProfile; however, there isthe new Jakarta Data specification in Jakarta EE 11.

To link the components of our application together, we use dependency injection with Jakarta Contexts and Dependency Injection (CDI) features (`@ApplicationScoped` and `@Inject` annotations) that are supported by MicroProfile.

As persistence and dependency injection are concepts that are not specific to RESTful web service implementation, we are not going to explain in more detail how to use them with Quarkus, MicroProfile, or Jakarta EE.

When we have the application code complete, we can start the application using the convenient Quarkus dev mode:

```
./mvnw compile quarkus:dev
```

Then we can try the API endpoints using the `curl` commands, creating the product with the `PUT` method:

```
curl -v -X PUT http://localhost:8080/api/products/AK12345 -d
'{"name":"testprod235","description":"test description","price":123.45}'
-H 'Content-Type: application/json'
```

We list all products:

```
curl -v http://localhost:8080/api/products/
```

In the next chapter, we will get the documentation of the API we have just created in the OpenAPI standard.

Getting the API specification from the code

We saw how the implementation-independent specification of an API can be generated for a Spring Boot application in *Chapter 3*.

A similar code-first approach is available for MicroProfile applications, thanks to the MicroProfile OpenAPI specification.

We will add just one dependency in the `pom.xml` file of our Quarkus application:

```
<dependency>
<groupId>io.quarkus</groupId>
<artifactId>quarkus-smallrye-openapi</artifactId>
</dependency>
```

Now we can enter the Swagger UI by opening the `http://localhost:8080/q/swagger-ui/` URL in a web browser.

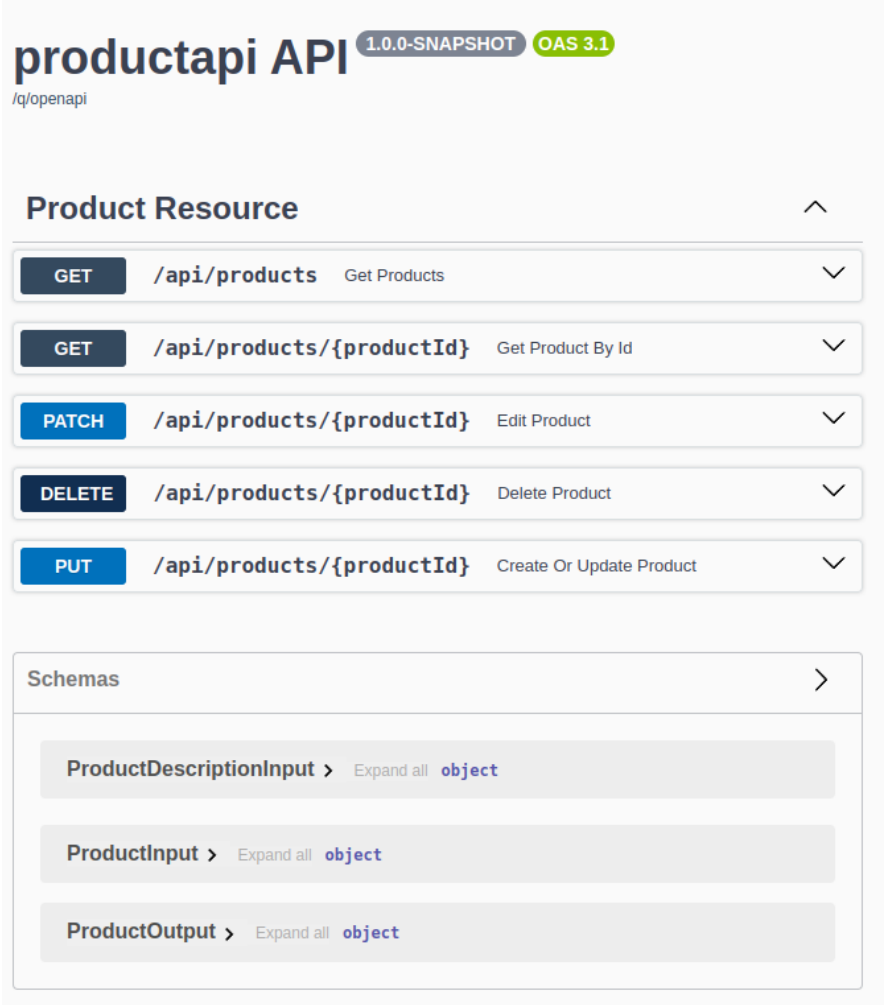


Figure 11.1 – Swagger UI generated from a Quarkus application

The OpenAPI specification (in YAML format) can be downloaded using the `http://localhost:8080/q/openapi` URL.

We have shown how to create and document the Product API with Quarkus using the code-first approach. In the next chapter, you will see that, even with Quarkus, we can create APIs specification-first, using Java code generation from OpenAPI.

Generating MicroProfile code from OpenAPI

We are going to create a stub (not the full implementation) of the Order Management API that was implemented with Spring Boot in *Chapter 4*.

We'll use the same `Order_Management_API.yml` specification file as in *Chapter 4*.

The `pom.xml` file contains the same Quarkus dependencies as in the *Exposing the Product API* section. Additionally, we add the plugin configurations to generate the code:

```
<plugin>
    <groupId>org.openapitools</groupId>
    <artifactId>openapi-generator-maven-plugin</artifactId>
...
            <configuration>
                <inputSpec>${project.basedir}/
                  src/main/resources/Order_Management_API.yml</inputSpec>
                <generatorName>jaxrs-spec</generatorName>
                <apiPackage>
                    com.packt.ordermanagementapi.adapter.inbound.rest
                </apiPackage>
                <modelPackage>
                    com.packt.ordermanagementapi.adapter.inbound.rest.dto
                </modelPackage>
                <modelNameSuffix>Dto</modelNameSuffix>
                <configOptions>
                    <useJakartaEe>true</useJakartaEe>
                    <interfaceOnly>true</interfaceOnly>
                    <useSwaggerAnnotations>false
...
```

Some configuration parameters are the same as with Spring: package names, model name suffix. The generator name is different; we are using the one compliant with the JAX-RS specification. We use the `useJakartaEe` flag because the current version of the specification requires `jakarta` instead of `javax` as the package prefix.

We use the `interfaceOnly` flag because we want our manually written classes to implement generated interfaces. This way, if our implementation does not match the generated interface, it will be reported by the compiler automatically.

Now we can run the Maven `compile` goal:

```
./mvnw clean compile
```

We can see the generated sources in the `target/generated-sources/openapi` directory.

If we try to run the Quarkus application now, and open the Swagger UI at `http://localhost:8080/q/swagger-ui/`, we will see that no REST endpoints (operations) are exposed:

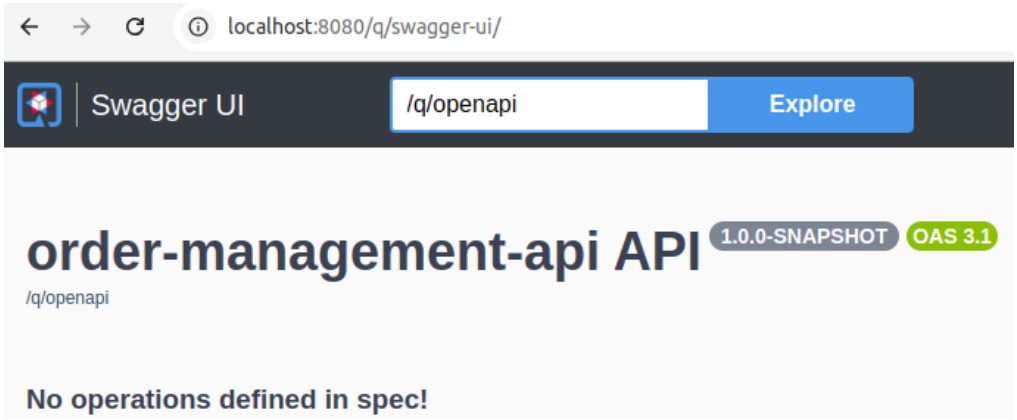


Figure 11.2 – Swagger UI showing no operations

We need to provide an implementation of the generated interface. Before doing so, we need to add one more plugin to make the generated sources accessible from our manually written code:

```
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>build-helper-maven-plugin</artifactId>
    <executions>
        <execution>
            <phase>generate-sources</phase>
            <goals>
                <goal>add-source</goal>
            </goals>
            <configuration>
                <sources>${basedir}/target/
                        generated-sources/openapi/src</sources>
            </configuration>
        </execution>
    </executions>
</plugin>
```

Just to demonstrate the concept, we will create a stub implementation of the generated interface:

```
public class OrderResource implements OrdersApi {
    @Override
    public OrderResponseDto ordersPost(OrderRequestBodyDto
                                       orderRequestBodyDto) {
        return null;
    }

    @Override
    public List<OrderResponseDto> ordersGet() {
        return List.of(new OrderResponseDto().id("1"));
    }
...
```

Opening the Swagger UI again shows that the operations are now available.

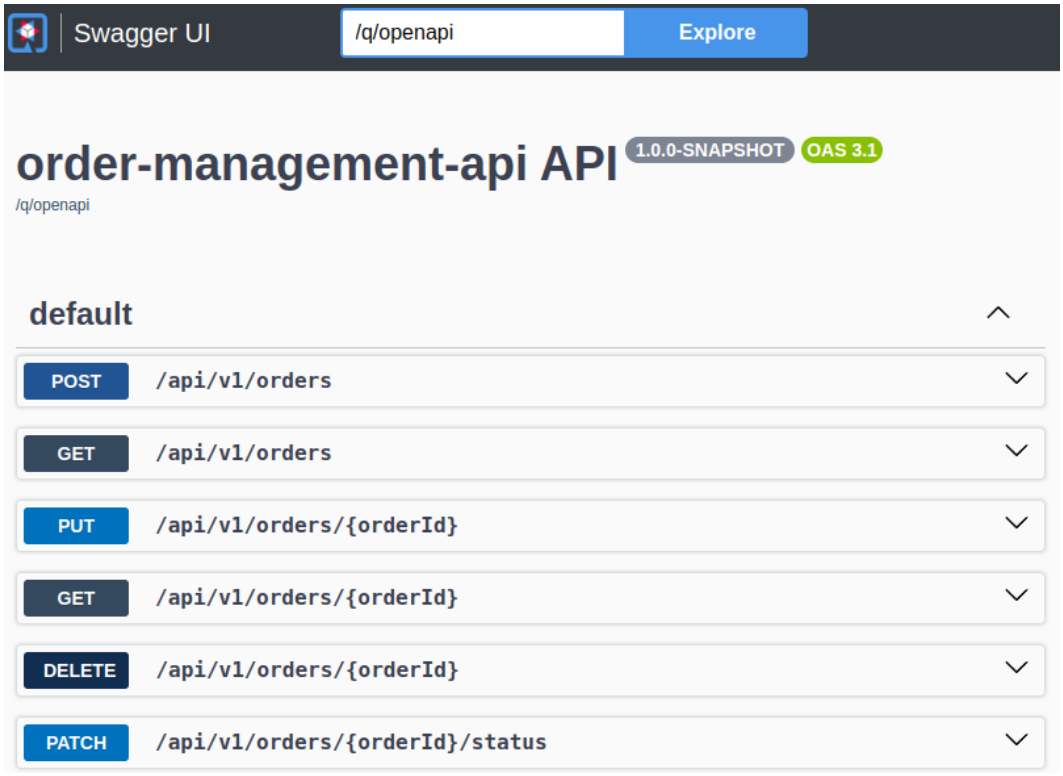


Figure 11.3 – Swagger UI of the Order Management API

We can test it, for example, by sending the POST request with the body. The response will be empty (because our implementation is not complete) but successful (HTTP 2xx status code).

```
{
  "customer":  {
    "customerType": "person",
    "firstName": "John",
    "lastName": "Doe",
    "streetAddress": "1234, Rest Street",
    "city": "Javatown",
    "postalCode": "12345"
  },
  "products": [
    {
      "productSKU": "AK12345",
      "quantity": 1
    }
  ]
}
```

In the last section of the chapter, we will see that similar code to what we used with Quarkus can work with another framework, Helidon.

Example implementation with Helidon

As we have used the MicroProfile annotations only, exactly the same `ProductResource` class code that was used with Quarkus works with Helidon, too.

Instead of the Panache repository-based data persistence implementation used with Quarkus, we use the `EntityManager` interface of the **Jakarta Persistence (JPA)** specification directly.

Obviously, compared to the Quarkus implementation, we had to put different dependencies in the `pom.xml` file, and we used different configuration files, too.

The application is built with this Maven command:

```
./mvnw package
```

Then we can start the application JAR file:

```
java -jar target/product-api-helidon.jar
```

Of course, the API of the running application will behave the same way as the other implementations, so it can be tested with the same `curl` commands.

Thanks to the MicroProfile OpenAPI compliance and Helidon OpenAPI UI module, when these dependencies are in our `pom.xml` file, we can see the documentation of the REST API in a web browser at `http://localhost:8080/openapi/ui`.

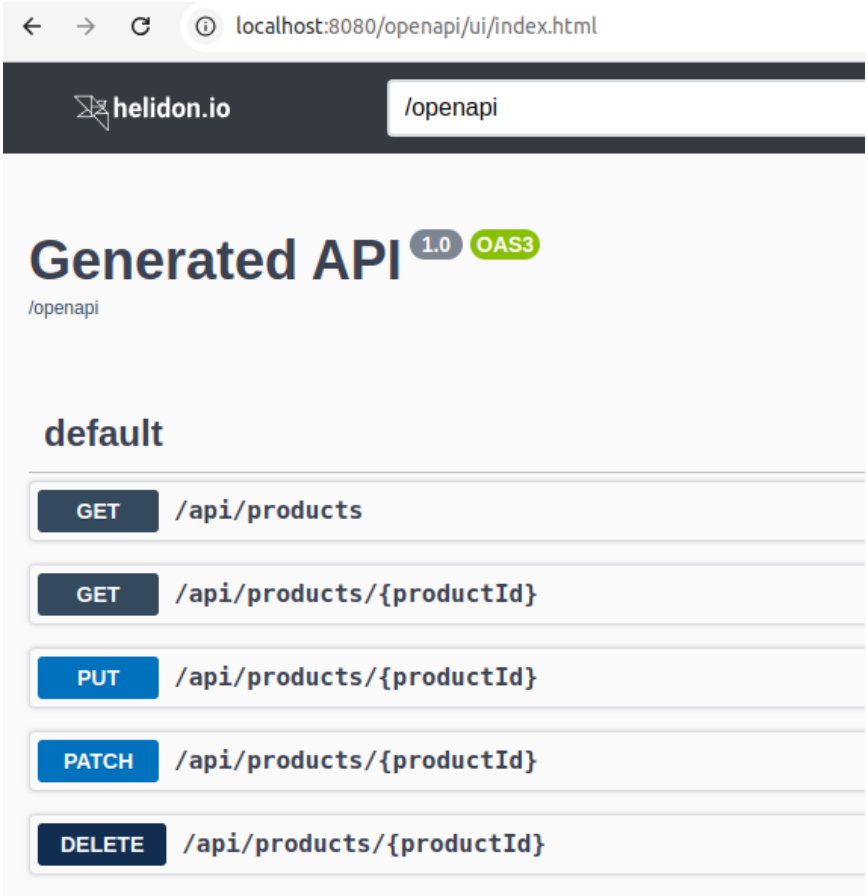


Figure 11.5 – OpenAPI UI generated by Helidon

We can see that, using the MicroProfile standard features, we get the same results when switching the frameworks from Quarkus to Helidon.

Summary

In this chapter, we summarized from a high-level perspective, the different frameworks that can be used to implement Java services exposing REST APIs. Using a historical overview, we explained the different goals and features that motivated the creation of these frameworks. We explained how vendor-neutral standards help prevent developer community fragmentation and lock-in to a particular implementation technology. By implementing parts of our example APIs in two MicroProfile-compliant frameworks (Quarkus and Helidon), we demonstrated that the various REST API design principles we went through in the previous chapters can be applied regardless of the framework used and lead to application code with the same basic structure.

12

Deploying APIs

Throughout our journey of mastering RESTful APIs with Java, we have explored various crucial aspects of development. From writing efficient code to implementing robust design principles and following industry best practices, we have built a solid foundation for creating high-quality APIs. However, the path to delivering production-ready software extends beyond development alone.

While our accumulated knowledge of API development is invaluable, it represents only part of the complete software delivery life cycle. The critical bridge between development and production deployment remains to be crossed. This final stage—the deployment process—transforms our well-crafted API into a production-ready service that delivers real value to end users.

For that, we will cover the following topics:

- Preparing APIs for deployment
- Containerization
- **Platform as a Service (PaaS)**
- Deployment best practices and patterns
- Practical examples throughout

Preparing APIs for deployment

Before deploying your RESTful API to production, you need to complete several important preparation steps. This preparation ensures your API will run properly, securely, and reliably in a production environment.

In this section, we will cover some of the key elements required to make your Java API ready for deployment. First, we will examine configuration management—how to structure your application settings to work across different environments without changing code. Then, we will discuss implementing health checks that monitor your API's status and help maintain system reliability.

These preparation steps form the foundation of a successful deployment process. They help prevent common problems and create a stable base for your API in production. By following these practices, you'll reduce deployment issues and make your API easier to maintain as it grows.

Let's now explore these preparation elements in detail, beginning with effective configuration management.

Configuration management

In modern RESTful API development, proper configuration management is essential for maintaining flexible, secure, and maintainable applications. In this section, we will explore the aspects of configuration management and its benefits.

Externalized configurations

Externalized configurations separate your application's core functionality from its configuration setting, avoiding the necessity of placing hardcoded configurations inside of the code. This approach allows you to modify application settings without needing to change the code. By keeping configuration data outside of the code base, you can easily adjust environment-specific settings—such as database URLs, security keys, or API endpoints—without recompiling or redeploying your application.

For example, using Spring Boot, you can externalize configuration by defining properties in an external `Properties` or YAML file rather than hardcoding them in your Java code. To externalize configuration in a Spring Boot application and bind it to a Java class, follow these steps:

1. Create an external configuration file (`application.yml`):

```
server:
    port: 8080
spring:
    datasource:
        url: jdbc:postgresql://localhost:5432/mydatabase
        username: myuser
        password: mypassword
api:
    timeout: 30
```

2. Use application configurations from `application.yml`:

```
@Configuration
@ConfigurationProperties(prefix = "api")
public class ApiProperties {
  @NotNull(message = "Timeout value is required")
  @Min(value = 1, message = "Timeout must be at least 1 second")
  private Integer timeout;

  // Getter and Setter
  public Integer getTimeout() {
    return timeout;
  }
  public void setTimeout(Integer timeout) {
    this.timeout = timeout;
  }
}
```

In this example, the API timeout setting is externalized in the `application.yml` file. The `ApiProperties` class binds to the `"api"` section, ensuring that any changes to the configuration file—such as adjusting the timeout—will take effect without modifying the application code.

This approach promotes flexibility and agility during deployment, enabling rapid adjustments to configuration values based on the deployment environment.

Benefits of externalized configurations

Separating configuration from your application code represents a foundational best practice in modern software architecture, particularly for RESTful APIs that must operate across diverse environments. By externalizing configuration, development teams gain tremendous flexibility in how applications are deployed, secured, and maintained throughout their life cycle. This approach fundamentally transforms how we manage application settings, moving from hardcoded values embedded in source code to dynamic, environment-specific variables that can be modified without recompilation. The strategic advantages of this approach extend across multiple dimensions of your development and operational processes:

- **Environment independence**: Developers benefit from seamless transitions between different environments without changing code. For example, your Order API can connect to a test product catalog during development that contains sample products, but automatically switch to the production catalog with real inventory when deployed to production.

This allows developers to create and test orders with test data locally without affecting real inventory or sending notifications to actual customers.

- **Simplified maintenance**: Operations teams gain the ability to modify application behavior without requiring developer intervention. For instance, when your Product API needs to increase the product image cache size from 100 MB to 500 MB due to new high-resolution images, operations staff can simply update the externalized configuration value rather than requesting code changes and redeployment. Similarly, the timeout settings for third-party shipping provider integrations in your Order API can be adjusted based on their performance characteristics without development team involvement.
- **Enhanced security**: Security is significantly strengthened by removing sensitive credentials from source code. For example, instead of hardcoding the payment gateway API key like `paymentGateway.setApiKey("sk_live_51MlkA...")` in your Order API code (where it would remain visible in commit history), you can use `paymentGateway.setApiKey(environment.getProperty("payment.api.key"))` and securely store the actual keys in environment variables. This allows different keys to be used for development versus production without exposing sensitive credentials.
- **Easier DevOps integration**: Continuous integration and deployment pipelines become more powerful with externalized configuration. For example, you can deploy the same Order API Docker container to testing and production environments using different settings:

  ```
  docker run -e "SPRING_PROFILES_ACTIVE=production" -e "PRODUCT_API_
  URL=https://product-api.example.com/v1"https://product-api.example.
  com/v1" -e "INVENTORY_CHECK_ENABLED=true" order-api-image.
  ```

  In the testing environment, you might set `INVENTORY_CHECK_ENABLED=false` to skip actual inventory verification, while enabling it in production to prevent orders for out-of-stock products.

As we have seen, externalizing configuration provides critical advantages for building robust, maintainable, and secure RESTful APIs. By decoupling configuration from code, we gain the flexibility to deploy across environments, simplify operational updates, enhance security, and streamline DevOps processes. Let us now take a look at how to handle profile-based configurations that will allow developers to have the flexibility to work and deploy automatically, selecting a specific set of configurations for each environment.

Profile-based configurations

When deploying your RESTful APIs across different environments—from a developer's local machine to test servers and ultimately to production—managing environment-specific settings becomes a critical challenge. Profile-based configuration provides an elegant solution that directly addresses deployment complexities by allowing your API to adapt its behavior based on where it's running, all without changing a single line of code.

For your Order API and Product API, this approach means you can maintain a single code base and deployment artifact while automatically adjusting crucial settings such as database connections, external service endpoints, security parameters, and feature toggles based on the target environment. This capability streamlines your CI/CD pipeline and dramatically reduces configuration-related deployment failures.

Common profiles

Imagine your Order API needs different database connections and behavior settings across environments. You can structure your configuration files as follows:

- **Development**: In the development environment—often referred to by the "dev" profile—developers focus on writing and testing code either locally or in shared development servers. This stage of the software life cycle typically sees very frequent code changes, constant experimentation, and rapid deployments. Because reliability is not yet critical, developers often use lightweight databases such as H2 or other embedded options and may rely on simulated services to stand in for external dependencies. Verbose logging is enabled so that any issues can be quickly diagnosed, and debugging features are turned up to gain deeper insight into the code. Because no actual customer data is involved and real transactions are not processed, there is minimal risk to the business at this stage. A practical example in the Order API might involve running an in-memory database for orders to facilitate speedy local testing, calling a locally running instance of the Product API, and simulating payment and shipping services to move faster without waiting on real integrations.
- **Testing**: When it's time to verify that everything works together seamlessly, the testing environment—aligned with the "test" profile—comes into play. This environment is generally more stable than development, as changes are introduced less often and are usually more refined. It mirrors production more closely by leveraging databases and external service configurations that resemble real-world conditions while still using test data. Throughout this phase, teams carry out integration tests, automated user interface

tests, performance validations, and other checks to ensure the system behaves as expected. Here, the Order API might connect to a dedicated test version of MySQL for managing orders, call a testing instance of the Product API, and possibly generate shipping labels via a test shipping service endpoint. To prevent real charges, payments are typically handled by a simulated gateway. This controlled environment allows QA teams and automated pipelines to validate complete workflows without jeopardizing real customer data or incurring unnecessary costs.

- **Production**: Once the application has passed all the necessary checks, it graduates to the production environment—commonly known by the "prod" profile. In this live environment, your APIs serve genuine end users and handle real-world interactions. Uptime, security, and performance become paramount. Code running here is thoroughly tested and generally changes less frequently, as stability takes priority over rapid iteration. The Order API in production will connect to robust, secure, and often clustered databases designed to handle large transactions at scale, while simultaneously integrating with authentic external services such as real payment providers and shipping carriers. Logs are carefully managed to collect essential information without adversely affecting performance. Given the high stakes and real customer data, asynchronous order fulfillment is frequently employed to ensure reliable processing under heavy loads. This environment demands close monitoring and swift response to any issues to maintain a seamless customer experience.

We can express this structure in Spring Boot. To take advantage of this approach, we need to organize all the variables used in each environment that receives the suffix of the file with its environment name, as follows:

```
src/
├── main/
│   ├── resources/
│   │   ├── application.yml          # Common settings for the Order API
│   │   ├── application-dev.yml      # Local development settings
│   │   ├── application-test.yml     # Automated testing environment settings
│   │   └── application-prod.yml     # Production deployment settings
```

Figure 12.1 – File structure of Spring profile-based configuration

Activating profiles during deployment

When deploying your Order API to different environments, you activate the appropriate profile through environment variables or command-line arguments:

- **Local development**:

```
# Developer's local machine
java -jar order-api.jar --spring.profiles.active=dev
```

- **Automated test environment**:

```
# In your CI/CD pipeline's test stage
export SPRING_PROFILES_ACTIVE=test
java -jar order-api.jar
```

- **Production deployment with Docker**:

```
# In your production deployment script
docker run -e "SPRING_PROFILES_ACTIVE=prod" -p 8080:8080 order-
api:latest
```

This approach ensures that when your Order API is deployed to production, it automatically connects to the production database, uses the live payment gateway, communicates with the production Product API instance, and applies appropriate performance and security settings—all without any code changes or manual configuration steps.

Profile-based configuration directly addresses the challenges of deploying the same API across multiple environments, making your deployment process more reliable, maintainable, and adaptable to changing infrastructure requirements.

Environment variables

Environment variables provide a secure and flexible way to manage configuration values, especially for sensitive data. When deploying your API, you can set sensitive configurations such as database credentials, API keys, and service endpoints as environment variables on your server. This approach keeps critical information out of your code base while making it accessible to your application at runtime.

In Spring Boot applications, the most elegant way to leverage environment variables is through the `application.yml` file, where you can reference these variables directly in your configuration:

```
# application.yml
spring:
    datasource:
        url:      ${DB_URL:jdbc:mysql://localhost:3306/orderdb}
    username: ${DB_USERNAME:dev_user}
    password: ${DB_PASSWORD:dev_password}
payment:
    gateway:
        api-key: ${PAYMENT_API_KEY:sandbox_key}
        timeout: ${PAYMENT_TIMEOUT:30}
product-service:
    url: ${PRODUCT_SERVICE_URL:http://localhost:8081/api/products}
    cache-size: ${PRODUCT_CACHE_SIZE:100}
```

While reviewing our examples, you may have noticed that both the profile-based configuration and environment variables sections use an `application.yml` file, yet they represent different approaches to configuration management. In the externalized configuration example, we focused on creating separate physical configuration files for different environments (such as `application-dev.yml` and `application-prod.yml`), which Spring Boot loads based on the active profile. The environment variables approach, by contrast, uses placeholder syntax, `${VARIABLE_NAME:default_value}`, within a single configuration file to dynamically inject values at runtime. This distinction is important: file-based externalization requires managing complete configuration files for each environment and deploying the correct file alongside your application, while environment variables allow you to maintain a single configuration template where only specific values are overridden at runtime through your deployment platform. The environment variable approach offers greater flexibility in containerized environments such as Docker and cloud platforms, simplifies secret management integration, and enables fine-grained configuration updates without replacing entire files. Many production systems leverage both patterns together—using profile-specific files for substantial configuration differences between environments while employing environment variables for sensitive credentials and frequently changed values. This hybrid approach gives development teams the ability to handle both structural configuration differences and dynamic value injection within a unified framework.

Health checks implementation

Health checks are vital for monitoring the operational status of your API. They allow you to verify that your application is running and ready to respond to the requests after that checkpoint. Implementing health checks can help you detect issues before they impact users, enabling proactive maintenance and quick recovery.

Consider this real-world scenario: An e-commerce company deployed a new version of its Order Management API on Friday afternoon. The deployment seemed successful—all services started without errors, and initial manual tests passed. However, by Monday morning, the customer service team was flooded with complaints about missing orders and inventory discrepancies. Investigation revealed that while the API was technically running, it could not properly connect to the product inventory database due to a configuration issue.

Without proper health checks in place, the deployment was considered successful despite this critical failure. Health checks serve as your application's vital signs monitoring system—much like how a doctor checks your pulse, blood pressure, and temperature to assess your overall health. For containerized applications especially, these checks are crucial because they inform orchestration tools whether your containers should receive traffic, be restarted, or be replaced entirely. A well-designed health check system distinguishes between temporary glitches and serious failures, preventing unnecessary restarts while ensuring genuine problems are addressed promptly.

For our Order Management and Product API, implementing comprehensive health checks means we can confidently automate deployments, knowing that our monitoring will catch issues that might otherwise go undetected until customers are affected. Let's explore how to implement these critical safeguards in our application.

Case scenario: weekend system maintenance

Imagine this scenario: Your team performs routine database maintenance on Saturday morning. After the maintenance is completed, all services appear to restart normally. However, without proper health checks, you might not discover until Monday that while the application is running, it cannot connect to the database properly. This would result in lost orders and frustrated customers.

With effective health checks, you would immediately know whether the database connection was properly restored after maintenance, allowing you to fix issues before they impact business operations.

Implementing basic health checks with Spring Boot Actuator

For our Order Management and Product API, let's implement a simple but effective health check system:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

With this dependency added to your project, Spring Boot automatically exposes a `/actuator/health` endpoint that returns the application's status.

Custom health indicators for order management

Let's create a simple health check for our order repository:

```
@Component
public class OrderRepositoryHealthIndicator implements HealthIndicator {

    private final OrderRepository orderRepository;

    public OrderRepositoryHealthIndicator(OrderRepository orderRepository)
{
        this.orderRepository = orderRepository;
    }

    @Override
    public Health health() {
        try {
            // Simple query to check database connectivity
            orderRepository.count();
            return Health.up().build();
        } catch (Exception e) {
            return Health.down()
                    .withDetail("error", e.getMessage())
                    .build();
        }
    }
}
```

Similarly, create a health check for products:

```
@Component
public class ProductRepositoryHealthIndicator implements HealthIndicator {

    private final ProductRepository productRepository;

    // Similar implementation to check product repository connectivity
}
```

Multi-level health checks

For our Order Management and Product API, implementing multi-level health checks helps distinguish between different types of operational states.

Liveness checks

Liveness checks answer a simple question: "Is the application running?" These checks should be lightweight and avoid dependencies on external systems:

```
@Component
public class LivenessHealthIndicator implements HealthIndicator {

    @Override
    public Health health () {
        // Simple check to verify application is running
        // Should always succeed unless the application is truly dead
        return Health.up().build();
    }
}
```

Configure this in your `application.yml` file:

```
management:
  endpoint:
    health:
      group:
        liveness:
          include: livenessState,diskSpace
```

Readiness checks

Readiness checks answer: "Can the application accept and process requests?" These checks should verify external dependencies:

```
@Component
public class ReadinessHealthIndicator implements ReadinessIndicator {
    private final OrderRepositoryHealthIndicator orderRepoHealth;
    private final ProductRepositoryHealthIndicator productRepoHealth;

    public ReadinessHealthIndicator(
            OrderRepositoryHealthIndicator orderRepoHealth,
            ProductRepositoryHealthIndicator productRepoHealth) {
        this.orderRepoHealth = orderRepoHealth;
        this.productRepoHealth = productRepoHealth;
    }

    @Override
    public Health health() {
        Health orderHealth = orderRepoHealth.health();
        Health productHealth = productRepoHealth.health();

        if (Status.UP.equals(orderHealth.getStatus()) &&
            Status.UP.equals(productHealth.getStatus())) {
            return Health.up().build();
        }

        return Health.down()
                .withDetail("orderRepository", orderHealth.getStatus())
                .withDetail("productRepository",productHealth.getStatus())
                .build();
    }
}
```

Configure this in `application.yml`:

```yaml
management:
  endpoint:
    health:
      group:
        readiness:
          include: readinessState,db,orderRepository,productRepository
```

Component health checks

Component health checks provide detailed status for specific parts of your system:

```
@Component
public class InventoryHealthIndicator implements HealthIndicator {

    private final ProductService productService;

    public InventoryHealthIndicator(ProductService productService) {
        this.productService = productService;
    }

    @Override
    public Health health() {
        try {
            // Check if inventory system is functioning
            boolean inventoryStatus = productService.
                checkInventoryStatus();
            if (inventoryStatus) {
                return Health.up()
                        .withDetail("inventorySystem", "operational")
                        .withDetail("lastSyncTime", new Date())
                        .build();
            } else {
                return Health.down()
                        .withDetail("inventorySystem", "degraded")
                        .build();
            }
```

```
        } catch (Exception e) {
            return Health.down()
                    .withDetail("inventorySystem", "error")
                    .withDetail("message", e.getMessage())
                    .build();
        }
    }
}
```

Practical usage of multi-level health checks

With these levels defined, you can use them for different purposes:

- **Liveness checks**: Used by Docker to determine whether the container should be restarted
- **Readiness checks**: Used to determine whether the API can receive traffic
- **Component checks**: Used by operators to diagnose specific system issues

Update your Docker Compose configuration to use these specific endpoints:

```
version: '3'
services:
  order-api:
    build: .
    ports:
      - "8080:8080"
    healthcheck:
      test: ["CMD", "curl", "-f",
             "http://localhost:8080/actuator/health/liveness"]
      interval: 30s
      timeout: 10s
      retries: 3
      start_period: 40s
    depends_on:
      database:
        condition: service_healthy
```

Basic configuration in application.properties

Configure the health check endpoints:

```
# Enable health endpoint
management.endpoints.web.exposure.include=health,info

# Show health details
management.endpoint.health.show-details=when_authorized

# Enable health groups
management.endpoint.health.group.liveness.include=livenessState
management.endpoint.health.group.readiness.
include=readinessState,db,orderRepository,productRepository
```

While health checks provide essential visibility into your application's operational status, they become even more powerful when integrated with modern deployment approaches. Now that we have established robust health monitoring for our Order Management and Product API, the next logical step is to package our application in a way that leverages these health checks effectively.

Containerization offers the perfect complement to our health check strategy. By encapsulating our application and its dependencies in lightweight, portable containers, we can ensure consistent behavior across different environments while making the most of our monitoring capabilities. Docker containers can be configured to automatically restart based on our health check results, providing a first line of defense against service disruptions.

In the next section, we will explore how to containerize our Order Management and Product API using Docker, creating a deployment pipeline that ensures our application runs reliably in any environment. We'll see how our health checks integrate seamlessly with Docker's built-in monitoring, creating a self-healing system that can detect and recover from many common failure scenarios automatically.

Containerization

Containerization has revolutionized the way applications are deployed by providing a method to package an application along with all its dependencies. This ensures that the application runs consistently across different environments, whether it be development, testing, or production. Unlike traditional **virtual machines (VMs)**, containers share the host operating system kernel, making them lightweight and efficient.

Containerization has revolutionized how we deploy and manage APIs in modern software development. At its core, containerization is the process of packaging an application and all its dependencies—libraries, configuration files, and runtime environments—into a standardized, self-contained unit called a **container**. This approach addresses one of the most persistent challenges in software deployment: "It works on my machine, why doesn't it work in production?"

This process has several key benefits:

- **Portability**: Applications can be moved seamlessly across various environments without modification
- **Scalability**: Containers can be rapidly scaled up or down based on demand
- **Isolation**: Each container runs independently, preventing conflicts between applications
- **Resource efficiency**: Containers share OS resources, making them more lightweight than VMs
- **Consistency**: Developers and operations teams work with identical environments
- **Rapid deployment**: New versions can be deployed in seconds rather than hours or days
- **Self-healing**: Failed containers can be automatically restarted based on health checks

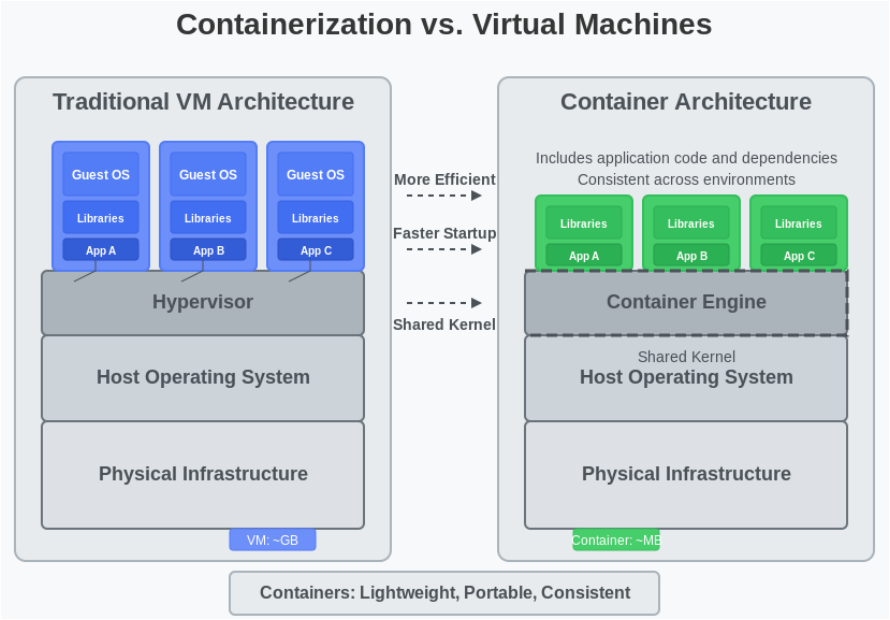


Figure 12.2 – Containerization vs. VMs

How containers work

Containers encapsulate an application and its dependencies into a single unit that can be executed anywhere, providing a consistent environment across various systems. They achieve this by leveraging the host operating system's kernel while maintaining isolation from other containers. This isolation is achieved through namespaces and **control groups** (**cgroups**) in the Linux operating system, which provide the following:

- **Namespaces**: These create separate environments for containers. Each container has its own filesystem, **process ID** (**PID**) space, user IDs, and network interfaces, preventing them from interfering with one another.
- **Control groups (cgroups)**: These limit and prioritize resource usage (CPU, memory, and disk I/O) for containers, ensuring that no single container can monopolize system resources.

Layers of a container

Containers are built from images, which are composed of a series of layers. Each layer represents a set of changes or additions to the base image. This layered architecture provides several advantages:

- **Shared layers**: Multiple containers can share the same underlying layers, reducing disk space usage and speeding up deployment
- **Version control**: Each layer can be versioned, allowing for easy rollbacks to previous versions of the application
- **Efficient builds**: When building a new image, Docker uses caching to only rebuild layers that have changed, resulting in faster build times

Common containerization tools

There are several tools available for containerization, each with its own unique features and benefits. Some of the most common tools are as follows:

- **Docker**: Currently the most popular containerization platform, Docker provides a comprehensive set of tools for building, managing, and running containers. It simplifies the entire container life cycle and has a large ecosystem of community support and resources.
- **Kubernetes**: While not a containerization tool itself, Kubernetes is a powerful orchestration platform that manages containerized applications at scale, automating deployment, scaling, and operations for applications.

- **OpenShift**: Based on Kubernetes, OpenShift adds additional features and tools specifically designed for enterprise environments, providing a PaaS experience for developers.
- **Podman**: An alternative to Docker that allows container management without requiring a daemon. Podman offers a similar command-line interface to Docker, making it easy for users to transition.
- **Rancher**: A complete container management platform that makes it easy to deploy and manage multiple Kubernetes clusters.

For this chapter, we will focus primarily on Docker, as it is the most widely adopted containerization solution today. Its ease of use, extensive documentation, and vibrant community make it an ideal choice for developers looking to implement containerization in their projects.

The typical structure of a Docker image consists of the following:

- **Base layer**: The foundation of the image, such as an operating system or runtime
- **Application layer**: The actual application code and dependencies added on top of the base layer
- **Configuration layers**: Any additional configurations or environment variables specified in the Dockerfile

Docker architecture

To understand containerization fully, we must explore the architecture of Docker, the most widely used containerization platform. Docker operates on a client-server architecture that consists of several key components:

- **Docker Engine**: This is the core component that allows users to build, run, and manage containers. It consists of the **Docker daemon (dockerd)**, which runs in the background and manages images, containers, networking, and storage, and the Docker **command-line interface (CLI)**, which allows users to interact with the Docker daemon.
- **Docker components**:
    - **Images**: Immutable templates used to create containers
    - **Containers**: Running instances of Docker images
    - **Volumes**: Persistent storage for containers
    - **Networks**: Enables communication between containers and external services
    - **Registry**: A repository for storing and distributing Docker images, such as Docker Hub

- **Docker workflow**: The workflow begins when developers create a Dockerfile, which defines the application environment. Docker Engine builds an image from this Dockerfile, which is then stored in a registry. A container is instantiated from the image, and the container runs, providing the application service.

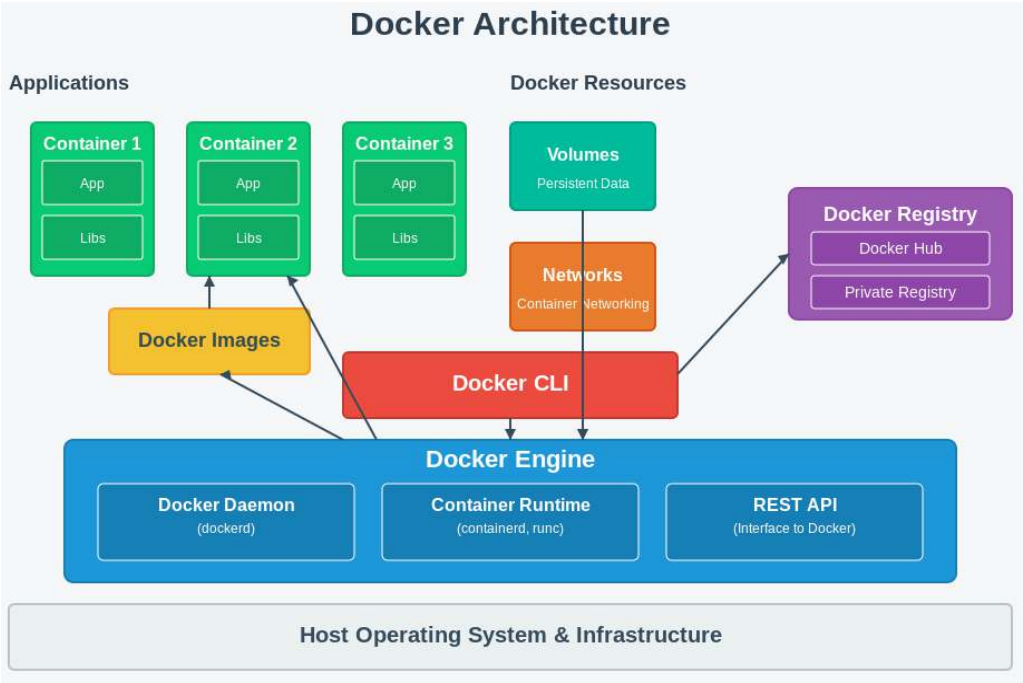


Figure 12.3 – Docker architecture

The diagram in *Figure 12.3* illustrates the Docker architecture we just described, showing how the various components interact to create the environment where our containers run. Understanding this architecture is crucial as we move forward to containerize our Order Management API. The diagram shows the layered structure of Docker, from the host operating system at the bottom to the individual containers at the top, highlighting how Docker efficiently manages resources while providing isolation between applications. Let us implement an example of creating a Docker container based on an image that provides us the **JRE (Java Runtime Environment)** and run it on our machine as described in the next topic.

Example: containerizing a RESTful API

To illustrate the containerization process, let us consider a practical example based on a RESTful API similar to the one found in the following GitHub repository: *RESTful API Book Example*. This example will guide you through the steps to create a Docker container for a Java/Maven-based RESTful API.

Step 1: Project structure

First, ensure your project is structured correctly. A typical structure for a Java/Maven-based RESTful API might look like this:

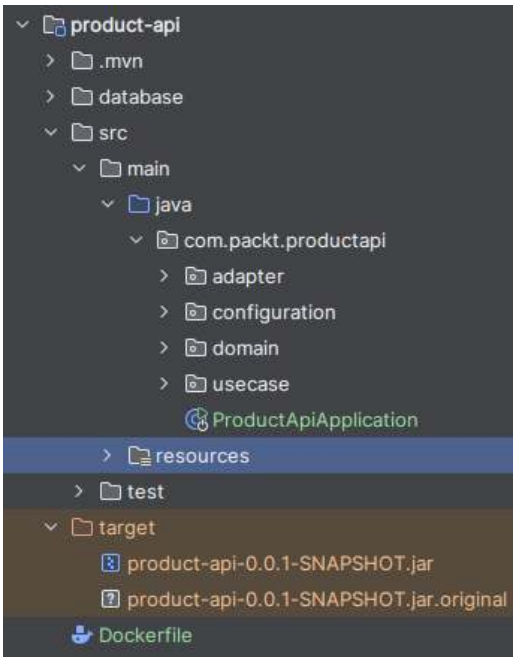


Figure 12.4 – Structure of directories of product-api

Regarding the structure, the essential points are as follows:

- The `target/` folder containing the JAR file
- The main source structure under `src/main/java/`
- The Dockerfile at the root level

In the next steps, we will create the Dockerfile and run commands that require the structure that we presented here in *Figure 12.4*.

Step 2: Creating the Dockerfile

Inside the root of your project directory, create a file named `Dockerfile` with the following content:

```
 # Use an official OpenJDK runtime as a base image
FROM openjdk:21-jdk-slim
 # Set the working directory inside the container
WORKDIR /app
 # Copy the Maven build artifact into the container
COPY target/product-api-0.0.1-SNAPSHOT.jar app.jar
 # Expose the application port
EXPOSE 8080
 # Define the command to run the application
ENTRYPOINT ["java", "-jar", "app.jar"]
```

Step 3: Building the Docker image

Before building the image, ensure your project is packaged correctly. If you are using Maven, you can build the JAR file by running the following:

```
mvn clean package
```

After the build is complete, you can create the Docker image with the following command:

```
docker build -t my-restful-api .
```

The `-t` flag tags the image with the name `my-restful-api`.

Step 4: Running the Docker container

Now that you have built the Docker image, you can run a container from it:

```
docker run -d -p 8080:8080 --name restful-api-container my-restful-api
```

Let's break down this command:

- `-d` runs the container in detached mode (in the background)
- `-p 8080:8080` maps port `8080` on the host to port `8080` in the container
- `--name restful-api-container` assigns a name to the container for easier management

Step 5: Testing the API

Once the container is running, you can test the API by sending a request to it. Open your web browser or use a tool such as `curl` or Postman to send a request:

```
curl http://localhost:8080/api/endpoint
```

Replace `/api/endpoint` with the actual endpoint you want to test. If everything is set up correctly, you should receive a response from your API.

Best practices for container deployment

To ensure effective container deployment, adhering to best practices is crucial:

- **Optimize image size**: Use lightweight base images such as `alpine` or `openjdk:21-slim` to minimize the final image size
- **Minimize layers**: Combine multiple commands into a single `RUN` command in the Dockerfile to reduce the number of layers in the image
- **Use multi-stage builds**: Build the application in one stage and copy only the necessary artifacts to the final image, reducing the overall size
- **Implement health checks**: Use Docker's `HEALTHCHECK` feature to ensure that containerized applications are running properly
- **Secure images**: Regularly scan images for vulnerabilities using tools such as Trivy or Docker Scout to maintain security standards

Challenges and considerations

While Docker simplifies deployment, it also presents challenges that need to be addressed:

- **Security vulnerabilities**: Images can contain vulnerabilities that need to be managed proactively.
- **Networking complexity**: Understanding and configuring container networking can be complex and requires careful planning.
- **Storage management**: Properly managing storage for persistent data can present challenges in containerized environments.
- **Monitoring and logging**: Integrating monitoring and logging solutions is essential for maintaining visibility into container performance.

- **Cloud-specific requirements**: Each cloud provider implements container orchestration differently, with unique services and configurations. Even within the AWS ecosystem alone, effective container deployment requires familiarity with several interconnected services such as ECR, ECS/EKS, IAM roles, VPC networking, and load balancing. While this chapter introduces containerization fundamentals, comprehensive coverage of provider-specific deployment workflows would extend beyond our scope. The preceding example offers a practical starting point, but production deployments typically require deeper platform-specific knowledge and expertise. Organizations often develop specialized DevOps teams focused on these deployment pipelines and infrastructure management concerns.

Containerization enables modern, scalable, and portable API deployments. By understanding Docker's architecture, image life cycle, and best practices, developers can efficiently deploy RESTful APIs in cloud and on-premises environments. This foundational knowledge equips you with the skills necessary to navigate the complexities of modern deployment strategies.

In the next section, we will explore **platform as a service (PaaS)** using AWS Elastic Beanstalk, an enterprise-grade platform for deploying and managing containerized applications.

PaaS

PaaS represents a cloud computing model that provides developers with a complete platform—including hardware, software, and infrastructure—to develop, deploy, and manage applications without the complexity of building and maintaining the infrastructure typically associated with such processes.

During this evolution of cloud computing services, PaaS is on the middle ground between **infrastructure as a service (IaaS)**, which provides only the basic computing infrastructure, and **software as a service (SaaS)**, which provides ready-to-use applications. This convenience makes PaaS particularly valuable for development teams focused on creating and deploying applications rather than managing servers and infrastructure.

Key components of PaaS

A typical PaaS offering includes the following:

- **Runtime environment**: Preconfigured platforms where your application runs
- **Development tools**: Integrated development environments, debugging tools, and version control systems
- **Middleware**: Database management systems, application servers, and messaging queues

- **Operating system**: Managed by the provider and regularly updated for security
- **Networking infrastructure**: Load balancers, firewalls, and security features
- **Scaling capabilities**: Automatic or manual scaling options to handle traffic fluctuations

Benefits of PaaS for API deployment

For RESTful API developers, PaaS offers several significant advantages:

- **Reduced development time**: By eliminating infrastructure setup and management tasks, developers can focus exclusively on API development, significantly reducing time to market
- **Simplified deployment pipeline**: Most PaaS providers offer integrated CI/CD capabilities or seamless integration with popular CI/CD tools, streamlining the deployment process
- **Built-in scalability**: APIs can be scaled horizontally or vertically with minimal configuration, accommodating traffic spikes without performance degradation
- **Enhanced security**: PaaS providers implement robust security measures, including regular updates, encryption, and compliance certifications, reducing the security burden on development teams
- **Global availability**: Leading PaaS providers operate data centers worldwide, enabling the deployment of APIs closer to end users for improved performance

Common PaaS providers for Java applications

Several PaaS providers offer excellent support for Java/Maven-based RESTful APIs:

- **AWS Elastic Beanstalk**: Amazon's PaaS solution that simplifies the deployment of Java applications on AWS infrastructure
- **Microsoft Azure App Service**: Provides a managed platform for building, deploying, and scaling Java web applications
- **Google App Engine**: Google's PaaS offering with specific support for Java applications
- **Heroku**: A cloud platform that supports multiple programming languages, including Java
- **Red Hat OpenShift**: An enterprise Kubernetes platform with strong Java support

PaaS versus traditional deployment

Compared to traditional on-premises deployment or basic IaaS solutions, PaaS offers the following:

Feature	Traditional deployment	PaaS
Infrastructure management	Manual, time-consuming	Automated, minimal effort
Scaling	Complex, requires planning	Simple, often automatic
Updates and patches	Manual process	Managed by provider
Development focus	Split between code and infrastructure	Primarily on application code
Time to market	Longer	Significantly shorter
Initial cost	High	Low

Table 12.1 – Traditional deployment vs PaaS

Considerations when choosing PaaS

While PaaS offers numerous benefits, developers should consider several factors before adoption:

- **Vendor lock-in**: Dependency on provider-specific features may complicate future migrations. This consideration is crucial because many PaaS providers offer proprietary services and tools that can significantly enhance your API development—but at a cost. For example, if you deeply integrate with AWS Elastic Beanstalk's specific deployment mechanisms or Azure App Service's built-in authentication features, you may find your application becomes tightly coupled to that platform. Should business needs change or pricing structures become unfavorable, migrating to another provider could require substantial code refactoring and architectural changes. To mitigate this risk, consider implementing abstraction layers for provider-specific services and maintaining infrastructure as code that could be adapted for different environments.
- **Customization limitations**: Less control over the underlying infrastructure can impact specific optimization needs. This is especially relevant for APIs with unique performance profiles or security requirements. While PaaS platforms handle most infrastructure concerns automatically, this convenience comes with reduced flexibility. For instance, if your Order Management API needs specific database configurations to handle complex inventory queries efficiently, or custom network settings to interact with legacy systems,

a PaaS solution may not allow these fine-grained adjustments. Organizations should carefully assess whether their API requirements fall within the parameters of what the PaaS platform allows, or whether more control through IaaS or container orchestration would be necessary.

- **Compliance requirements**: Ensuring the PaaS provider meets your industry's regulatory standards is non-negotiable for many industries. For an order management system that processes customer information and payment data, compliance with standards such as PCI DSS, GDPR, or HIPAA might be mandatory. Not all PaaS providers offer the same level of compliance certifications or security controls. Some may provide comprehensive compliance documentation and built-in controls, while others might leave more responsibility to the developer. Thoroughly investigate a provider's compliance offerings, data residency options, and security practices before committing your business-critical APIs to their platform.
- **Cost structure**: Understanding the pricing model is essential to avoid unexpected expenses as your API usage grows. PaaS platforms typically charge based on resource consumption—CPU usage, memory allocation, storage, and data transfer. For an Order Management API that might experience seasonal spikes (such as holiday shopping periods), costs could fluctuate dramatically. Some providers offer auto-scaling capabilities that automatically adjust resources based on demand, which can be efficient but also potentially expensive if not properly configured. Carefully analyze the pricing structure, set up monitoring and alerts for resource usage, and consider implementing cost optimization strategies such as resource scheduling for non-production environments.
- **Performance needs**: Evaluating whether the PaaS can meet your specific performance requirements is critical for maintaining user satisfaction. Your Order Management API may need to handle thousands of concurrent requests during peak periods while maintaining low latency for inventory checks and order processing. Different PaaS providers offer varying levels of performance capabilities, from basic shared environments to dedicated premium tiers. Consider aspects such as geographic distribution of data centers (to reduce latency for global customers), available instance sizes, database performance options, and caching capabilities. Conduct load testing on your chosen platform to verify it can handle your expected traffic patterns before fully committing to a provider.

By thoroughly evaluating these considerations in the context of your specific API requirements, you can make an informed decision about whether PaaS is the right approach for your project, and if so, which provider best aligns with your needs.

Having carefully evaluated the considerations we just discussed—vendor lock-in, customization limitations, compliance requirements, cost structure, and performance needs—we have selected AWS Elastic Beanstalk for our practical example. This choice provides an excellent balance between abstraction and control, making it ideal for demonstrating PaaS deployment of our Order Management and Product API.

Additionally, my greater familiarity with AWS allows me to share practical insights that go beyond theoretical implementation, helping you navigate potential challenges that might arise when deploying your own Order Management API.

Let's now walk through a complete deployment of our Java RESTful API on AWS Elastic Beanstalk, showing how these considerations translate into practical implementation decisions.

Practical example: Deploying a RESTful API on AWS Elastic Beanstalk

Let's walk through a complete deployment of a Java RESTful API on AWS Elastic Beanstalk. This example assumes you have a Spring Boot application ready for deployment.

The first step involves preparing your application. First, ensure your application is properly packaged:

```
# Build your application with Maven
mvn clean package
```

This creates a JAR file in the target directory. For Elastic Beanstalk deployment, this JAR should be self-contained with an embedded server.

Next, you need to set up the AWS **Elastic Beanstalk (EB)** environment. Install the AWS CLI and the EB CLI:

```
pip install awscli awsebcli
```

Then, initialize the EB CLI in your project. Navigate to your project's root directory and run the following:

```
eb init
```

This launches an interactive setup where you'll need to do the following:

- Select a Region
- Create or select an AWS credential profile
- Enter an application name

- Choose Java as the platform
- Select the Java version
- Set up SSH for instance access (optional)
- To create an environment, use the following:

```
eb create my-api-environment
```

This command triggers several processes:

- Creates an environment named `my-api-environment`
- Sets up necessary AWS resources (EC2 instances, load balancer, etc.)
- Deploys your application
- Configures health monitoring

During creation, you can also specify the following:

- Environment type (load balanced or single instance)
- Instance type
- EC2 key pair for SSH access
- VPC settings

Next is the application configuration. Create a file named `.ebextensions/java-options.config` in your project root:

```
option_settings:
    aws:elasticbeanstalk:application:environment:
        SPRING_PROFILES_ACTIVE: production
        API_TIMEOUT: 30
        LOGGING_LEVEL_ROOT: INFO
    aws:elasticbeanstalk:container:java:
        JVM Options: -Xms256m -Xmx1024m
    aws:autoscaling:asg:
        MinSize: 2
        MaxSize: 4
    aws:elasticbeanstalk:environment:
        ServiceRole: aws-elasticbeanstalk-service-role
```

This configuration sets the following:

- Environment variables for your application
- JVM memory settings
- Auto-scaling parameters
- Service role for your environment

After making changes to your application, deploy the updates with the following:

```
mvn clean package
eb deploy
```

The EB CLI will automatically upload the new version and perform a rolling update.

Finally, you need to monitor your application. Access monitoring data through the following:

```
eb health
```

Or view detailed metrics and logs via the AWS Management Console:

1. Navigate to Elastic Beanstalk.
2. Select your environment.
3. Click on the **Monitoring** tab.
4. View logs through the **Logs** section.

Advantages and disadvantages of AWS Elastic Beanstalk (EB)

AWS Elastic Beanstalk is a service that makes deploying applications easier by handling the infrastructure work for you. When you upload your code, Elastic Beanstalk automatically sets up servers, load balancers, and monitoring without requiring you to configure these components manually.

This approach sits between two extremes: fully managed services that give you little control, and manual server management that requires deep infrastructure knowledge. Elastic Beanstalk provides a balanced solution where you can focus on writing code while still having access to AWS features when needed.

However, like any deployment option, Elastic Beanstalk has both benefits and drawbacks. Some teams find it perfect for their needs, while others discover it doesn't fit their specific requirements. Understanding these pros and cons will help you decide whether Elastic Beanstalk is the right choice for your RESTful API project.

Let's explore what makes Elastic Beanstalk useful and where it might fall short for API deployment.

Here are the advantages:

- **Simplified operations**: Elastic Beanstalk abstracts away infrastructure complexities, automatically handling provisioning, load balancing, auto-scaling, and monitoring. This allows developers to focus on application code rather than infrastructure management.
- **Integrated development workflow**: The EB CLI provides a streamlined workflow for deployment, making it easy to push updates, monitor health, and manage environments directly from your development environment.
- **AWS service integration**: Elastic Beanstalk seamlessly integrates with other AWS services, including the following:
    - RDS for database management
    - CloudWatch for monitoring and alerting
    - S3 for storage
    - IAM for security
    - CloudFormation for infrastructure definition
- **Platform flexibility**: While abstracting away complexity, Elastic Beanstalk still allows customization through configuration files (`.ebextensions`), enabling fine-grained control over the environment.
- **Versioning and rollbacks**: The platform maintains versions of your deployed application, making it easy to roll back to previous versions if issues occur.
- **Cost management**: You only pay for the underlying resources (EC2 instances, load balancers, etc.) with no additional charge for Elastic Beanstalk itself.

Now, let's look at the disadvantages:

- **Limited control in some areas**: While Elastic Beanstalk provides customization options, you have less control over certain infrastructure aspects compared to direct EC2 or container-based deployments
- **Deployment delays**: The provisioning and deployment process can take several minutes, which may be slower than more container-focused services such as ECS or EKS
- **Cold starts**: New environment creation can take 5 to 10 minutes, making it less suitable for ephemeral or rapid test environment creation

- **Configuration complexity**: As applications grow more complex, managing `.ebextensions` files can become challenging, potentially leading to configuration drift or inconsistencies
- **Vendor lock-in risk**: Heavy dependence on Elastic Beanstalk-specific configuration can make it challenging to migrate to other platforms in the future
- **Custom container limitations**: While supporting Docker, Elastic Beanstalk offers less flexibility in container orchestration compared to dedicated container services like ECS or EKS
- **Debugging challenges**: When issues occur, the abstraction layer can sometimes make troubleshooting more difficult, requiring deeper AWS knowledge to diagnose problems

When to choose Elastic Beanstalk

Elastic Beanstalk is particularly well-suited for the following:

- Teams new to AWS or cloud deployment
- Applications with standard infrastructure requirements
- Projects requiring rapid initial deployment
- APIs with predictable scaling patterns
- Development and testing environments
- Small to medium-sized development teams

For more complex architectures, microservices with custom requirements, or applications requiring fine-grained infrastructure control, solutions such as ECS, EKS, or direct EC2 management might be more appropriate.

Best practices for Elastic Beanstalk deployments

To maximize the benefits of Elastic Beanstalk while mitigating its limitations follow these best practices:

- **Use environment-specific configuration files**: Maintaining separate configuration files for development, staging, and production environments is crucial for ensuring consistency and reliability across your deployment pipeline. When your Order Management API moves through these environments, it encounters different infrastructure requirements, security needs, and performance expectations. By creating environment-specific configurations, you prevent issues such as exposing development debugging tools in production or overloading lower-tier development environments with production-scale resources. This practice also facilitates easier troubleshooting since you can pinpoint whether an issue is environment-specific or inherent to your application code.

- **Implement blue-green deployments**: Blue-green deployments using Elastic Beanstalk's **swap URL** feature allow you to maintain two identical production environments, deploying new code to the inactive environment before switching traffic over. This approach is invaluable for your Order Management API because it eliminates downtime during deployments—customers can continue placing orders without interruption. It also provides an instant rollback mechanism; if a critical issue is discovered after deployment, you can immediately switch back to the previous environment. This practice is particularly important for high-availability applications where even minutes of downtime can result in lost revenue or damaged customer trust.
- **Automate deployments**: Integrating with CI/CD pipelines using AWS CodePipeline and CodeBuild transforms your deployment process from error-prone manual steps to a consistent, repeatable workflow. For an Order Management API, this automation ensures that every deployment follows the same testing, validation, and deployment sequence. This reduces human errors, enforces quality gates before production deployment, and creates auditable records of all changes. Automated deployments also enable more frequent, smaller updates, reducing the risk associated with each deployment while allowing your team to deliver new features and fixes more rapidly.
- **Monitor resource usage**: Setting up CloudWatch alarms to alert on abnormal resource consumption provides early warning of potential issues before they impact customers. Your Order Management API's resource needs will fluctuate with business cycles—holiday shopping periods may drive significantly higher order volumes than normal. Without proper monitoring, these spikes could lead to degraded performance or even outages. CloudWatch alarms allow you to detect trends such as gradually increasing memory usage (potentially indicating a memory leak) or CPU spikes during specific operations. This practice enables proactive scaling decisions and helps identify optimization opportunities in your application.
- **Implement proper health checks**: Designing comprehensive health check endpoints for your API goes beyond simply confirming the application is running—it verifies that all critical components are functioning correctly. For an order management system, this might include checking database connectivity, verifying that inventory services are responsive, and ensuring that payment processing systems are available. Well-designed health checks allow Elastic Beanstalk to make accurate decisions about when to replace failing instances and when to route traffic to healthy ones. This practice directly impacts availability and resilience, preventing scenarios where technically "running" but functionally impaired instances continue to receive customer traffic.

- **Version control configuration**: Keeping `.ebextensions` files in your source code repository ensures configuration consistency and provides a historical record of infrastructure changes alongside code changes. This practice prevents configuration drift between environments and makes it possible to correlate application issues with specific infrastructure changes. For your Order Management API, this means you can easily trace whether an order processing problem began after a code change or an infrastructure configuration change. It also simplifies compliance audits by providing clear documentation of how your infrastructure has evolved over time.
- **Use Elastic Beanstalk environment properties**: Leveraging environment properties for configuration rather than hardcoding values enables you to change application behavior without redeployment. This is particularly valuable for your Order Management API when you need to adjust throttling limits during high-volume periods, modify integration endpoints, or temporarily disable features. Environment properties also enhance security by keeping sensitive information such as API keys or database credentials out of your source code. This practice increases operational flexibility while maintaining security and reducing the need for rapid code deployments to make configuration changes.
- **Regular database backups**: Implementing regular automated backups for RDS instances used by your Order Management API protects against data loss and corruption. Order history, customer information, and inventory data are the lifeblood of your business—losing this data could be catastrophic. Regular backups with appropriate retention policies ensure you can recover from database failures, accidental data deletion, or even malicious attacks. This practice also supports compliance requirements that may mandate specific data protection measures and retention periods.
- **Security group management**: Carefully configuring security groups to allow only necessary inbound traffic forms a critical security boundary for your Order Management API. By restricting network access to only required ports and source IP ranges, you significantly reduce the attack surface available to potential intruders. For instance, your database should only accept connections from your application servers, not from the public internet. This practice helps prevent unauthorized access to sensitive customer and order data while still ensuring legitimate traffic flows smoothly.

- **Log rotation and management**: Configuring proper log rotation prevents storage issues that could impact application availability. Without proper rotation, logs from high-volume periods could consume all available storage, potentially crashing your Order Management API servers. Well-managed logs also provide valuable troubleshooting information when investigating order processing issues or performance problems. This practice balances the need for operational insights with resource constraints, ensuring you maintain visibility into application behavior without risking stability.

PaaS evolution and future trends

As cloud technologies rapidly advance and enterprise requirements become increasingly sophisticated, the PaaS landscape is undergoing a transformative evolution that promises to revolutionize how Java RESTful APIs are developed, deployed, and managed across the digital ecosystem:

- **Kubernetes-based PaaS**: Providers are increasingly offering Kubernetes-based PaaS solutions that combine the simplicity of traditional PaaS with the flexibility of container orchestration
- **Serverless PaaS**: The lines between PaaS and serverless computing are blurring, with many platforms now offering hybrid models that combine the best of both worlds
- **Edge computing integration**: PaaS providers are expanding to edge locations, enabling RESTful APIs to be deployed closer to end users for reduced latency
- **AI-driven operations**: Machine learning is being incorporated into PaaS offerings to provide predictive scaling, anomaly detection, and automated optimization

As these trends develop, Java developers should stay informed about how PaaS offerings are evolving to take advantage of new capabilities that can enhance API deployment and performance.

Summary

In this chapter, we covered the fundamentals of deploying Java RESTful APIs, from preparation through configuration management and health checks to containerization with Docker and platform services via AWS Elastic Beanstalk. These approaches offer practical pathways to move your APIs from development to production.

The deployment methods presented here are deliberately straightforward and suitable for individual developers or small teams looking to quickly implement functional deployment workflows. They provide a solid starting point for those new to API deployment.

In an environment that involves development across bigger structures and teams, it might be necessary to implement a more complex process of deployment that requires a better understanding of the DevOps culture and implementation of pipelines for CI and CD. However, it would require a study of all the concepts and technologies that will pave the way for this goal. It would probably require a new book just for this topic.

As your applications scale and your organizational requirements grow, you'll likely need to adopt more sophisticated deployment strategies—including automated testing pipelines, approval workflows, enhanced security scanning, cloud infrastructure, and comprehensive monitoring solutions. These advanced practices build upon the foundation established in this chapter. Our goal was to provide you with practical, implementable knowledge to successfully deploy your Java RESTful APIs, creating a springboard for your continued exploration of deployment practices as your needs evolve.

packtpub.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

At www.packtpub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

Software Architecture with Spring

Wanderson Xesquevixos

ISBN: 978-1-83588-060-9

- Translate complex business needs into clear and implementable design
- Design resilient systems with common architectural styles
- Transform monolithic applications into microservices following best practices
- Implement event-driven architecture with Kafka
- Monitor, trace, and ensure robust testing, security, and performance
- Identify bottlenecks and optimize performance using patterns, caching, and database strategies
- Automate development workflows with CI/CD pipelines, using Jenkins to deploy the application to Kubernetes

Spring System Design in Practice

Rodrigo Santiago

ISBN: 978-1-80324-901-8

- Implement microservices for scalable, resilient web systems
- Break down business goals into well-structured product requirements
- Weigh tradeoffs between writing asynchronous vs. synchronous services and SQL vs. NoSQL storage
- Accelerate service development and reliability through the adoption of test-driven development
- Identify and eliminate hidden performance bottlenecks to maximize speed and efficiency
- Achieve real-time processing and responsiveness in distributed environments

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share your thoughts

Now you've finished *Mastering RESTful Web Services with Java*, we'd love to hear your thoughts! If you purchased the book from Amazon, please `click here to go straight to the Amazon review page` for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily.

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below:

`https://packt.link/free-ebook/9781835466100`

2. Submit your proof of purchase.
3. That's it! We'll send your free PDF and other benefits to your email directly.