

# Hands-on Spring 6 and Spring Boot 3.0

A pragmatic approach to REST, GraphQL, reactive  
programming and messaging in Spring



Sagara Gunathunga

bpb

# Hands-on Spring 6 and Spring Boot 3.0

A pragmatic approach to REST, GraphQL, reactive  
programming and messaging in Spring



Sagara Gunathunga

bpb

*OceanofPDF.com*

# Hands-on Spring 6 and Spring Boot 3.0

---

*A pragmatic approach to REST,  
GraphQL,  
reactive programming and messaging  
in Spring*

---

**Sagara Gunathunga**



[www.bpbonline.com](http://www.bpbonline.com)

*OceanofPDF.com*

First Edition 2025

Copyright © BPB Publications, India

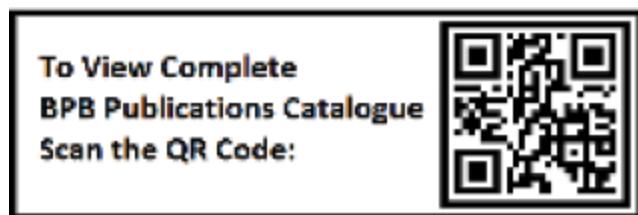
ISBN: 978-93-65892-437

*All Rights Reserved.* No part of this publication may be reproduced, distributed or transmitted in any form or by any means or stored in a database or retrieval system, without the prior written permission of the publisher with the exception to the program listings which may be entered, stored and executed in a computer system, but they can not be reproduced by the means of publication, photocopy, recording, or by any electronic and mechanical means.

#### **LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY**

The information contained in this book is true to correct and the best of author's and publisher's knowledge. The author has made every effort to ensure the accuracy of these publications, but publisher cannot be held responsible for any loss or damage arising from any information in this book.

All trademarks referred to in the book are acknowledged as properties of their respective owners but BPB Publications cannot guarantee the accuracy of this information.



[www.bpbonline.com](http://www.bpbonline.com)

[OceanofPDF.com](http://OceanofPDF.com)

**Dedicated to**

*My mom **Reeta** and My wife **Sajee**  
for everything they have done for me*

*Dr. **Sanjiva Weerawarana** for his inspiration*

*OceanofPDF.com*

# About the Author

**Sagara Gunathunga** is a software architect with over 15 years of industry experience, specializing in application development, integration, and identity solutions. His early career saw him immersed in the Spring Framework that led him to apply Spring-related technologies like Spring Security, Spring Integration, and Spring Services to a number of enterprise projects.

After joining WSO2, as the product lead he led the effort to integrate Spring profile support and JavaEE profile into the WSO2 Application Server. Additionally, he was a key contributor to Spring support within WSO2's Java microservices framework, MSF4J. Currently, Sagara serves as Head of DevRel IAM at WSO2.

Sagara has shared his expertise at international conferences such as the European Identity & Cloud Conference, Consumer Identity World, ApacheCon NA, and WSO2Con events. He has been an open-source advocate and a PMC Member & Committer at the Apache Software Foundation and contributed to projects like Apache Axis2, Apache WebServices, Apache Synapse, and Apache Camel as well.

*OceanofPDF.com*

# Acknowledgement

I would like to express my sincere gratitude to all those who contributed to the completion of this book.

First and foremost, I extend my heartfelt appreciation to my family for their support and encouragement throughout this journey. Without their love and encouragement, this book would not exist.

I would also like to extend my special thanks to Dr. Sanjiva Weerawarana for his inspiration throughout my career and Prabath Siriwardena for his encouraging words.

I am immensely grateful to BPB Publications for their guidance and expertise in bringing this book to fruition. Their support and assistance were invaluable in navigating the complexities of the publishing process.

I would also like to acknowledge the reviewers, technical experts, and editors who provided valuable feedback and contributed to the refinement of this manuscript. Their insights and suggestions have significantly enhanced the quality of the book.

Last but not least, I want to express my gratitude to the readers who have shown interest in the book. Your support and encouragement have been deeply appreciated.

Thank you to everyone who has played a part in making this book a reality.

*OceanofPDF.com*

# Preface

Since the first release of the Spring Framework, it has been almost two decades, yet the popularity of the Spring Framework and demand for Spring developers has never faded, and the number of enterprise application and service development frameworks for Java has. The Spring Boot project started a decade ago hugely affected the current success of Spring within Java developers due to its simplifications and production-ready opinionated configuration approach.

Comprising eleven insightful chapters, this book covers fundamental concepts of the Spring Framework, such as dependency injections and building blocks of Spring Boot's opinioned programming model. The book gradually introduces and covers essential concepts and walks through the hands-on experience, providing everything required to build production-grade web applications and APIs using Spring. From there, the book introduces the concept of reactive programming and how to use reactive programming with Spring and proceeds to discuss Spring messaging and real-world applications of Spring messaging.

The tenth chapter of the book focuses on providing the reader with absolutely required particle concepts of running a Spring Boot application in production, such as running an application in different development environments, configuring SSL/TLS for security, packaging options such as Docker and GraalVM, connecting with identity providers for production-grade security and explore observability options.

This book is designed to be used by readers with basic Java development experience and does not require prior knowledge of Spring and Spring Boot. The book also serves as a source for those who have used Spring and Spring Boot previously to upgrade their knowledge to the latest advances of Spring and Spring Boot projects and get an introduction to new programming paradigms such as reactive programming and native images.

Through practical examples, comprehensive explanations, and a structured approach, this book also introduces a sample project throughout the book, where each chapter focuses on developing a certain part of the sample project.

**Chapter 1: Introduction to Spring and Spring Boot** - This chapter provides a concise overview of the Spring Framework's history and its rise in popularity. It then looks into fundamental architectural concepts like dependency injection and declarative programming, accompanied by basic examples to aid in understanding the concepts presented in this book. The second part of the chapter explores the essential features of the Spring Boot framework and how they facilitate the development of production-ready enterprise applications and services.

**Chapter 2: Getting Started with Spring Boot** - This chapter walks you through creating your first Spring Boot application begin by covering the setup of your development environment, the utilization of Spring Initializer, and the typical structure of a Spring project. Next, we look into the VS Code SpringBoot IDE to craft your Hello World application, highlighting the Spring and Spring Boot fundamentals introduced in the previous chapter. Additionally, we delve into building, running, and packaging your first application. Finally, we explore Java build system alternatives, application packaging choices, and other language support available in Spring.

**Chapter 3: Spring Essentials for Enterprise Applications** - As the final chapter of the fundamental section, this chapter introduces three vital concepts important for building any production-ready enterprise application. We start by introducing **test-driven development (TDD)** as a development methodology, highlighting its role in improving robustness and testability and how Spring/Spring Boot fits into the TDD. Subsequently, we are looking into writing our first test case using the Spring test module and discussing how mocking technologies such as Mockito can be used. Moving forward, we discuss security as another critical concept, providing an introduction to the Spring Security project and its features. Lastly, we introduce the concept of observability and the golden triangle of observability - Metrics, Logs, and Traces.

**Chapter 4: Building Spring MVC Web Applications** - This chapter provides an introduction to MVC architecture and the Spring MVC project, explaining the concepts of Model, View, and Controller separately. Additionally, this chapter provides a discussion on view templating technologies and introduces Thymeleaf. Subsequently, we introduce the sample use case that we will use throughout this book: the implementation of a book management system, which moves into implementing the sample project using Spring MVC. Following this, we look into the practical aspects of handling errors and error pages, session management, and input validation.

**Chapter 5: Working with Spring Data Access** - In this chapter, we introduce you to the Spring Data project and explore various options available for implementing persistence, such as JDBC, ORM, and JPA. We begin by implementing the persistence layer of the sample application using the Spring JDBC template, followed by utilizing JPA and Hibernate as the secondary options. Additionally, we cover techniques for testing applications both with and without databases.

**Chapter 6: Building RESTful Spring Services** - In this chapter, we begin by examining RESTful design concepts. Subsequently, we proceed to implement a RESTful service in accordance with the requirements of the sample use case. Additionally, we explore the utilization of Swagger/OpenAPI specification for describing RESTful services and for testing. Moving forward, we will delve into the usage of the REST Template for consuming a RESTful service.

**Chapter 7: Building GraphQL Spring Services** - In this chapter, we start by exploring GraphQL design concepts and the GraphQL Schema Definition Language. Following this, we implement a GraphQL service to meet the requirements of the sample use case. Additionally, we look into the use of GraphQL UI clients for testing purposes. Finally, we discuss methods for securing GraphQL services and implementing observability measures.

**Chapter 8: Building Reactive Spring Applications** - In this chapter, we begin by introducing the reactive programming model and its benefits. Subsequently, we look into implementing a reactive service using Spring Webflux and explore supporting persistence with Spring Data R2DBC.

**Chapter 9: Working with Spring Messaging** - We begin by discussing the architectural basics of asynchronous communication and eventing.

Following this, we proceed to implement sample use cases using JMS and MQTT, utilizing ActiveMQ and RabbitMQ as brokers.

**Chapter 10: Running Spring Boot in Production** - In this important chapter, we will look into some of the most important concepts for running Spring Boot applications in production environments. Beginning with Spring profile to run the same application in different development environments, packaging options such as Docker and GraalVM, and configuring SSL/TLS for your application. The chapter also covers how to integrate your applications with identity providers and observability options for production applications.

**Chapter 11: Emerging Trends in Spring Framework** - In this final chapter, we explore key emerging topics in the Spring ecosystem. We start with an introduction to Spring AI, which simplifies integrating artificial intelligence (AI) capabilities into Spring applications. Next, we discuss Spring Authorization Server and Spring Vault projects. Finally, we highlight upcoming features in the Spring Framework and Spring Boot projects.

# Code Bundle and Coloured Images

Please follow the link to download the *Code Bundle* and the *Coloured Images* of the book:

<https://rebrand.ly/b6a49b>

The code bundle for the book is also hosted on GitHub at <https://github.com/bpbpublications/Hands-on-Spring-6-and-Spring-Boot-3.0>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at <https://github.com/bpbpublications>. Check them out!

## Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

[errata@bpbonline.com](mailto:errata@bpbonline.com)

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Did you know that BPB offers eBook versions of every book published, with PDF and ePUB files available? You can upgrade to the eBook version at [www.bpbonline.com](http://www.bpbonline.com) and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at :

[business@bpbonline.com](mailto:business@bpbonline.com) for more details.

At [www.bpbonline.com](http://www.bpbonline.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 BPB books and eBooks.

### 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 [business@bpbonline.com](mailto:business@bpbonline.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 [www.bpbonline.com](http://www.bpbonline.com). We have worked with thousands of developers and tech professionals, just like you, to help them share their insights 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.

### Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit [www.bpbonline.com](http://www.bpbonline.com).

## Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

[https://discord\(bpbonline.com](https://discord(bpbonline.com)



*OceanofPDF.com*

# Table of Contents

## 1. Introduction to Spring and Spring Boot

- Introduction
- Structure
- Objectives
- Introduction to Spring Framework
- Dependency injection
- Spring IoC container and beans
- Introduction to Spring Boot
- Spring Boot starters
- Spring Boot auto-configuration
- Spring Boot actuators
- What is new in Spring 6 and Spring Boot 3
- Conclusion

## 2. Getting Started with Spring Boot

- Introduction
- Structure
- Objectives
- Setting up the developer environment
- Maven vs. Gradle
- Setting up the IDE
- Spring Initializr
- Writing your first Spring Boot application

Packaging options

Conclusion

### **3. Spring Essentials for Enterprise Applications**

Introduction

Structure

Objectives

Building enterprise applications

*Testability*

Test-driven development

Writing your first Spring Boot test

*Writing mock tests*

*Writing tests for web applications*

Securing Spring Boot applications

*Spring Security*

*Writing your first Spring Security app*

Observability of an application

Spring Boot Actuators

Spring Boot metrics

Spring Boot logging

Spring Boot tracing

Conclusion

### **4. Building Spring MVC Web Applications**

Introduction

Structure

Objectives

Introduction to Spring MVC

Writing your first Spring MVC web application

BookClub use case

## *Building BookClub application*

- Data validation
- Error handling
- Securing Spring MVC applications
- Observability
- Conclusion

## **5. Working with Spring Data Access**

- Introduction
- Structure
- Objectives
- Data access in Spring
- Spring JDBC Data Access
- Spring Data JPA Data Access
- Spring Data MongoDB Data Access
- Conclusion

## **6. Building RESTful Spring Services**

- Introduction
- Structure
- Objectives
- Introduction to REST
- Spring RESTful services
- Error handling
- REST service testing
- REST service documentation
- RESTful Clients
- Securing RESTful services
- Conclusion

## **7. Building GraphQL Spring Services**

- Introduction
- Structure
- Objectives
- Introduction to GraphQL
- Spring GraphQL services
- GraphQL service testing
- GraphQL clients
- Securing GraphQL services
- Conclusion

## **8. Building Reactive Spring Applications**

- Introduction
- Structure
- Objectives
- Introduction to reactive programming
- Spring reactive services
- Writing test for reactive services
- Functional endpoint model
- Building reactive clients
- Conclusion

## **9. Working with Spring Messaging**

- Introduction
- Structure
- Objectives
- Introduction to messaging
- Spring Messaging with ActiveMQ
- Spring Messaging with RabbitMQ

Conclusion

## 10. Running Spring Boot in Production

Introduction

Structure

Objectives

Spring Profiles

Spring Boot with Docker

Spring Boot with GraalVM

Use Jetty and Undertow with Spring Boot

Configuring SSL for Spring Boot applications

Identity provider integration for Spring Boot

Working with logs, metrics, and traces

Conclusion

## 11. Emerging Trends in Spring Framework

Introduction

Structure

Objectives

Introduction to Spring AI

Introduction to Spring Authorization Server

Introduction to Vault

Upcoming features in Spring

Conclusion

## Index

# CHAPTER 1

# Introduction to Spring and Spring Boot

## Introduction

This chapter will provide a concise overview of the Spring Framework's history and its rise in popularity. Then, we will explore the fundamental architectural concepts like dependency injection and declarative configuration model, accompanied by basic examples to aid in understanding the concepts presented in this book. In the latter part of the chapter, we will explore the essential features of the Spring Boot framework and how they facilitate the development of production-ready enterprise applications and services.

## Structure

The chapter covers the following topics:

- Introduction to Spring Framework
- Dependency injection
- Spring IoC container and beans
- Introduction to Spring Boot
- Spring Boot starters
- Spring Boot actuators

- What is new in Spring 6 and Spring Boot 3

## Objectives

By the end of this chapter, you will understand the key features of Spring and Spring Boot frameworks and why they are so popular. You will also learn about core design principles like Spring dependency injection. Furthermore, we will discuss essential framework features such as the Spring container, beans, and Spring configuration models, laying a strong foundation for the rest of the book.

## Introduction to Spring Framework

The Spring Framework 1.0 version was released in 2004, but its origins date back to concepts and code presented in the book *Expert One-on-One J2EE Design and Development* (Wrox, 2002) by *Rod Johnson* in 2002. Rod's aim was to provide a lightweight alternative to the complex and heavy mainstream **Java 2 Enterprise Edition (J2EE)** technology of that time. The original framework code was named **Interface21 framework**, and it was later renamed and released as the Spring Framework as we know it today by an open-source project led by *Rod Johnson, Juergen Hoeller, Yann Caroff, and others*.

As of the time of writing this book, it has been exactly two decades since the release of the 1.0 version of the Spring Framework. However, its popularity and relevance among Java developers have never declined. In fact, the core concepts introduced in the Spring Framework have inspired and been adopted by later JSR specifications, such as Dependency Injection for Java (JSR-330) and Enterprise JavaBeans 3.0 (JSR-220).

Before we go further, we should answer an important question, what exactly the Spring Framework is? The simplest answer would be Spring is a lightweight framework to build enterprise Java applications. To elaborate further, we can break this definition into the following points:

- **A lightweight framework:** Here, the term **lightweight** does not refer to the size of the framework distribution or the size of an application developed using Spring. Instead, it refers to the architectural fundamentals of the framework. Spring promotes the use of simple Java

objects, sometimes called **Plain Old Java Objects (POJOs)** or Java beans, instead of complex objects tightly coupled with internal interfaces of a framework. The Spring Bean container, which we will explore in this chapter, allows developers to focus on implementing business logic without worrying about managing object lifecycle, dependency, and configuration management by themselves.

- **Building Java applications:** One distinguishing characteristic of the Spring Framework is its versatility; it is not specific to certain types of applications. With Spring, you can build a wide range of applications, from traditional server-side web applications to **Single-Page Applications (SPAs)**, RESTful services to GraphQL services, standalone applications, batch processes, periodic jobs, and modern reactive applications.
- **Building enterprise applications:** Enterprise applications are not just another type of application; rather, they represent a specific quality characteristic that a software system should possess to be considered an enterprise application. These characteristics include scalability, reliability, security, customizability, extensibility, and observability, among others. The Spring Framework is designed to support the adoption of these enterprise features seamlessly to your application. For example, Spring simplifies the implementation of security features in a web application. This includes defining public and protected resources and implementing user authentication and authorization, either as part of the application or by integrating with an identity provider.

Let us take a quick look at the following core features of the Spring Framework that have contributed to its widespread adoption. We will explore these concepts in detail throughout this book and understand how they can be used to build a complete solution based on an example use case:

- **Inversion of Control container:** The **Inversion of Control (IoC)** container is the core of the Spring Framework, it acts as the container for the components of an application, manages the lifecycle and configuration of objects (beans), and uses **dependency injection (DI)** to decouple application components.
- **Aspect-oriented Programming:** **Aspect-oriented Programming (AOP)** provides modularization of cross-cutting aspects, such as

logging, transaction management, and security of an application, improving code maintainability by reducing coupling and improving cohesion within a specific module.

- **Web framework:** Spring MVC employs a model-view-controller architecture that is highly configurable and supports various template engines, including Thymeleaf, Mustache, and FreeMarker, for generating dynamic content. It facilitates the development of both RESTful services and traditional web applications.
- **Relational data access:** It simplifies working with relational data sources by offering templates and helper utilities to reduce the need for repetitive JDBC coding when querying, updating, and processing results. Spring data access support integrates with popular ORM and similar frameworks such as Hibernate, JDO, iBATIS, jOOQ, and **Java Persistence API (JPA)**.
- **Non-relational data access:** It simplifies interaction with non-relational data sources by providing templates and helper utilities that support a wide range of non-relational data sources, including MongoDB, Couchbase, Neo4j, Cassandra, Redis, Solr, Elasticsearch, and LDAP.
- **Reactive framework:** Spring WebFlux, introduced in Spring 5.0, is built on the reactive streams paradigm to support the development of asynchronous, non-blocking, and event-driven applications. It serves as an alternative to the traditional Spring MVC framework, enabling developers to create highly scalable and resilient applications.
- **Messaging:** Spring Framework provides a comprehensive messaging model with templated abstractions, enabling integration with a variety of message brokers such as RabbitMQ, ActiveMQ, Artemis, and Kafka. Dedicated sub-projects like Spring AMQP and Spring Kafka further enhance messaging capabilities within the Spring ecosystem.
- **Security:** Spring Security offers a wide range of security features to protect Spring-based applications, covering aspects such as authentication, authorization, and defense against common security vulnerabilities. Additionally, the Spring Security project supports securing services by implementing both server and client-side implementations of OAuth2 and OpenID Connect standards.

## Dependency injection

**Inversion of Control (IoC)** is a fundamental and broad design principle that inverts the control flow of a program compared to traditional procedural programming. In Inversion of Control, the framework or an external entity is responsible for the control of the flow and calls into the custom code or components. **Dependency injection (DI)** is a specific design pattern very closely related to IoC; however, they are not exactly the same. In the context of Java programming, DI primarily involves providing dependencies to an object from outside rather than creating them inside the object. It is more accurate to describe Dependency Injection as a pattern that can be used to achieve IoC.

A proper understanding of the concepts of IoC and DI is essential to successfully use the Spring Framework in your projects. Let us explore both of these concepts with a real-world example.

Who would not enjoy starting their day with a perfect café-style coffee? Crafting that perfect cup manually involves multiple tricky steps, each requiring experience and judgment. It begins with selecting the best water source and boiling it just right. Then, grinding the coffee beans to the perfect level is crucial. Next, brewing one or two shots of espresso as per your preference. Heating up skimmed milk using a milk frothing pitcher and steam wand follows. Finally, the artistic touch as you mix the espresso with frothed milk. While some enjoy each step, for many, it is time-consuming and error-prone for daily use.

In contrast to that, using an automatic coffee machine transforms this morning ritual into a hassle-free experience. You can simply fill the machine with tap water, as its filtering mechanism ensures no impact on taste. Then, through the machine's interface, you specify the grinding level, whether it is espresso or long-black. The machine precisely boils the water to the required temperature and measures the exact amount of coffee beans, grinding them according to the configured grinding level. Next, the machine's control unit supplies the ground and tamped coffee to the brewing unit, passing the right amount of boiled water through the coffee for a perfect brew. The frothing pitcher and steam wand of the machine guide you through the process, allowing you to enjoy your artistic work without any hassle.

We can draw a parallel between traditional procedural programming and manually making coffee, where each step requires close attention and control. On the other hand, an automatic coffee machine simplifies the process; you supply the required inputs like water and coffee beans and specify your desired outcome, such as one shot or double shot, grinding levels, and type of coffee. Unlike the manual process, you do not need to worry about the sequence of steps or the details of water temperature and coffee bean grinding. The machine's separate units or modules handle each step efficiently, while the control unit orchestrates the entire process by providing inputs to each unit and passing the outputs to the next step. Essentially, the control unit takes control away from you, this is a good example of IoC and DI in action in the real world. We can closely compare the IoC container of the Spring Framework to the control unit of our coffee machine, of course, we use Java beans instead of coffee beans in Spring.

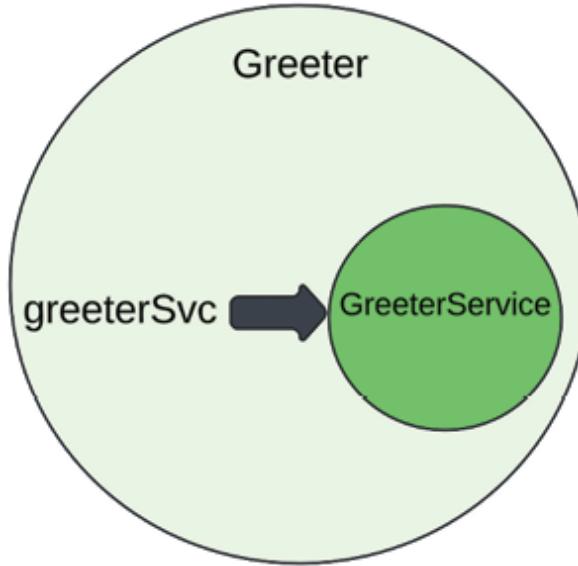
Let us look at the following code segment from the **GreetingApplication** that we are going to develop in the next chapter.

```
1. public class GreetingService {  
2.  
3.     public String greet() {  
4.         // .....  
5.     }  
6. }
```

The **GreetingService** has a method called `greet` that returns either **Good Morning** or **Good Afternoon** based on the time of the day. Now, let us take a look at a typical application code that is called the `greet` method:

```
1. public class greeter {  
2.  
3.     public void greet() {  
4.         greetingsvc greetingsvc = new greetingsvc();  
5.         string msg = greetingsvc.greet();  
6.         system.out.println(msg);  
7.     }  
8. }
```

As you can clearly notice, in *line number 4*, we have directly instantiated **GreetingService** class to create the `greetingsvc` object within the **GreetingApplication** class. This direct coupling can be seen in the following figure:



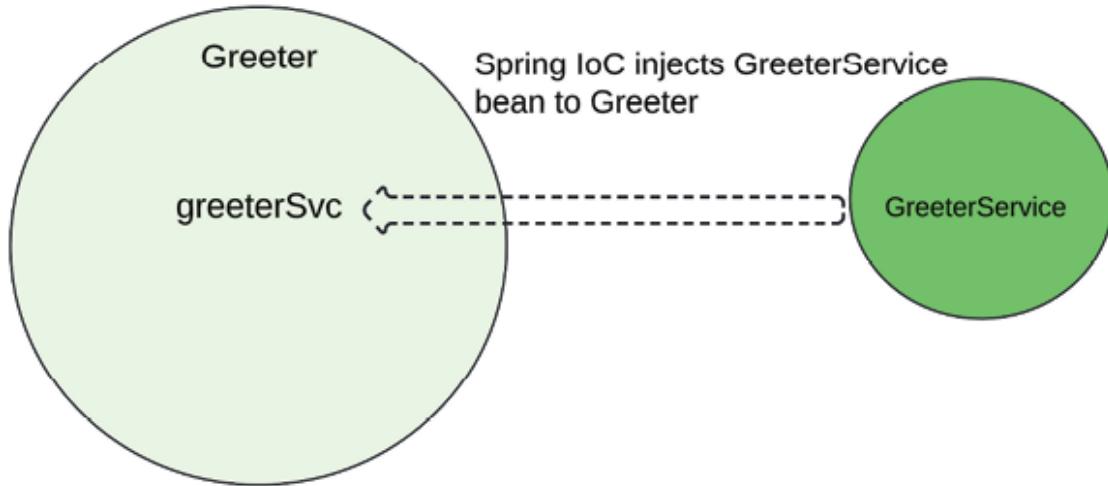
*Figure 1.1: Coupling between Greeter and Greeter service*

Now, let us look at the same **GreetingApplication** rewritten using the Spring Framework:

```

1. public class Greeter {
2.
3.     @Autowired
4.     GreetingService greetingSvc;
5.
6.     public void greet() {
7.         String msg = greetingSvc.greet();
8.         System.out.println(msg);
9.     }
10. }
```

Unlike the previous example, we have not directly created the **greetingSvc** object in our application. Instead, we have informed the Spring IoC container that our application expects a suitable **GreetingService** type object for the **greetingSvc** field. This is one way of applying dependency injection in Spring applications. In this scenario, Spring scans the classpath to find a suitable Java object to inject into the **GreetingApplication**. This approach can be seen in the following figure:



*Figure 1.2: Dependency injection*

In the Spring Framework, there are three main approaches supported for applying dependency injection:

- Constructor injection
- Setter injection
- Field injection

In **constructor injection**, dependencies are provided as arguments to the class constructor during instantiation. The following code segment provides an example of constructor injection:

```

1. public class Greeter {
2.
3.     private GreetingService greetingSvc;
4.
5.     @Autowired
6.     public GreetingApplication(GreetingService greetingSvc) {
7.         this.greetingSvc = greetingSvc;
8.     }
9. }
```

Constructor injection is generally recommended for required dependencies because it ensures dependencies are available when an object is created, usually at the start-up time of an application. Otherwise, it can throw exceptions if constructors fail to satisfy dependencies. Constructor injection also promotes immutable objects as dependencies. Immutable objects are objects whose state cannot be modified after creation and remains constant throughout their lifetime. Immutable objects offer several advantages, they

can be cached, resistant to unintended state changes, and are easy to use in applications requiring concurrency and thread safety. Additionally, constructor injection exposes excessive coupling of your application components, making it easier to identify classes with too many dependencies.

Among Spring developers, there is a debate about constructor injections and cyclic dependencies. If cyclic dependencies are found when constructor injections are used, it can cause exceptions. This behavior is sometimes seen as a drawback of constructor injections. However, it can also be viewed as a helpful tool for detecting cyclic dependencies and resolving them through code refactoring. When it is absolutely necessary to use cyclic dependencies, you could employ lazy-loading or another dependency injection approach like setter injections.

In **setter injection**, dependencies are provided by calling setter methods on objects after instantiation. This approach is mainly suitable for optional dependencies. As discussed earlier, setter injections are useful in situations involving circular dependencies or partial dependencies as well. The following code segment provides an example of setter injection:

```
1. public class Greeter {  
2.  
3.     private GreetingService greetingSvc;  
4.  
5.     @Autowired  
6.     public setGreetingSvc (GreetingService greetingSvc) {  
7.         this.greetingSvc = greetingSvc;  
8.     }  
9. }
```

In **field injection**, dependencies are injected directly into fields annotated with **@Autowired**. Field injections are convenient and easy to use as they do not require adding boilerplate code to your classes. However, they are the least recommended approach for developing enterprise Spring applications due to drawbacks related to testability, coupling, immutability, and lack of clarity. The following code segment provides an example of field injection:

```
1. public class Greeter {  
2.  
3.     @Autowired  
4.     GreetingService greetingSvc;
```

```
5.
6. public void greet() {
7.     String msg = greetingSvc.greet();
8.     System.out.println(msg);
9. }
10. }
```

## Spring IoC container and beans

In the previous section, we talked about how the IoC container is the heart of the Spring Framework, responsible for implementing the IoC design pattern through DI. In this section, let us delve into the Spring IoC container in detail. But first, let us define what a bean means in the context of the Spring Framework. A bean is simply a Java object managed by the Spring IoC container. Put even simply, it is an instance of a Java class that Spring takes care of, handling its instantiation, configuration, and management.

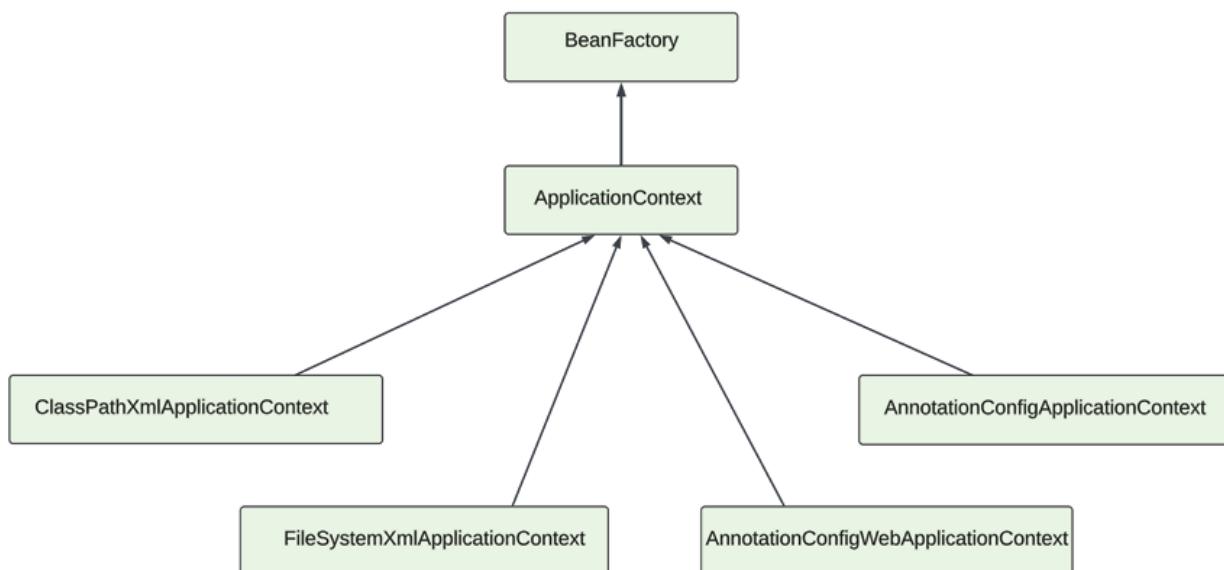
There are two interfaces that are particularly important when discussing Spring IoC container, which are:

- **BeanFactory**: The **BeanFactory** interface is the root interface for the IoC container. It provides the basic functionality for managing the lifecycle of beans by instantiating, configuring, and managing based on the configuration metadata provided.
- **ApplicationContext**: The **ApplicationContext** interface extends the **BeanFactory** and incorporates additional enterprise-specific functionalities, such as resolving textual messages from a properties file, publishing events, defining bean scopes, and loading multiple configurations.

Generally speaking, it would be fair to call **ApplicationContext** the Spring IoC container, and as in our applications, it is the main interaction point to interact with the Spring IoC container. There are a number of implementations of **ApplicationContext** interface available to support the loading of configuration metadata from application-specific contexts.

The **ClassPathXmlApplicationContext** is a commonly used implementation during the early releases of the Spring Framework and is responsible for loading metadata definitions from one or more XML files located in the classpath. Another implementation,

**XmlWebApplicationContext**, is used to load metadata definitions from one or more XML files in the config file directories of Java web applications, either with the help of web.xml or using **WebApplicationInitializer**. Introduced with the Spring 3.0 release, **AnnotationConfigWebApplicationContext** loads metadata definitions from one or more Java-based configuration classes annotated with **@Configuration**. The following diagram illustrates some widely used implementations of **ApplicationContext**:



*Figure 1.3: Most common implementations of ApplicationContext*

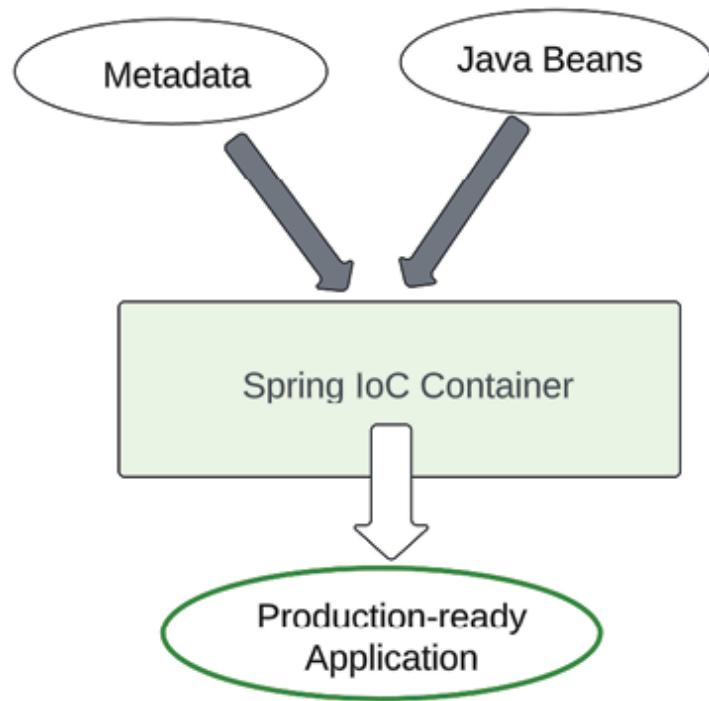
The following code segment shows you how you can use **ApplicationContext** and access a bean. However, thanks to the capabilities offered by the Spring Boot framework, now, in most common cases, we do not need to explicitly create the **ApplicationContext**.

```

1. public class GreetingApplication {
2.
3.     public static void main(String[] args) {
4.         // Create the ApplicationContext
5.         ApplicationContext context = new
6.             AnnotationConfigApplicationContext(AppConfig.class);
7.
8.         // Get the bean from the context
9.         Greeter greeter = context.getBean(Greeter.class);
10.        greeter.greet();
11.    }
  
```

10. }

To recap what we have discussed so far using the coffee machine analogy from the previous section, we compare the Spring IoC container to the control unit of a coffee machine. We input water and coffee beans to produce our perfect coffee, using the coffee machine's interface to configure settings such as grinding level and coffee type. Similarly, we provide Java bean classes representing domain objects and business logic to the Spring IoC container as input, along with configuration metadata, which is like instructions for the coffee machine. The Spring IoC container then produces a set of beans and assembles our application. The following diagram illustrates this concept:



*Figure 1.4: Spring IoC container*

There are mainly three approaches to defining metadata for a spring application, which are listed as follows:

- XML configuration
- Annotation-based configuration
- Java-based configuration

Let us take a detailed look at them:

- **XML configuration:** This is the traditional method of configuring

beans in Spring, which was the only option supported until the Spring 2.5 release. Beans are defined in an XML file, and the ApplicationContext is loaded from this file. There are multiple implementations of ApplicationContext to support different application environments, such as searching metadata files from file locations, the classpath, or the context of a web application. The following example code segment demonstrates how you could define beans using XML:

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.   xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.spring
   framework.org/schema/beans/spring-beans.xsd">
5.
6.   <bean id="greetingSvc" class="com.example.GreetingService"/>
7.
8. </beans>
```

- **Annotation-based configuration:** Starting from Spring 2.5, it began supporting loading configuration metadata from annotated Java classes as an alternative to XML configuration. Annotations are directly added to Java classes to indicate how they should be handled by the Spring IoC container. Spring uses a component scanning mechanism to automatically detect and register beans annotated with annotations. Component scanning is disabled by default, and you can enable it using the `<context:component-scan>` tag in XML or the `@ComponentScan` annotation in Java. Once component scanning is enabled, Spring's annotation processing starts the scanning process. Spring utilizes Java's reflection capabilities to inspect annotated classes during the initialization phase. This approach is very convenient and easy to use, eliminating the need for XML configurations. However, the main drawback is that your bean classes become tightly coupled with Spring-specific annotations, and in some cases, component scanning can be a costly operation as well. The following example code segment demonstrates how you could define beans using annotations:

```
1. @Component
2. public class GreetingService {
3.
4.   public String greet() {
5.     //.....
```

```
6.  }
7. }
```

- **Java-based configuration:** This approach was introduced with Spring 3.0 and uses Java classes annotated with **@Configuration** to explicitly define bean configuration. The Java-based approach eliminates the drawback of tightly coupling your Java beans with Spring annotations, as you can only use annotations in the Java classes related to the configuration classes. Java-based configuration is more explicit than annotation-based configuration. The following example code segment demonstrates how you could define Java-based configuration. In this example, **@Configuration** annotation indicates these classes as configuration classes, while **@Bean** annotation is used to define individual beans.

```
1. @Component
2. public class GreetingService {
3.
4.     public String greet() {
5.         //.....
6.     }
7. }
```

At the beginning of this section, we briefly discussed that a Spring bean is an object instantiated, assembled, and managed by the Spring IoC container. Let us look into Spring beans before concluding this section. Although Spring beans are POJOs, within the Spring Framework, they have a defined lifecycle. The main advantage of this lifecycle is that developers can listen to lifecycle events using extension points known as lifecycle callbacks to achieve customizations such as adding custom initialization and cleanup logic, logging and monitoring, and testing. Developers can even intervene in the activities of the Spring IoC container to some extent, which is important when integrating Spring with other third-party frameworks. We will explore some of the lifecycle events later.

Spring also defines a concept called bean scope. Bean scope determines the lifespan and visibility of beans within the application context. Choosing the right scope is crucial for managing memory usage, ensuring thread safety, and optimizing application performance. The following are scopes defined in the Spring Framework:

- **Singleton:** By default, all beans are Singleton, meaning only one

instance of the bean is created and shared throughout the application context. This scope is appropriate for stateless services and resources, like data sources which are expensive to create and ideally should be reused.

- **Prototype:** Each time the bean is requested, a new instance is created. This is beneficial for stateful beans, ensuring that the state of the bean is not shared. This scope is particularly useful when separate bean instances are required, such as session-specific beans or beans that are not thread-safe.
- **Request:** For each HTTP request, a new instance of the bean is created, but it is only valid in web-aware Spring contexts. This scope assists in handling HTTP requests separately from each other, ensuring the isolation of data.
- **Session:** A new instance of the bean is created for each HTTP session and is only valid in web-aware Spring contexts. This scope is beneficial for storing user-specific data that needs to persist within the lifetime of a session across multiple HTTP requests.
- **Application:** A single instance of the bean is created for the entire web application and is only valid in web-aware Spring contexts. This scope is akin to singleton scope but is bound to the lifecycle of a ServletContext.
- **WebSocket:** This is a relatively new addition to the scopes, introduced in the Spring 4.1 release. A new instance of the bean is created for each WebSocket session, making it useful for managing data specific to each WebSocket session.

There are multiple ways supported in the Spring Framework to define the scope of beans. In XML-based configuration, you can use the **scope** attribute or **<bean>** elements to specify the scope as given in the following code segment:

```
1. <bean id="greetingService" class="com.example.GreetingService" scope="singleton"/>
2.
3. <bean id="dataHolder" class="com.example.DataHolder"
4.     scope="prototype"/>
```

In annotation and Java-based approaches, you can use one of the scope-specific annotations such as **@RequestScope**, **@SessionScope**, or

**@ApplicationScope** to annotate your bean classes. Alternatively, you can use generic **@Scope** annotation with the name of the specific scope or use constants from the **ConfigurableBeanFactory** interface to define the bean scope. The following code segments an example of the use of the **@RequestScope** annotation:

```
1. @Component
2. @RequestScope
3. public class RequestBean {
4.
5. }
```

The following code segment defines prototype scope using the generic **@Scope** annotation:

```
1. @Configuration
2. public class AppConfig {
3.
4.     @Bean
5.     @Scope("prototype")
6.     public Service prototypeService() {
7.         return new ServiceImpl();
8.     }
9. }
```

## Introduction to Spring Boot

Sometimes, developers tend to use **Spring** to refer to both the Spring Framework and Spring Boot interchangeably. Although there is no big issue with that, Spring Framework and Spring Boot are not the same. Spring Boot, released in 2014, aims to enable developers to quickly get started, reducing the initial learning curve and accelerating the process of creating production-ready Spring applications. It achieves this by providing opinionated default configuration and auto-configuration capabilities to reduce the amount of boilerplate code and configuration required. Following the convention-over-configuration principle, Spring Boot helps reduce configuration complexity by providing sensible default configuration to cover common use cases. Some of the important Spring Boot features are highlighted below:

- **Opinionated initial configuration:** It provides an opinionated initial configuration based on the convention-over-configuration principle,

which works out of the box with sensible default values, allowing customizations whenever required. Spring Boot also makes sensible assumptions; for instance, it assumes the presence of the main application class in the root package of an application.

- **Starters for dependency management:** It provides starter dependencies tailored to the type of application you are building. These dependencies automatically configure compatible versions of related libraries according to your preferred build tool.
- **Actuator for management and monitoring:** It provides observability endpoints such as health checks, metrics endpoints, and management endpoints to manage an application.
- **Embedded web servers:** They provide self-contained web servers using embedded server options such as Tomcat, Jetty, Undertow, or Netty eliminating the need for configuring and deploying into external web servers. It also simplifies SSL/TLS configurations and allows easy switching between servers. This feature is crucial for developing microservices-style applications. If needed, developers can disable the embedded web server support by changing a single value in the configuration.
- **Classpath-based auto-configuration:** It provides automatic configuration capabilities based on the dependencies available in the classpath, reducing configuration time and complexity.
- **Multiple environment support:** Spring Boot provides externalized configuration, allowing the same application to be run on different environments with different configuration settings. Environment-specific values and configuration can be injected into applications using properties files, YAML files, environment variables, and command-line arguments.
- **Command-line runner:** It provides a command-line runner interface, allowing specific code to be executed when the application starts up. It is useful for performing activities, including initialization tasks such as populating initial data to a database or changing the database schema, data migration, and background tasks such as scheduled jobs.
- **Developer tools:** They provide a set of developer tools to speed up the development cycle. Features include automatic restarts of an application

when it detects code or configuration changes in the classpath, eliminating manual restarts. It also offers live reload support that triggers a browser refresh when a static resource is modified, the ability to disable caching, remote debugging, and the ability to set up global settings for the entire developer environment.

## Spring Boot starters

One of the main reasons for the widespread adoption of the Spring Framework is its ability to coexist and integrate with other frameworks quite easily. Well-designed extension points make the integration with third-party frameworks a simple task. The Spring team focuses on delivering the best possible experience for what it is good at while allowing other third-party framework developers to integrate what they are good at with Spring easily so that Spring users can benefit from the best of both Spring and third-party frameworks.

Due to this reason, there is a very large number of Spring integrations with third-party projects. In a typical enterprise Spring application, you should be able to identify a set of third-party frameworks that work together with Spring. These could include a data access framework, web framework, job scheduling framework, observability framework, or even a simple logging framework.

Having discussed the bright side of the story, there is also a dark side. As a Spring developer, when you bring a couple of third-party frameworks to coexist with the Spring Framework, you face a couple of challenges:

- **Discovering the right set of dependencies:** Unless the third-party framework you want to bring is quite simple and provides excellent and up-to-date developer documentation, it can be a tedious and time-consuming task to discover the required modules from the third-party framework. Build tools such as Maven and Gradle can be excellent companions for discovering transitive dependencies, but this can still be a challenging task when working with third-party frameworks with multiple modules. At least at the beginning of the project, whenever you want to add a feature from a third-party framework, you will end up spending a good amount of time reading the documentation, posting in forums, or trying with trial and error.

- **Incompatible dependencies:** The transitive dependency resolution of build tools sometimes becomes a problem because you may easily end up with a number of incompatible versions of dependencies that cause runtime exceptions due to the nature of Java classloading behavior. A good example is when every Java framework brings Log4J as a dependency, and it is quite common for you to end up with multiple and incompatible Log4J dependencies in your application classpath. This requires a decent amount of time to study the transitive dependencies of the frameworks you used and resolve them manually using the build tool configuration file to remove incompatible dependencies.

To rescue you from the practical nightmares discussed above, Spring Boot introduces a concept called **starters**. Spring Boot starters bundle related, compatible, and proven-to-work sets of dependencies from Spring's own modules, sub-projects, and third-party frameworks. These starters are designed to achieve one or more common use cases. For example, there is a starter called `spring-boot-starter-web` that provides all the necessary dependencies to build Spring MVC web applications and includes an embedded web server as well. Another example is **spring-boot-starter-data-jpa**, which brings all the JPA and Hibernate-related dependencies required to build data-backed applications.

There is a large number of starters available to support common use cases, and you can also define your own starters to be used with your own organizational use cases.

## Spring Boot auto-configuration

Spring Boot auto-configuration tries to automatically configure applications based on the dependencies present on the classpath, reducing the need for explicit configuration. Auto-configuration helps to reduce boilerplate configuration code that developers have to add, ensures consistent and efficient configuration, and provides opinionated defaults. Auto-configuration also provides you the flexibility to customize the configuration whenever required using properties files, YAML files, environment variables, and command-line arguments. Here are a couple of example cases:

- When Spring MVC dependencies are present in your application classpath, Spring Boot assumes the application as a web application and

auto-configures components like DispatcherServlet, ViewResolver, and static resources.

- If the Spring Security dependencies are present in your classpath, Spring Boot tries to auto-configure security components like authentication providers, filters, a user with a generated password, and an auto-rendered login form.
- When HSQLDB or H2 database dependency is found on the classpath, Spring Boot auto-configures an in-memory database for you. This comes in very handy in testing data-backed applications easily.

You can enable auto-configuration by adding **@EnableAutoConfiguration** or **@SpringBootApplication** to one of your configuration classes. If you want to understand how the auto-configuration feature works behind the scenes, you can start your Spring Boot application by adding the **--debug** switch, which would log conditions report to the console. Spring Boot auto-configuration classes are associated with a priority order, which you can control by adding **@AutoConfigureBefore** and **@AutoConfigureAfter** annotations.

Spring Boot auto-configuration is designed to work non-invasively. This means if you have already defined a `DataSource` explicitly within your Spring metadata configuration, then Spring Boot avoids configuring additional or duplicate `DataSource`. If you are not happy with auto-configuration, you can fully disable the feature as well.

## Spring Boot actuators

In modern software engineering practices, observability stands as an important characteristic of any enterprise software system. Put simply, observability denotes the capability to understand the internal state of a running system from an external standpoint, relying solely on the outputs emitted by an application. Typically, these outputs include logging, metrics, and traces. Spring Boot actuators play a crucial role in facilitating production-ready observability features for Spring applications.

Enabling Spring Boot actuators in your applications is straightforward, simply add them as dependencies using your build tool, such as Maven or Gradle. When the actuator dependency is detected in the classpath, Spring

Boot automatically exposes a set of HTTP endpoints to provide necessary data and facilitate application management. You can configure precisely what you want to expose using configuration files. The exposed endpoints can be discovered using the auto-generated discovery page of an application. If Spring Security dependencies are detected in your classpath, Spring Boot exposes these endpoints as protected endpoints. The following are some important features enabled by Spring Boot actuators:

- **System metrics:** It provides system metrics about the application which includes CPU and memory usage, HTTP requests, and more. These metrics are exposed in a format that can be consumed by external tools such as Grafana, Datadog, InfluxDB, and ELK.
- **Health checks:** They provide information about the health status of the internal components of an application, such as databases, caches, messaging systems, and more. There are built-in health indicators available, and you can develop custom health checks for your own components as well. When an application is deployed in Kubernetes, liveness and readiness information exposed by this endpoint can be consumed by Kubernetes livenessProbe and readinessProbe to maintain healthy application containers.
- **Audit events:** They provide audit events of the application in a format that can be consumed by external auditing tools.
- **Application info:** It provides general information important for the maintenance of an application that includes Git properties, build details, environment details, and Java runtime and operating system details.
- **HTTP tracing:** It provides HTTP request tracing support for debugging purposes.
- **Beans inspection:** It provides beans inspection capabilities for debugging purposes.
- **Dynamic configuration:** It provides capabilities to refresh configuration without restarting the applications. This is also important for debugging purposes.
- **Loggers:** It provides information about loggers in the application and allows to modify them.
- **Prometheus:** It provides the capability to expose metrics in a format

that can be scraped by Prometheus.

- **Sessions:** These provide retrieval and deletion of user sessions from the session store of the Spring web application.
- **Shutdown:** It provides the capability to gracefully shut down an application.
- **Threaddump:** It provides the capability to obtain a threaddump from a running application.

## What is new in Spring 6 and Spring Boot 3

Spring Framework 6.0 and Spring Boot 3.0, released in late 2023, embrace the latest innovations of the Java ecosystem and follow the leading trends in the industry. One of the main features introduced in these releases was baseline support for Java 17 language features, enabling Spring developers to utilize new language features introduced in Java 17, which include features like sealed classes and improved switch expressions.

Furthermore, Spring Framework 6.0 and Spring Boot 3.0 are compatible with Jakarta EE 9 and 10 APIs, such as Servlet 6.0 and JPA 3.1. This compatibility enables Spring developers to leverage the latest dependencies, such as Tomcat 10 and marks the migration from the javax namespace to the Jakarta namespace.

In addition, Spring Framework 6.0 introduced the foundation for a new optimization feature called **ahead-of-time (AOT)** compilation support, aimed at improving performance by compiling applications into native code at build time. These releases also provide first-class support for generating GraalVM native images.

## Conclusion

In this chapter, we introduced the Spring Framework and its most important features, such as dependency injection, IoC container, and beans. We also introduced the Spring Boot project and its key features to you.

In the next chapter, we will learn how to set up your Spring Boot development environment and write your first Spring Boot application.

# CHAPTER 2

# Getting Started with Spring Boot

## Introduction

This chapter will walk you through creating your first Spring Boot application. We will begin by covering the setup of your development environment, the utilization of Spring Initializer, and the typical structure of a Spring project. Next, we will look into the **Visual Studio Code (VS Code)** to create your first application, highlighting the Spring and Spring Boot fundamentals introduced in the previous chapter. Additionally, we will study the fundamentals of building, running, and packaging your Spring Boot applications. Finally, we will explore Java build tool alternatives and application packaging choices.

## Structure

The chapter covers the following topics:

- Setting up the developer environment
- Maven vs. Gradle
- Setting up the IDE
- Spring Initializr
- Writing your first Spring Boot application
- Packaging options

## Objectives

By the end of this chapter, you will understand how to set up a VS Code-based Spring Boot development environment and use the Spring Initializr project to kick-start your projects. You will be able to create your first Spring Boot application and understand build tools and packaging options.

## Setting up the developer environment

Before we write our first Spring Boot application, we need to set up the development environment properly. A well-configured development environment saves a lot of time, simplifies the development workflow, helps to write quality and readable code, and also results in building production-ready enterprise applications. It is also essential to make it a habit to set up the development environment, even for the simple examples we discuss here.

Since Spring is a Java framework, installing a **Java Development Kit (JDK)** is essential. As we use Spring Framework 6 and Spring Boot 3, the code samples in this book are based on Java 17. There are several open-source and commercial Java distributions available. The code samples in this book and the accompanying GitHub repository were tested on the OpenJDK 17 distribution. If you have not installed Java in your development environment, now is the time to install the Java 17 distribution.

**TIP: SDKMAN can be a good companion if you frequently switch among different Java versions or distributions. SDKMAN is a command-line tool that allows you to install, manage, and switch between multiple versions of various SDKs, including JDKs. You can directly install SDKMAN on macOS and Linux, while Windows users are required to have the Windows Subsystem for Linux (WSL).**

## Maven vs. Gradle

The JDK ships with basic tools to compile and package Java applications. Although these tools are sufficient for building simple applications, real-world enterprise applications require more sophisticated build tools. These tools can help with tasks like organizing code into modules, versioning, compiling, managing dependencies, running tests, and packaging applications.

Apache Ant, released in 2000, marked the first notable Java build tool. Ant uses XML-based build scripts to define the build process and dependencies declaratively. Apache Ant is less popular among Java developers today. Maven and Gradle are currently the two most popular build tools used by Java developers.

Apache Maven, first released in 2004, gained popularity among developers and remains the most widely used Java build tool today. Maven defines a standard project structure, build lifecycle, and transitive dependency resolution mechanism based on a convention-over-configuration approach, reducing boilerplate configuration steps. It uses an XML file known as the **Project Object Model (POM)** to specify project structure, dependencies, and build lifecycle. Maven automatically downloads declared dependencies from the central repository, called Maven Central, and other repositories, relieving developers from the burden of manually managing dependencies.

Gradle, a modern build tool released in 2012, was developed to address some of the shortcomings of Maven while closely following the success of the Maven project. Unlike Maven, which relies on XML-based configuration, Gradle utilizes Groovy or Kotlin to define project structure and dependencies, resulting in more straightforward and readable build configurations. Gradle also supports incremental builds, rendering it faster than Maven, especially for large projects. Gradle offers full interoperability with Maven repositories, facilitating not only easy migration from Maven to Gradle but also the utilization of the ecosystem built around Maven repositories.

You can use Maven or Gradle to build Spring Boot applications based on your preferences or organizational policies. This book and the accompanying sample projects use Maven as the build tool simply because most Java developers are already familiar with Maven.

## Setting up the IDE

There are a bunch of popular Java IDEs available, including IntelliJ IDEA, Eclipse, Apache NetBeans, and VS Code, facilitating the development, running, and debugging of Java applications. The examples used in this book use VS Code as the IDE due to its lightweight nature and popularity among Java developers. However, you can use any of your favorite Java IDEs to

build these samples.

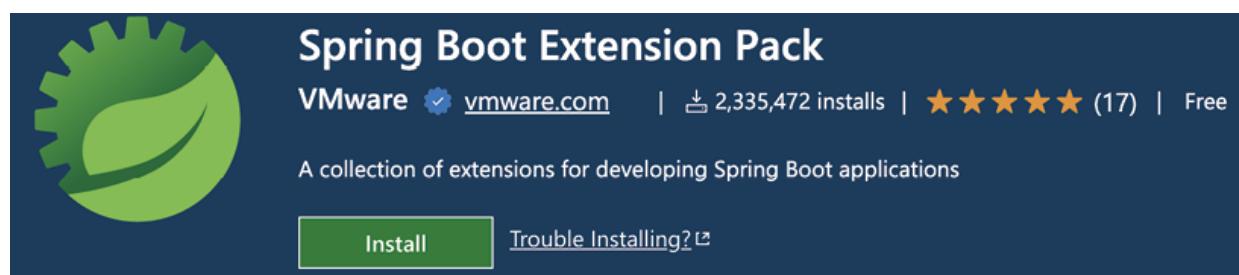
VS Code is available to download freely and supports Windows, macOS, and Linux operating systems. Once you download and install VS Code, we need to install a couple of plugins to simplify the Spring Boot application development process. These plugins are as follows:

- **Extension pack for Java:** This extension pack consists of several extensions and provides Java language features such as code navigation, auto-completion, refactoring, and debugging. Additionally, this pack installs extensions related to Maven and test frameworks such as JUnit and TestNG. (*Figure 2.1*)



*Figure 2.1: VS Code extension pack for Java*

- **Spring Boot extension pack:** This is a collection of extensions helpful for developing and deploying Spring Boot applications. Additionally, this pack enables Spring Initializr and Spring Boot Dashboard, accessible from the VS Code IDE. (*Figure 2.2*)

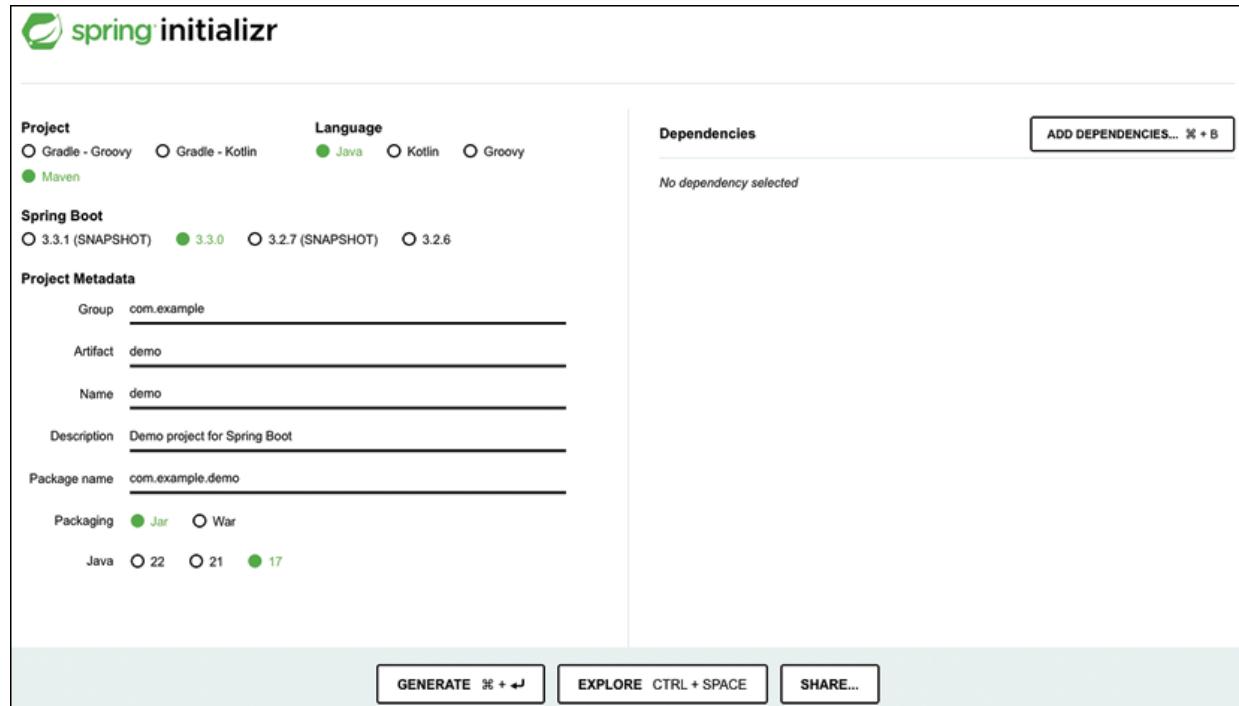


*Figure 2.2: VS Code extension pack for Spring Boot*

## Spring Initializr

Spring Initializr is a web-based tool from the Spring Framework maintainers that can be used to quickly bootstrap Spring Boot applications with default

and working configuration and skeleton code. Some IDEs, such as VS Code, support accessing Spring Initializr within the IDE. The following figure shows you the Spring Initializr with sample project details:



*Figure 2.3: Spring Initializr*

You can access Spring Initializr at <https://start.spring.io>. Once there, you can specify your desired build tool, whether Maven or Gradle, and then determine the metadata for your project, such as group, artifact, and version, as well as the package type, whether JAR or WAR. Most importantly, you can add the dependencies based on the application you intend to develop. Spring Initializr presents these as groups of dependencies required for a specific type of application, saving time by avoiding the need to add required dependencies manually and resolving version conflicts among them. Once you specify all the dependencies, Spring Initializr generates a project you can download as a ZIP file.

## Writing your first Spring Boot application

As we have configured our developer environment for Spring Boot, now let us try to develop our very first Spring Boot application. Following the tradition, we start with the Hello World application.

As the first step, you need to visit Spring Initializr at <https://start.spring.io> and provide the required details as follows. Once you download the generated project as a zip file, you extract it and open it using VS Code.

**Project: Maven**

**Language: Java**

**Spring Boot: Latest stable version ( e.g. – 3.3.0)**

**Project Metadata:**

**Group: com.bpb.hssb.ch2**

**Artifact: helloworld**

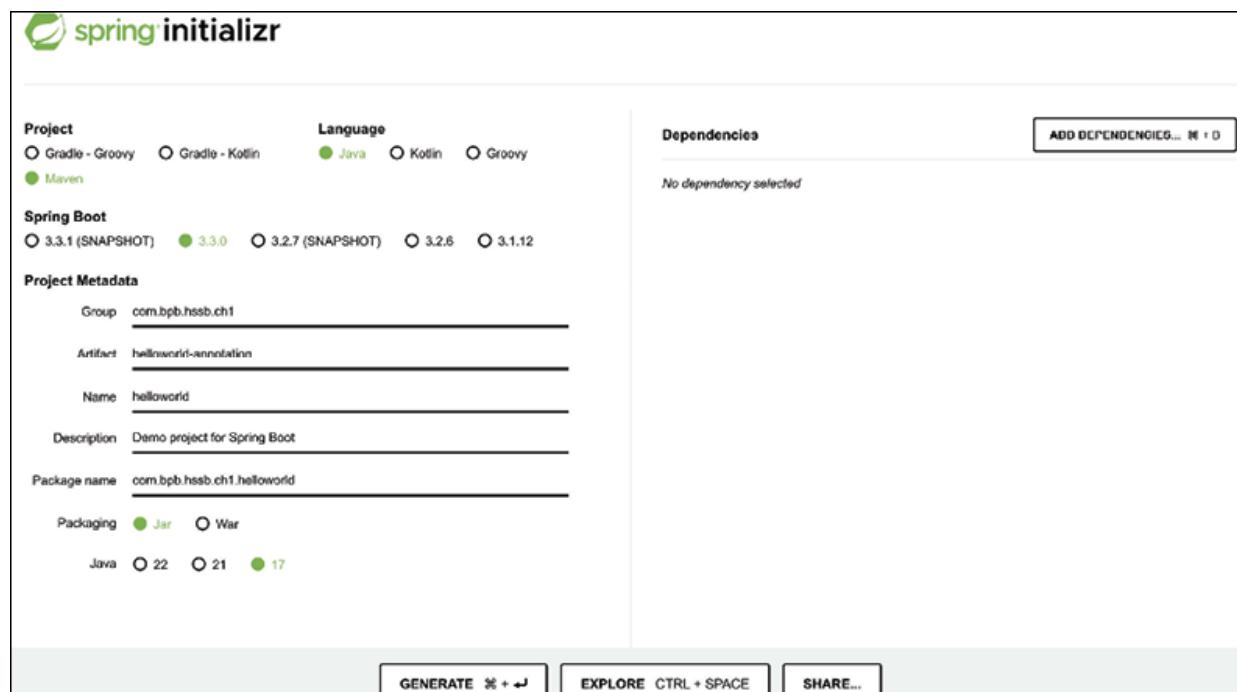
**Name: helloworld**

**Packaging: Jar (Default)**

**Java: 17 (Default)**

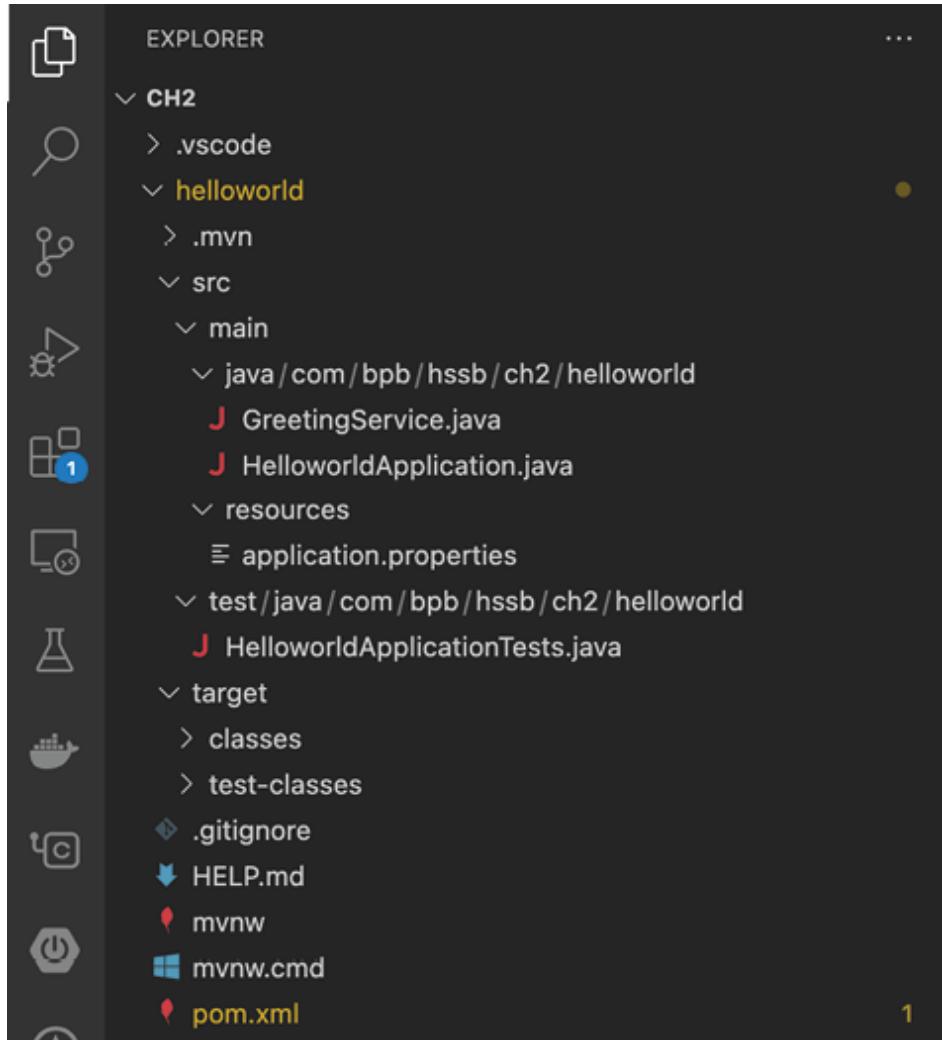
**Dependencies: No dependencies required**

The following figure shows the Spring Initializr with the above project details:



**Figure 2.4: Spring Initializr configuration for helloworld application**

Once you open the project in VS Code, you should see a project structure similar to the following figure:

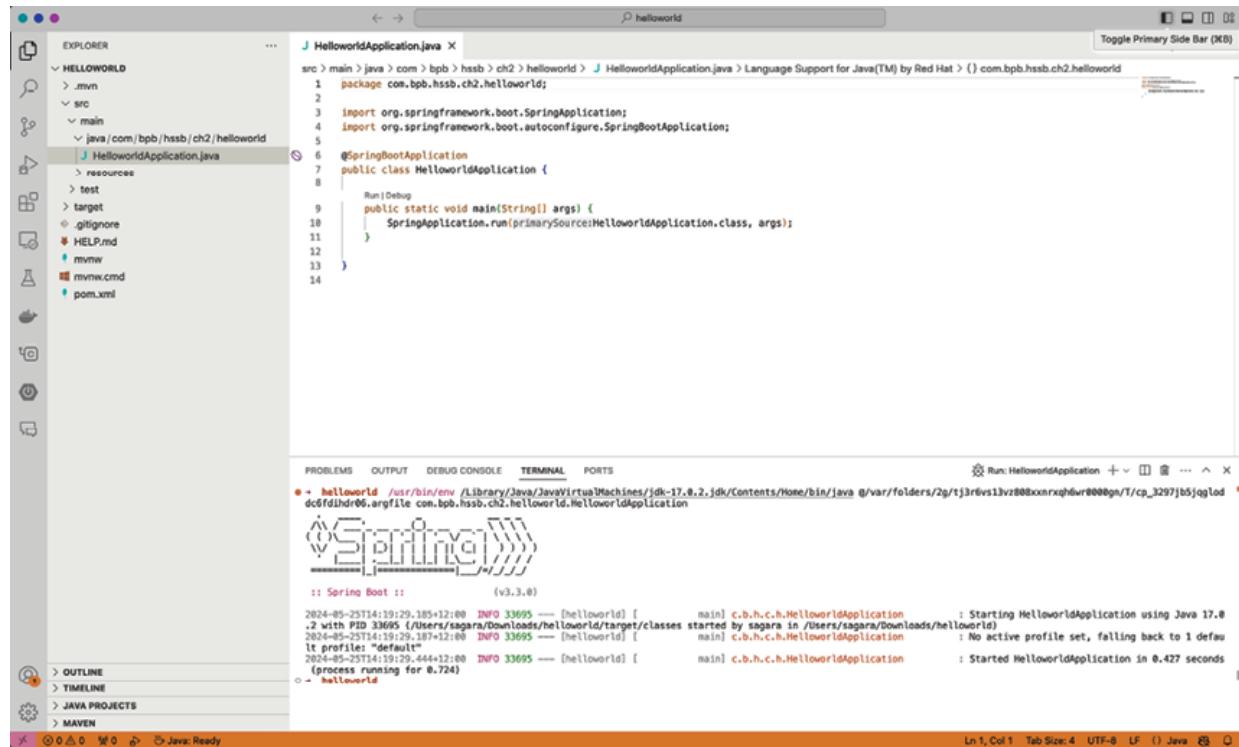


*Figure 2.5: Spring Boot project structure*

As this is our first Spring Boot application, it is worth discussing the default project structure of a Spring Boot application at this point. The project structure is as follows:

- **helloworld** directory is the root of the project.
- **src** directory contains all the source codes and resources of the application.
- **target** directory contains the compiled Java classes and packaged artifacts such as **.jar** or **.war** files.
- These are the Maven wrappers that help run Maven without installing it on your machine.
- The **pom.xml** file is the main configuration file of the project and is used by Maven.

Without going further, let us open the **HelloworldApplication.java** file and run the application using the run button available in your VS Code IDE. If you have configured your environment correctly, you should be able to run the application and see the screen, as shown in the following figure:



*Figure 2.6: running the application using VS Code*

Congratulations on running your first application! We have yet to add any code or change any configuration, but we have a fully working Spring application, all thanks to Spring Initializr and Spring starters. Over the next few steps, we will add some code to the project to achieve our desired results. However, first, let us discuss what has happened and delve a little further into the details of our first project.

First, let us open the **pom.xml** file and discuss what is important for us. If you open the **pom.xml** file using VS Code, you will be able to see the following code:

1. `<?xml version="1.0" encoding="UTF-8"?>`
2. `<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"`
3. `xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">`

```
4. <modelVersion>4.0.0</modelVersion>
5.
6. <parent>
7.   <groupId>org.springframework.boot</groupId>
8.   <artifactId>spring-boot-starter-parent</artifactId>
9.   <version>3.3.0</version>
10.  <relativePath/>
11. </parent>
12.
13. <groupId>com.bpb.hssb.ch2</groupId>
14. <artifactId>helloworld</artifactId>
15. <version>0.0.1-SNAPSHOT</version>
16. <name>helloworld</name>
17. <description>Demo project for Spring Boot</description>
18. <properties>
19.   <java.version>17</java.version>
20. </properties>
21.
22.
23. <dependencies>
24.   <dependency>
25.     <groupId>org.springframework.boot</groupId>
26.     <artifactId>spring-boot-starter</artifactId>
27.   </dependency>
28.
29.   <dependency>
30.     <groupId>org.springframework.boot</groupId>
31.     <artifactId>spring-boot-starter-test</artifactId>
32.     <scope>test</scope>
33.   </dependency>
34. </dependencies>
35.
36. <build>
37.   <plugins>
38.     <plugin>
39.       <groupId>org.springframework.boot</groupId>
40.       <artifactId>spring-boot-maven-plugin</artifactId>
41.     </plugin>
42.   </plugins>
43. </build>
```

44.  
45. </project>

The explanation is as follows:

- In *lines 6 to 11*, the **pom.xml** file defines **spring-boot-starter-parent** as the parent project for our project. By inheriting from **spring-boot-starter-parent**, our project automatically inherits default configurations and dependencies managed by the Spring Boot project, including the definition of compatible versions of core dependencies such as the Spring Framework. It also defines plugins and their versions, such as the Maven Compiler Plugin and Maven Surefire Plugin.
- In *lines 13 to 20*, we can see the project metadata we provided to Spring Initializr at the beginning.
- In *lines 23 to 34*, the dependencies required for our application are included in the form of Spring starters. At this point, we can only see two dependencies: **spring-boot-starter** and **spring-boot-starter-test**. That is more than enough for our first application.
- In *lines 39 to 41*, you can find the declaration of the **spring-boot-maven-plugin**, which we will use to run our application using Maven.

Next, let us open **HelloworldApplication.java** and examine the most critical points related to our discussion. When you open **HelloworldApplication.java** you should be able to see the following Java code:

```
1. package com.bpb.hssb.ch2.helloworld;  
2.  
3. import org.springframework.boot.SpringApplication;  
4. import org.springframework.boot.autoconfigure.SpringBootApplication;  
5.  
6. @SpringBootApplication  
7. public class HelloworldApplication {  
8.  
9.     public static void main(String[] args) {  
10.         SpringApplication.run(HelloworldApplication.class, args);  
11.     }  
12.  
13. }
```

The main noticeable difference in the above code is the presence of the **@SpringBootApplication** annotation. The **@SpringBootApplication** is a

somewhat special annotation; it does not perform any actions itself but acts as a convenience annotation that is equivalent to declaring **@SpringBootConfiguration**, **@EnableAutoConfiguration**, and **@ComponentScan** altogether. Here is a brief description of what each annotation does:

- **@SpringBootConfiguration**: This annotation is a specialization of Spring Framework's **@Configuration** annotation, defined by the Spring Boot project, indicating that the class can be used as a source of bean definition.
- **@EnableAutoConfiguration**: Activates Spring Boot's auto-configuration mechanism, as discussed in the previous chapter, attempting to automatically configure beans based on the dependencies present on the classpath of the application.
- **@ComponentScan**: Enables component scanning, instructing Spring to scan the package and all sub-packages where the annotated class resides to detect Spring beans and construct the **ApplicationContext**.

Here is a part of the definition of **@SpringBootApplication** annotation:

1. `@Target(ElementType.TYPE)`
2. `@Retention(RetentionPolicy.RUNTIME)`
3. `@Documented`
4. `@Inherited`
5. `@SpringBootConfiguration`
6. `@EnableAutoConfiguration`
7. `@ComponentScan(excludeFilters = { @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),`
8.  `@Filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class) })`
9. `public @interface SpringApplication {`

Then, the only other code segment you can see in the **HelloworldApplication** is the following code:

10. `SpringApplication.run(HelloworldApplication.class, args);`

In the above code, the **HelloworldApplication** class with runtime arguments is passed to a static method called **run**, defined in the class called **SpringApplication**, provided by the Spring Boot application. This is considered the entry point for a Spring Boot application. It creates an **ApplicationContext** instance based on the configuration defined in the **HelloworldApplication** class, wires all the beans together, and starts the application. This is what you see when you run the application using VS

Code.

There is another important file for us to look at, which is the **application.properties** file:

1. `spring.application.name=helloworld`

Currently, it does not do anything special other than defining the application's name. However, as we progress with real-world use cases, this file will become more and more important for us. Apart from that, another class called **HelloworldApplicationTests** is present in our project, but we will not concern ourselves with this class until the next chapter.

Although we have a fully working Spring Boot application, we have not added any code or changed any configuration. Now, it is time to get our hands dirty with some coding. First, let us try to print a **Hello World** message in the console, as we would do with any other programming books. The most straightforward approach is to implement the **CommandLineRunner** interface provided by the Spring Framework within the **HelloworldApplication** class, as shown in the following code segment:

```
1. package com.bpb.hssb.ch2.helloworld;
2.
3. import org.springframework.boot.CommandLineRunner;
4. import org.springframework.boot.SpringApplication;
5. import org.springframework.boot.autoconfigure.SpringBootApplication;
6.
7. @SpringBootApplication
8. public class HelloworldApplication implements CommandLineRunner {
9.
10.     public static void main(String[] args) {
11.         SpringApplication.run(HelloworldApplication.class, args);
12.     }
13.
14.     @Override
15.     public void run(String... args) throws Exception {
16.         System.out.println("Hello World");
17.     }
18.
19. }
```

In the above code, we have implemented the run method defined in the **CommandLineRunner** interface and added a simple statement to print the

**Hello World** message in the console. If you rerun the application, you will be able to see the **Hello World** message printed in the console, as shown in the following figure:

```
cd /Users/sagara/Dev/gdrive/Book-Spring6/code/ch2 ; /usr/bin/env /Library/Java/JavaVirtualMachines/jdk-17.0.2.jdk/Contents/Home/bin/java @var/folders/2g/tj3r6vs13vz888xxnrq6wr0000gn/T/cp_b6go7akx8axz1mg2beeh98emy.argfile com.bpb.hssb.ch2.helloworld.HelloworldApplication
o < ch2 git:(master) ✘
o < ch2 git:(master) ✘ cd /Users/sagara/Dev/gdrive/Book-Spring6/code/ch2 ; /usr/bin/env /Library/Java/JavaVirtualMachines/jdk-17.0.2.jdk/Contents/Home/bin/java @var/folders/2g/tj3r6vs13vz888xxnrq6wr0000gn/T/cp_b6go7akx8axz1mg2beeh98emy.argfile com.bpb.hssb.ch2.helloworld.HelloworldApplication


```

```
;; Spring Boot :: (v3.3.0)

2024-05-24T13:53:22.632+12:00  INFO 8558 --- [helloworld] [           main] c.b.h.c.h.HelloworldApplication      : Starting HelloworldApplication using Java
17.0.2 with PID 8558 (/Users/sagara/Dev/gdrive/Book-Spring6/code/ch2/helloworld/target/classes started by sagara in /Users/sagara/Dev/gdrive/Book-Spring6/code/
ch2)
2024-05-24T13:53:22.635+12:00  INFO 8558 --- [helloworld] [           main] c.b.h.c.h.HelloworldApplication      : No active profile set, falling back to 1
default profile: "default"
2024-05-24T13:53:22.925+12:00  INFO 8558 --- [helloworld] [           main] c.b.h.c.h.HelloworldApplication      : Started HelloworldApplication in 0.488 se
Hello World
```

*Figure 2.7: Output running the helloworld application*

Now, let us try to modify our application to put the concept of **dependency injection (DI)** that we discussed in the previous chapter into action. Let us assume we want to enhance our message a little bit by adding a greeting depending on the time of the day. For that, we define the **GreetingService** class, which defines a method called `greet` to return a greeting depending on the time of the day. Add a new class using VS Code and add the following code:

```
1. package com.bpb.hssb.ch2.helloworld;
2.
3. import java.time.LocalTime;
4.
5. import org.springframework.stereotype.Component;
6.
7. public class GreetingService {
8.
9.     public String greet() {
10.         LocalTime now = LocalTime.now();
11.         int currentHour = now.getHour();
12.         String msg = "Hello world";
13.         if (currentHour < 12) {
14.             return msg + ", it's a wonderful morning!";
15.         } else {
16.             return msg + ", it's a wonderful afternoon!";
17.         }
18.     }
}
```

```
19. }
```

The next task would be to modify the **HelloworldApplication** class to declare **GreetingService** as a dependency and use the greet method to improve the message printed in the console. You can modify the **HelloworldApplication** class as follows:

```
1. package com.bpb.hssb.ch2.helloworld;
2.
3. import org.springframework.beans.factory.annotation.Autowired;
4. import org.springframework.boot.CommandLineRunner;
5. import org.springframework.boot.SpringApplication;
6. import org.springframework.boot.autoconfigure.SpringBootApplication;
7.
8. @SpringBootApplication
9. public class HelloworldApplication implements CommandLineRunner {
10.
11.     private GreetingService greetingService;
12.
13.     @Autowired
14.     public void setGreetingService(GreetingService greetingService) {
15.         this.greetingService = greetingService;
16.     }
17.
18.     public static void main(String[] args) {
19.         SpringApplication.run(HelloworldApplication.class, args);
20.     }
21.
22.     @Override
23.     public void run(String... args) throws Exception {
24.         System.out.println(greetingService.greet());
25.     }
26. }
```

In the modified **HelloworldApplication** class, we have defined **GreetingService** as a field and introduced a setter method that takes **GreetingService** as a parameter and sets it into the above-mentioned field. The most important modification we have made in this class is annotating the setter method with the Spring **@Autowired** annotation. When a setter method is annotated with the **@Autowired** annotation, Spring will try to inject the constructor arguments with matching beans from the **ApplicationContext**. In simple terms, we are declaring dependencies

expected by a particular class.

Now, we have modified the **HelloworldApplication** class, declared the dependencies, and added **GreetingService**. So, without any further ado, let us run our application again.

If you have followed exactly so far, you will encounter the following error message instead of the **Hello World** message.

1. \*\*\*\*
2. APPLICATION FAILED TO START
3. \*\*\*\*
- 4.
5. Description:
- 6.
7. Parameter 0 of constructor in com.bpb.hssb.ch2.helloworld.HelloworldApplication required a bean of type 'com.bpb.hssb.ch2.helloworld.GreetingService' that could not be found.
- 8.
9. Action:
- 10.
11. Consider defining a bean of type 'com.bpb.hssb.ch2.helloworld.GreetingService' in your configuration.

This should be the expected outcome. Spring is also being very helpful by providing a useful error message for us. According to the error message, Spring cannot find a suitable **GreetingService** typed bean to be injected as a constructor parameter. The error message also provides a useful hint suggesting that we define a bean of the type **GreetingService** in our Spring configuration. When Spring auto-scans the application classpath, it cannot detect a bean to inject into **HelloworldApplication**.

To explain it further, we have correctly annotated **HelloworldApplication** using the **@Autowired** annotation, so Spring knows that **HelloworldApplication** expects a **GreetingService** as a dependency. However, that is only one part of the puzzle. The other part is that we need to provide a suitable bean to Spring using one of the supported configuration methods, and that is what we have to do now.

As discussed in the previous chapter, we can use XML, annotations, or Java configuration to define the Spring metadata configuration. In this sample, let us use the annotation approach as it is the simplest. Let us add the **@Component** annotation to **GreetingService**. When Spring scans the

classpath during application start-up, it detects classes annotated with **@Component** and registers them as beans in the **ApplicationContext**, so this should solve our problem. Spring provides particular types of annotations called stereotype annotations, which can automatically create Spring beans when the annotated Java classes are detected during the auto-scan process. The primary stereotype annotation is **@Component**, and several role-specific stereotype annotations are available, such as **@Service**, **@Repository**, and **@Controller**. We can add **@Component** annotation to the **GreetingService** class as shown in the following code:

```
1. package com.bpb.hssb.ch2.helloworld;
2.
3. import java.time.LocalTime;
4.
5. import org.springframework.stereotype.Component;
6.
7. @Component
8. public class GreetingService {
9.
10.    public String greet() {
11.        LocalTime now = LocalTime.now();
12.        int currentHour = now.getHour();
13.        String msg = "Hello world";
14.        if (currentHour < 12) {
15.            return msg + ", it's a wonderful morning!";
16.        } else {
17.            return msg + ", it's a wonderful afternoon!";
18.        }
19.    }
20. }
```

If you run the application now, you should be able to see a greeting message, either **wonderful morning** or **wonderful afternoon**, at the end.

In the above sample application, we used annotations to define the configuration. However, we can also use Java or XML to provide the configuration. Since XML-based configuration is no longer popular, let us try to modify our application to use Java to define the configuration.

First, we have to remove the **@Component** annotation from the **GreetingService** class, as follows:

```
1. package com.bpb.hssb.ch2.helloworld;
```

```

2.
3. import java.time.LocalTime;
4.
5. import org.springframework.stereotype.Component;
6.
7. public class GreetingService {
8.
9.     public String greet() {
10.         LocalTime now = LocalTime.now();
11.         int currentHour = now.getHour();
12.         String msg = "Hello world";
13.         if (currentHour < 12) {
14.             return msg + ", it's a wonderful morning!";
15.         } else {
16.             return msg + ", it's a wonderful afternoon!";
17.         }
18.     }
19. }

```

Then, we need to add a new class called **HelloworldConfiguration**:

```

1. package com.bpb.hssb.ch2.helloworld;
2.
3. import org.springframework.context.annotation.Bean;
4. import org.springframework.context.annotation.Configuration;
5.
6. @Configuration
7. public class HelloworldConfiguration {
8.
9.     @Bean
10.    public GreetingService greetingService() {
11.        return new GreetingService();
12.    }
13.
14. }

```

As you can see, we have annotated the **HelloworldConfiguration** class with the **@Configuration** annotation. This annotation tells the Spring container that this class contains one or more methods that define beans to be managed by the container and can be considered equivalent to XML-based Spring configuration.

The **greetingService** method is annotated with the **@Bean** annotation. This

annotation indicates to the Spring container that this method will return an object that should be registered as a bean in the Spring application context. In our case, a bean named **greetingService** will be registered with the **ApplicationContext**. When required, you can change the default name of a bean by specifying the name attribute of the **@Bean** annotation. For example, **@Bean(name="bestGreetingService")** would register a bean with the name **bestGreetingService**.

In addition to the **@Bean** annotation, other annotations related to resource loadings, such as **@Import**, **@ImportResource**, and **@PropertySource**, can also be used in configuration classes annotated with the **@Configuration** annotation.

As the final step, we can remove both the **@SpringBootApplication** annotation and **@Autowired** from the **HelloworldApplication** class and slightly modify it as follows:

```
1. package com.bpb.hssb.ch2.helloworld;
2.
3. import org.springframework.boot.CommandLineRunner;
4. import org.springframework.boot.SpringApplication;
5. import org.springframework.boot.SpringApplication;
6. import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
7. import org.springframework.context.annotation.Import;
8.
9. @SpringBootConfiguration
10. @Import({ HelloworldConfiguration.class })
11. @EnableAutoConfiguration
12. public class HelloworldApplication implements CommandLineRunner {
13.
14.     private GreetingService greetingService;
15.
16.     public HelloworldApplication(GreetingService greetingService) {
17.         this.greetingService = greetingService;
18.     }
19.
20.     public static void main(String[] args) {
21.         SpringApplication.run(HelloworldApplication.class, args);
22.     }
23.
24.     @Override
```

```
25. public void run(String... args) throws Exception {  
26.     System.out.println(greetingService.greet());  
27. }  
28. }
```

Let us look at what exactly we have done in the above code:

- First, we have removed the **@SpringBootApplication** annotation from the **HelloworldApplication** class. This means that Spring no longer considers this class a source of bean definitions and does not activate component scanning and auto-configuration features altogether. If required, we need to specify appropriate annotations to enable each specific feature explicitly.
- We have added **@SpringBootConfiguration** annotation to indicate that the annotated class provides Spring configuration. This can be used as an alternative for **@SpringBootApplicationannotation** annotation.
- In the next line, we have included the **@Import** annotation, which allows configuration to be imported from another class. In our case, we have passed the **HelloworldConfiguration** class as an attribute. This tells the Spring container to explicitly load the **HelloworldConfiguration** class to load the bean definitions.
- We have also added the **@EnableAutoConfiguration** annotation. In our example, there is no special use for this annotation, but in most practical cases, it is beneficial to enable the Spring Boot auto-configuration feature by adding this annotation.
- The other noticeable change that we have made is that we introduced a constructor, but we have not annotated it with **@Autowired** or any other annotation. However, if you run the application, it will work perfectly fine. In this case, Spring automatically injects suitable bean dependencies into the constructor when creating the beans. This is known as constructor injection in Spring and is highly recommended over any other injection approaches. Whenever possible, you should use constructor injection in your applications.

If you run the modified application, you should be able to see the same result again, but this time using the Java configuration approach.

At this stage, you have successfully developed your first Spring Boot application, configured it using both annotations and Java configuration, and

put DI into action as well. One important point to highlight at this stage is that Spring dependency injection is not limited to beans or Java objects; instead, you can inject any resource supported by Spring's resource abstraction. One simple example would be maintaining environment-specific properties such as database URLs and credentials out of your application code and injecting them at runtime.

To illustrate property injection further, let us modify our **helloworld** application again. Let us say instead of printing **Hello World** all the time, you want to change it to something like **Hello Earth** or a specific country name.

Let us start by changing the **GreetingService** class:

```
1. package com.bpb.hssb.ch2.helloworld;
2.
3. import java.time.LocalTime;
4.
5. public class GreetingService {
6.
7.     private String subject;
8.
9.     public GreetingService(String subject) {
10.         this.subject = subject;
11.     }
12.
13.     public String greet() {
14.         String msg = "Hello " + subject;
15.         LocalTime now = LocalTime.now();
16.         int currentHour = now.getHour();
17.         if (currentHour < 12) {
18.             return msg + ", it's a wonderful morning!";
19.         } else {
20.             return msg + ", it's a wonderful afternoon!";
21.         }
22.     }
23. }
```

In the above code, we have defined a new field called `subject` and introduced a class constructor to receive a value for this field. Here, we use the constructor injection approach discussed earlier to inject a property during the initialization of the **GreetingService** bean. The next step would be to

modify the **HelloworldConfiguration** class to facilitate the property injection:

```
1. package com.bpb.hssb.ch2.helloworld;
2.
3. import org.springframework.beans.factory.annotation.Value;
4. import org.springframework.context.annotation.Bean;
5. import org.springframework.context.annotation.Configuration;
6.
7. @Configuration
8. public class HelloworldConfiguration {
9.
10.    @Bean
11.    public GreetingService greetingService(@Value("${helloworld.application.subject}") String subject) {
12.        return new GreetingService(subject);
13.    }
14.
15. }
```

In the above code, we have introduced the **@Value** annotation and annotated the method argument with it. This annotation can be used to inject property values from external sources, such as property files or environment variables, into fields, constructor arguments, or method arguments of beans. In our case, it tries to inject the value of a property called **helloworld.application.subject** into the **greetingService** bean. Based on your preferences, you can come up with your own naming convention to name these properties.

The final step is to provide this value from an external source. For that, you can define your own property file, or we can use the **application.properties** file available with our Spring Boot application. As the default behavior, Spring Boot automatically loads the **application.properties** file, and we do not need to include any specific configuration. For simplicity, let us use the default **application.properties**. You can open this file from VS Code and define the value of **helloworld.application.subject** as follows:

```
1. spring.application.name=helloworld
2. helloworld.application.subject=world
```

You do not need to change anything in **HelloworldApplication** as this only affects **GreetingService**. If you run the application, you will see the same

result again. You can also run the application again after changing the value of **helloworld.application.subject** to something else.

At this stage, we have covered all the topics we intended to discuss in this chapter. The overly complicated **helloworld** samples we have developed throughout this chapter are helpful for grasping the core concepts of Spring and Spring Boot. However, they do not effectively illustrate the power and simplicity provided by Spring. So, let us create a simple RESTful service to understand how easy it is to develop production-ready applications using Spring.

As we did at the beginning of this chapter, visit the Spring Initializr at <https://start.spring.io> and provide the required details as given below. Once you download the generated project as a zip file, extract it and open it using VS Code. Other than changing the name of the application, we need to add Spring Web as a dependency this time. Last time we did not add any specific dependency.

**Project: Maven**

**Language: Java**

**Spring Boot: Latest stable version ( e.g. – 3.3.0)**

**Project Metadata:**

**Group : com.bpb.hssb.ch2**

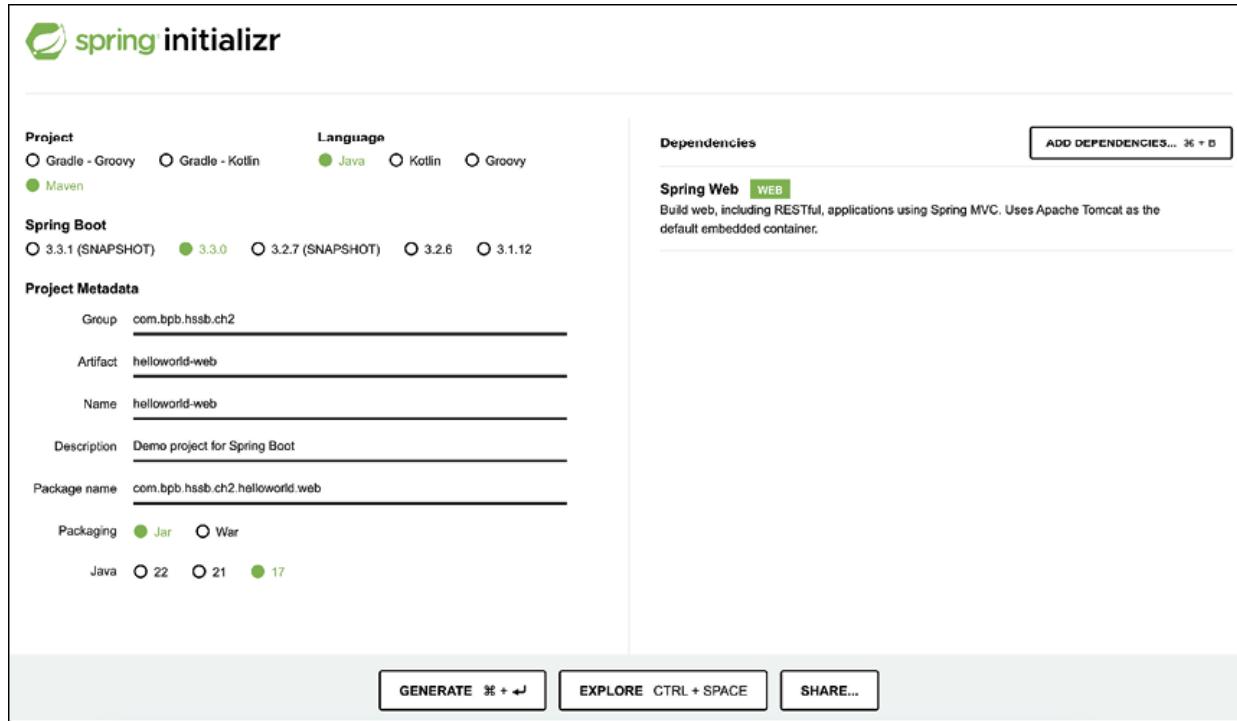
**Artifact: helloworld-web**

**Name: helloworld-web**

**Packaging: Jar (Default)**

**Java: 17 (Default)**

**Dependencies: Spring Web**



*Figure 2.8: Spring Initializr for web project*

Let us open the downloaded project in the VS Code IDE and copy the **GreetingService** from our previous samples into this project:

```

1. package com.bpb.hssb.ch2.helloworld.web;
2.
3. import java.time.LocalTime;
4.
5. import org.springframework.stereotype.Component;
6.
7. @Component
8. public class GreetingService {
9.
10.    public String greet() {
11.        LocalTime now = LocalTime.now();
12.        int currentHour = now.getHour();
13.        String msg = "Hello world";
14.        if (currentHour < 12) {
15.            return msg + ", it's a wonderful morning!";
16.        } else {
17.            return msg + ", it's a wonderful afternoon!";
18.        }
19.    }

```

20. }

Next, we will add a new class called **HelloController** as follows:

```
1. package com.bpb.hssb.ch2.helloworld.web;
2.
3. import org.springframework.beans.factory.annotation.Autowired;
4. import org.springframework.web.bind.annotation.GetMapping;
5. import org.springframework.web.bind.annotation.RestController;
6.
7. @RestController
8. public class HelloController {
9.
10.     private GreetingService greetingService;
11.
12.     @Autowired
13.     public void setGreetingService(GreetingService greetingService) {
14.         this.greetingService = greetingService;
15.     }
16.
17.     @GetMapping("/hello")
18.     public String hello() {
19.         return greetingService.greet();
20.     }
21. }
```

In the code above, we have introduced a REST controller class and annotated it with **@RestController**, indicating that this class will serve as a controller for managing incoming REST API requests. The hello method is also annotated with **@GetMapping**, which is utilized to map HTTP GET requests to specific handler methods within a REST controller class. In our example, the hello method handles HTTP GET requests with the **/hello** path. We have a dedicated chapter for RESTful service development.

If you open the **HelloworldWebApplication** class, you will notice that it contains the same initial code as our first application:

```
1. package com.bpb.hssb.ch2.helloworld.web;
2.
3. import org.springframework.boot.SpringApplication;
4. import org.springframework.boot.autoconfigure.SpringBootApplication;
5.
6. @SpringBootApplication
7. public class HelloworldWebApplication {
```

```

8.
9. public static void main(String[] args) {
10. SpringApplication.run(HelloworldWebApplication.class, args);
11. }
12.
13. }

```

Now, let us try to run the application. You will see something similar to the following figure in your console:



```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

o + helloworld-web git:(master) x /usr/bin/env /Library/Java/JavaVirtualMachines/jdk-17.0.2.jdk/Contents/Home/bin/java @var/folders/2g/tj3r6vs13vz808xxnrqh6wr000
@gn/T/cp_22ig88gcola6wcjtbhbc45eqq.argfile com.bpb.hssb.ch2.helloworld.web.HelloworldWebApplication

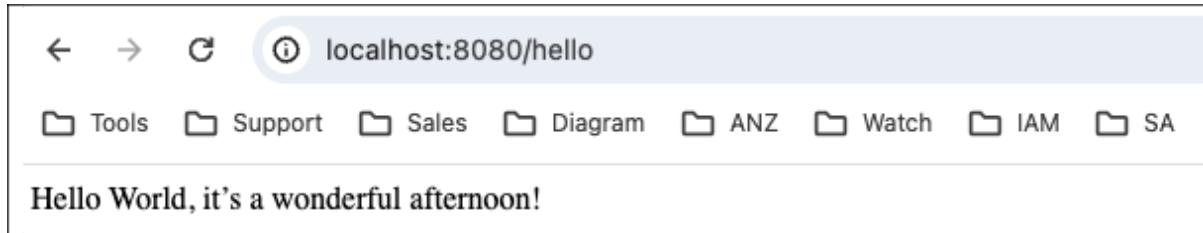
:: Spring Boot ::          (v3.3.0)

2024-05-26T12:42:01.786+12:00 INFO 97492 --- [helloworld-web] [main] c.b.h.c.h.web.HelloworldWebApplication : Starting HelloworldWebApplication using
Java 17.0.2 with PID 97492 (/Users/sagara/Dev/gdrive/Book-Spring6/code/ch2/helloworld-web/target/classes started by sagara in /Users/sagara/Dev/gdrive/Book-Spring
6/code/ch2/helloworld-web)
2024-05-26T12:42:01.788+12:00 INFO 97492 --- [helloworld-web] [main] c.b.h.c.h.web.HelloworldWebApplication : No active profile set, falling back to
1 default profile: "default"
2024-05-26T12:42:02.292+12:00 INFO 97492 --- [helloworld-web] [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port 8080 (http
)
2024-05-26T12:42:02.301+12:00 INFO 97492 --- [helloworld-web] [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2024-05-26T12:42:02.301+12:00 INFO 97492 --- [helloworld-web] [main] o.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat
/10.1.24]
2024-05-26T12:42:02.337+12:00 INFO 97492 --- [helloworld-web] [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplica
tionContext
2024-05-26T12:42:02.338+12:00 INFO 97492 --- [helloworld-web] [main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initializ
ion completed in 519 ms
2024-05-26T12:42:02.525+12:00 INFO 97492 --- [helloworld-web] [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 8080 (http) with
context path '/'
2024-05-26T12:42:02.531+12:00 INFO 97492 --- [helloworld-web] [main] c.b.h.c.h.web.HelloworldWebApplication : Started HelloworldWebApplication in 0.9
73 seconds (process running for 1.23)

```

*Figure 2.9: Running helloworld-web application*

Although the application classes in the **helloworld** and the current **helloworld-web** projects contain identical code, there is a notable difference during the application's runtimes. Unlike the **helloworld** sample, here you can notice that Spring has started an embedded Tomcat server and is awaiting incoming requests. This serves as a good example of Spring Boot's auto-configuration feature being in action. You may recall that, in contrast to the **helloworld** example, we included Spring-web as a dependency when generating the application through Spring Initializr. In this case, Spring Boot has detected the Spring-web dependencies in the application classpath and has reasonably assumed that our application requires a web server to run, assuming that it should be an embedded server, given that we selected Jar as the packaging option. Additionally, Spring Boot has made some assumptions regarding default values, such as using Tomcat as our web server and serving requests on port 8080. If you enter the URL <http://localhost:8080/hello> in your web browser, you will be greeted with the following screen displaying the output message:



*Figure 2.10: Output from helloworld-web application*

While Spring Boot has made several reasonable, opinionated assumptions, it also simplifies the process of modifying configurations as needed. Let us try to modify the default listener port of the Tomcat server from 8080 to 6060. Open the **application.properties** file in your project and append the following line:

```
1. server.port=6060
```

If you rerun the application, you will not receive any results from the **http://localhost:8080/hello** URL. Instead, you must use the **http://localhost:6060/hello** URL, as the embedded Tomcat server is now listening on port number 6060.

So far, we have been using VS Code IDE to run our applications; now, let us look at how we can run the applications using Maven. You can run your applications for testing purposes using the following command.

In Windows:

```
1. mvnw.cmd spring-boot:run
```

In Linux and macOS:

```
1. ./mvnw spring-boot:run
```

## Packaging options

So far, we have been using the VS Code IDE to run our applications. Now, let us explore how we can run the applications using Maven. For testing purposes, you can run your applications with the following command:

- **Executable JAR:** Spring Boot allows you to package your application as an executable JAR file, which can be run using the `java -jar` command. This is one of the most convenient ways to deploy Spring Boot applications, especially if using a microservices-based architecture. It is also ideally suited for cloud deployments and containerized environments like Docker. Sometimes, these executable

JARs are referred to as *fat jars* or *uber jars* because they are self-contained and include all dependencies and resources in a single deployment unit. Executable JARs also support property injection via environment and system variables.

- **Traditional WARs:** Spring Boot also supports packaging your application as a traditional **Web Application Archive (WAR)** file, which can be deployed to web servers like Tomcat or Jetty that usually act as a central application server. This approach is suitable for traditional monolithic application architectures, where you deploy and manage multiple WAR files on the same application server. In this approach, dependencies and resources are included inside the WAR file, but the internal structure and packaging mechanisms differ from those of executable JARs.

You can use the following Maven command to package your application as specified in the Maven POM file:

```
1. ./mvnw package
```

The above Maven command will package your application as an executable JAR file and place it in the project's target directory. You can then directly execute your application using the following Java command. In this example, we use the RESTful application created in this chapter:

```
1. java -jar target/helloworld-web-0.0.1-SNAPSHOT.jar
```

There are several options available for changing configuration parameters from outside the application, including Spring profiles, specifying a custom configuration file, and passing them as environment variables. For instance, you can use the following command to change the listening port of the embedded Tomcat server to port number 9090.

```
1. java -Dserver.port=9090 -jar target/helloworld-web-0.0.1-SNAPSHOT.jar
```

## Conclusion

In the next chapter, we will explore Spring test-driven development, core security concepts to secure your applications, and observability concepts. In the next chapter, we will explore three crucial aspects of modern software development: TDD with Spring, essential security concepts, and observability. TDD will guide us through writing tests before implementing code, ensuring functionality and maintainability from the start. This includes

unit tests with tools like JUnit and integration tests to verify system behavior. We will also dive into key security concepts using Spring Security, covering topics like authentication, authorization, and protection against common vulnerabilities. Lastly, we will discuss observability principles, focusing on how logs, metrics, and traces help monitor applications, diagnose issues, and ensure smooth performance in real-world environments.

*OceanofPDF.com*

# CHAPTER 3

## Spring Essentials for Enterprise Applications

### Introduction

This chapter will introduce three important concepts required in building production-ready enterprise applications. We will start by introducing **test-driven development (TDD)** as a development methodology, highlighting its role in improving robustness and testability and how Spring Boot streamlines the TDD process. Subsequently, we will also write your first test case using the Spring test module and discuss how mocking technologies can be used. Moving forward, we will discuss security as another critical concept, introducing the Spring Security project and its features. Lastly, we will introduce the concept of observability and the golden triangle of observability: metrics, logs, and traces. We will also explore the configuration of logs, metrics, and traces within your Spring applications utilizing logging frameworks and Micrometer.

### Structure

The chapter covers the following topics:

- Building enterprise applications
- Test-driven development
- Writing your first Spring Boot test

- Securing Spring Boot applications
- Observability of an application
- Spring Boot Actuators
- Spring Boot metrics
- Spring Boot logging
- Spring Boot tracing

## Objectives

By the end of this chapter, you will understand TDD and how to use the Spring test module and mocking capabilities to write test cases. You will also understand basic security concepts and will apply security to an application that we created in the previous chapter. Finally, you will learn how to use Spring Boot Actuators and other observability features.

## Building enterprise applications

In the first chapter, we briefly discussed that enterprise applications are not just another type of application; they possess specific quality characteristics that a software system should have, such as scalability, reliability, security, customizability, extensibility, and observability, among others. This chapter will discuss three fundamental quality characteristics: testability, observability, and security.

## Testability

The testability of an application is an attribute that determines how easily the software can be tested and how readily faults or defects can be discovered through testing. There is a set of principles defined to design testable applications, which include:

- **Modularity:** Breaking down an application into smaller, independent modules with well-defined responsibilities allows each module to be tested separately from the others. These modules should be loosely coupled and highly cohesive. Loose coupling means that modules have minimal dependencies on each other, so changes in one module have little or no impact on others. High cohesion means that the components

within a module belong together and focus on a single responsibility. Highly cohesive modules make it easier to write comprehensive test cases that cover the specific capability handled by each module.

- **Interfaces:** There should be well-defined, clear interfaces for the components of an application, and components should interact with one another only through these interfaces. This allows for easier testing of individual components and ensures that changes in one module do not unnecessarily affect others. Additionally, implementing test hooks and entry points enables testers to control and observe the internal state of a component.
- **Test-driven development (TDD):** Adopting TDD ensures you have a comprehensive set of test cases even before implementing your business logic. We will discuss TDD in detail in the next section.
- **Observability:** Implementing proper observability features, such as logging and debugging, helps you detect and diagnose issues more easily. There is a separate section later in this chapter to discuss the observability features of Spring Boot.
- **Automated testing:** A comprehensive set of test cases provides little value unless they are integrated into a continuous integration and automated testing (CI/CD) process and run regularly, especially after modifications. Build tools like Maven and Gradle make it easy to run test cases as part of the build process, making it easy to integrate into your application building and releasing process.

## Test-driven development

**Test-driven development (TDD)** gained popularity as part of **Extreme Programming (XP)** in the late 1990s. The fundamental idea behind TDD is to write tests before writing the actual code. This approach compels you to think about all possible test scenarios, functional and non-functional, before implementing the code. By doing so, it enhances test coverage and builds confidence in the code.

The TDD process begins with writing a unit test for the desired functionality. This test should be specific and focus on a single aspect of the functionality. Running the test initially results in failure, ensuring that the specific

functionality has not yet been implemented. The next step involves implementing the actual functionality and iteratively refining the code until all the test cases pass. This confirms that all intended functionality has been covered.

Whenever modifications are made to an application, following TDD involves first implementing failing test cases for the changes. Then, the modifications are made to ensure that the tests no longer fail. This iterative process helps maintain code quality, serves as documentation for the code and its use cases, and aids in preventing regression issues when modifying existing code, whether written by others or yourself over time.

Although we have been using test cases as a generic term so far, in practice, there are several different types of test cases that you have to write, and each type serves a specific purpose. Let us discuss some of the most important types of tests as follows:

- **Unit test:** Unit tests are important for ensuring code quality, specifically targeting the functionality of individual components or units of code—usually a single interface or method. These tests are typically written and executed using test frameworks such as JUnit or TestNG, both widely used in Spring Boot applications. To maintain isolation from other application components, unit tests require mocking of dependencies. In Spring Boot, you can utilize mock object features to implement the necessary mocking capabilities for your unit tests.
- **Integration test:** Integration tests serve the purpose of verifying that various components and modules function together as intended, demonstrating the data flow and communication within an application. In many cases, the same testing frameworks used for unit tests can also be employed for integration tests.
- **Performance test:** Performance tests evaluate how fast, responsive, and stable an application performs under different conditions. Tools like JMeter are commonly used to execute and automate these tests.
- **Functional test:** Functional tests aim to ensure that software functions as intended based on requirements. They validate the complete workflow of an application, simulating real user scenarios from start to finish. Frameworks like Selenium and Cypress are commonly utilized for implementing functional tests.

Before looking at Spring Boot test capabilities, let us explore a set of principles related to writing unit tests. These principles are best explained using the concept **F.I.R.S.T.**, introduced by software developer and trainer Sergey Kargopolov in his blog:

- **Fast:** The *F* in the F.I.R.S.T principle stands for *Fast*. Unit tests are relatively small code segments that test the functionality of individual components independently from other dependencies like database or network calls. Therefore, unit tests should be faster than other types of tests.
- **Independent:** The *I* in the F.I.R.S.T principle stands for *Independent*. Unit tests must operate independently of each other and should not rely on the outcome of another unit test. This allows us to run unit tests simultaneously or in random order, optimizing test execution time and resource usage. Writing unit tests that require specific execution orders is risky and should be avoided.
- **Repeatable:** The *R* in the F.I.R.S.T principle stands for *Repeatable*. Running a unit test multiple times should consistently produce the same results, regardless of variations in execution environments such as hardware specifications, operating systems, or JDK versions used to run the test.
- **Self-validating:** The next letter stands for *Self-validating*. Unit tests should autonomously validate their results without requiring manual intervention, offering comprehensive test outcomes. Frameworks like JUnit or TestNG include features that support automatic validation and reporting capabilities.
- **Thorough:** The final letter stands for *Thorough*. This principle emphasizes that developers should consider and implement not only the ideal scenarios generally known as the happy path but also account for failure cases, ensuring the application fails appropriately when necessary.

## Writing your first Spring Boot test

Up to this point in this chapter, we have looked at the significance of testability as a critical characteristic of enterprise applications, discussed the

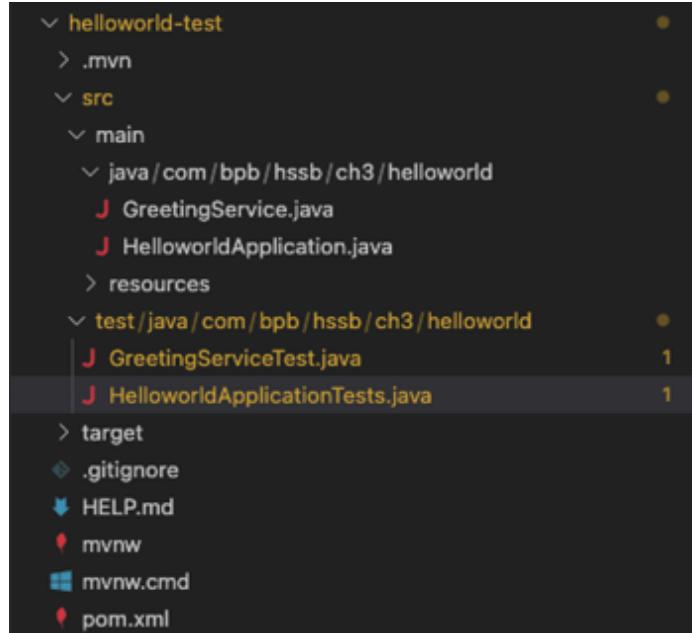
benefits of adopting TDD, and reviewed some best practices for writing unit tests. Now, let us proceed to write our first Spring Boot test case and put into practice what we have discussed so far.

The Test module in the Spring Framework supports a variety of testing types, including unit tests, integration tests, and end-to-end tests. It includes features and utilities that integrate seamlessly with popular testing frameworks such as JUnit and TestNG. Key functionalities include dependency injection for tests, transaction management, mocking, and stubbing for dependencies.

The first step is to include the **spring-boot-starter-test** starter as a dependency in your application. You can achieve this by adding it to the POM file of your application, as shown in the following code segment. Suppose you used Spring Initializr to create your sample application, as we did in our hello world example in the previous chapter. In that case, you do not need to add the `spring-boot-starter-test` dependency manually. Spring Initializr automatically includes it when generating the application. Additionally, Spring Initializr added a **HelloworldApplicationTests** test case in the project, with Spring testing already configured and operational by default.

1. `<dependency>`
2. `<groupId>org.springframework.boot</groupId>`
3. `<artifactId>spring-boot-starter-test</artifactId>`
4. `<scope>test</scope>`
5. `</dependency>`

Next, create a test class named **GreetingServiceTest.java**. Use the same package name as the **GreetingService.java** class but place it in the `/src/test/java` directory instead of `/src/main/java`, following the default Spring Boot project structure convention. The project structure after adding the test class is illustrated in the following figure:



*Figure 3.1: Project structure with test cases*

Now, you can go ahead to complete the **GreetingServiceTest** using the following code. In this code, we are trying to write a test for the greet method of the **GreetingService** class.

```
1. package com.bpb.hssb.ch3.helloworld;
2.
3. import static org.junit.jupiter.api.Assertions.assertEquals;
4. import static org.junit.jupiter.api.Assertions.assertNotNull;
5. import static org.junit.jupiter.api.Assertions.assertTrue;
6.
7. import java.time.LocalTime;
8.
9. import org.junit.jupiter.api.Test;
10. import org.springframework.beans.factory.annotation.Autowired;
11. import org.springframework.boot.test.context.SpringBootTest;
12.
13. @SpringBootTest
14. public class GreetingServiceTest {
15.
16.     @Autowired
17.     private GreetingService greetingService;
18.
19.     @Test
20.     public void testGreet() {
21.         assertNotNull(greetingService);
```

```

22.     assertNotNull(greetingService.greet());
23.     assertTrue(greetingService.greet().contains("Hello world"));
24.     if (isAM()) {
25.         assertEquals("Hello world, it's a wonderful morning!",
26.             greetingService.greet());
27.     } else {
28.         assertEquals("Hello world, it's a wonderful afternoon!",
29.             greetingService.greet());
30.
31.     private boolean isAM() {
32.         if (LocalTime.now().getHour() < 12) {
33.             return true;
34.         }
35.         return false;
36.     }
37.
38. }

```

On *line 13* of the code, we have annotated the **GreetingServiceTest** class with **@SpringBootTest**, a Spring Boot annotation that enables several key features:

- **Loads the full application context:** Enable Spring Framework to load the complete Spring application context, including beans, configurations, and dependencies, similar to how the application would be started using the **HelloworldApplication** class.
- **Automatic configuration:** This annotation also enables automatic search and loading of Spring configuration classes from the test classpath.
- **Environment properties:** Default property values can be overridden within the test execution scope by setting the **properties** attribute.
- **Application arguments:** The **args** attribute in this annotation can be used to pass application arguments to the tests.
- **Web environment modes:** For web applications, this annotation can configure the **webEnvironment** attribute to set the environment's behavior. For instance, it can provide a mock web environment without starting an embedded server or an embedded server on a random port.

- **TestRestTemplate and WebTestClient:** This annotation also registers **TestRestTemplate** and **WebTestClient** for testing RESTful services with an embedded web server.

For further clarity, a portion of the **SpringBootTest** class definition is provided as follows:

1. `@Target(TYPE)`
2. `@Retention(RUNTIME)`
3. `@Documented`
4. `@Inherited`
5. `@BootstrapWith(SpringBootTestContextBootstrapper.class)`
6. `@ExtendWith(org.springframework.test.context.junit.jupiter.SpringExtension.class)`
7. `public @interface SpringBootTest`

Then, on *line 19*, we annotated the **testGreet** method with **@Test**, an annotation provided by the JUnit 5 framework (also known as JUnit Jupiter). Methods annotated with **@Test** are identified as test methods by JUnit Jupiter. Inside the **testGreet** method, we utilized several static assertion methods provided by JUnit to validate the test results.

For completeness in our discussion, on *line 16*, we injected **GreetingService** using the **@Autowired** annotation to be utilized within our **test** method.

You have a fully functional test case that you can execute using the VS Code IDE. Running the test case will display a screen similar to the one in [Figure 3.2](#), showing graphical test results and logs from the test execution. If any tests fail, failure messages will also be displayed:



```

PROBLEMS 1 OUTPUT DEBUG CONSOLE TEST RESULTS TERMINAL PORTS
%TESTC 1 v2
%TSTTRE2,com.bpb.hssb.ch3.helloworld.GreetingServiceTest,true,1,false,1,GreetingServiceTest,,[engine:junit-jupiter]
er]/[class:com.bpb.hssb.ch3.helloworld.GreetingServiceTest]
%TSTTRE3,testGreet(com.bpb.hssb.ch3.helloworld.GreetingServiceTest),false,1,false,2,testGreet(),,[engine:junit-jupiter]
[er]/[class:com.bpb.hssb.ch3.helloworld.GreetingServiceTest]/[method:testGreet()]
%TESTS 3,testGreet(com.bpb.hssb.ch3.helloworld.GreetingServiceTest)
%TESTE 3,testGreet(com.bpb.hssb.ch3.helloworld.GreetingServiceTest)
%RUNTIME916

```

Test run at 6/22/2024, 5:21:05 PM  
OK testGreet()

*Figure 3.2: Running test cases using VS Code*

You can use the following Maven command to run the tests:

1. `./mvnw test`

When you run tests using Maven, the results will be displayed as follows:

```

2024-06-22T18:38:12.715+12:00 INFO 21772 --- [helloworld] [main] c.b.b.h.c.helloworld.GreetingServiceTest : Starting GreetingServiceTest using Java 18.0.2.1 with PID 21772 (started by sagara in /Users/sagara/Dev/gdrive/Book-Spring5/code/ch3/helloworld-test)
2024-06-22T18:38:12.715+12:00 INFO 21772 --- [helloworld] [main] c.b.b.h.c.helloworld.GreetingServiceTest : No active profile set, falling back to 1 default profile : "default"
2024-06-22T18:38:12.762+12:00 INFO 21772 --- [helloworld] [main] c.b.b.h.c.helloworld.GreetingServiceTest : Started GreetingServiceTest in 0.06 seconds (process running for 1.297)
Hello world, it's a wonderful afternoon!
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.068 s --- in com.bpb.hssb.ch3.helloworld.GreetingServiceTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO]
[INFO] Total time: 2.425 s
[INFO] Finished at: 2024-06-22T18:38:12+12:00
[INFO]

```

Figure 3.3: Running test cases using Maven

## Writing mock tests

Mocking is a testing technique used to create simulated objects, known as **mock objects**, that mimic the behavior of real objects. These mock objects enable the testing of component functionality in isolation from their actual dependencies. This approach is especially valuable in unit testing, where the objective is to test specific units of code under controlled conditions. Mocking is also useful when accessing dependencies is complex, expensive, or unreliable.

Let us explore the concept of mocking using a practical example. In our previous **hello world** sample, the **GreetingService** uses the server's local time to generate a greeting message appropriate for the time of day. Now, we want to change this behavior to return a greeting based on the caller's time instead of the server's time. To achieve this, we will start by introducing an interface called **TimeService**:

1. package com.bpb.hssb.ch3.helloworld;
- 2.
3. public interface TimeService {
- 4.
5. public boolean isAM();
- 6.
7. }

Next, we can modify the code of **GreetingService** to depend on the **TimeService** interface, as shown in the following code segment:

1. package com.bpb.hssb.ch3.helloworld;
- 2.
3. import org.springframework.stereotype.Component;
- 4.
5. @Component
6. public class GreetingService {

```

7.
8. private TimeService timeService;
9.
10. public GreetingService(TimeService timeService) {
11.     this.timeService = timeService;
12. }
13.
14. public String greet() {
15.     String msg = "Hello world";
16.     if (timeService.isAM()) {
17.         return msg + ", it's a wonderful morning!";
18.     } else {
19.         return msg + ", it's a wonderful afternoon!";
20.     }
21. }
22. }
```

In the refactored **GreetingService** code above, we utilize constructor injection to inject a bean of type **TimeService**. This bean is then used within the `greet` method to determine the current time.

We deliberately refrain from implementing the **TimeService** interface for now, yet we still aim for the unit tests of **GreetingService** to pass successfully because we should be able to test **GreetingService** isolate from other dependencies, including the **TimeService**. Let us explore how we can achieve this using the mocking concept. First, we will slightly improve the **GreetingServiceTest** from the previous section by splitting it into two test methods to enhance our test coverage. The following is the complete code for **GreetingServiceTest**:

```

1. package com.bpb.hssb.ch3.helloworld;
2.
3. import static org.junit.jupiter.api.Assertions.assertEquals;
4. import static org.junit.jupiter.api.Assertions.assertNotNull;
5. import static org.junit.jupiter.api.Assertions.assertTrue;
6.
7. import org.junit.jupiter.api.Test;
8. import org.springframework.beans.factory.annotation.Autowired;
9. import org.springframework.boot.test.context.SpringBootTest;
10.
11. @SpringBootTest
12. public class GreetingServiceTest {
```

```

13.
14. @Autowired
15. private GreetingService greetingService;
16.
17. @Test
18. void testGreetAM() {
19.
20.     assertNotNull(greetingService);
21.     assertNotNull(greetingService.greet());
22.     assertTrue(greetingService.greet().contains("Hello world"));
23.     assertEquals("Hello world, it's a wonderful morning!",
24.                  greetingService.greet());
25. }
26.
27. @Test
28. void testGreetPM() {
29.
30.     assertNotNull(greetingService);
31.     assertNotNull(greetingService.greet());
32.     assertTrue(greetingService.greet().contains("Hello world"));
33.     assertEquals("Hello world, it's a wonderful afternoon!",
34.                  greetingService.greet());
35. }
36. }
```

Now, feel free to run the test case as we did in the previous section. If everything has been done correctly so far, you will likely encounter test failures accompanied by error logs. These errors indicate that the Spring Framework cannot find any bean of type **TimeService**, and consequently, it cannot successfully build the **ApplicationContext**. This failure occurs because **GreetingServiceTest** expects a **GreetingService** bean, which depends on a **TimeService** bean that is currently missing.

This is the perfect moment for us to bring the mocking concept into action. First, modify the **GreetingServiceTest** as per the following code segment:

```

1. package com.bpb.hssb.ch3.helloworld;
2.
3. import static org.junit.jupiter.api.Assertions.assertEquals;
4. import static org.junit.jupiter.api.Assertions.assertNotNull;
```

```
5. import static org.junit.jupiter.api.Assertions.assertTrue;
6. import static org.mockito.Mockito.when;
7.
8. import org.junit.jupiter.api.Test;
9. import org.springframework.beans.factory.annotation.Autowired;
10. import org.springframework.boot.test.context.SpringBootTest;
11. import org.springframework.boot.test.mock.mockito.MockBean;
12.
13. @SpringBootTest
14. public class GreetingServiceTest {
15.
16.     @Autowired
17.     private GreetingService greetingService;
18.
19.     @MockBean
20.     private TimeService remoteTimeService;
21.
22.     @Test
23.     void testGreetAM() {
24.
25.         when(remoteTimeService.isAM()).thenReturn(true);
26.
27.         assertNotNull(greetingService);
28.         assertNotNull(greetingService.greet());
29.         assertTrue(greetingService.greet().contains("Hello world"));
30.         assertEquals("Hello world, it's a wonderful morning!",
31.             greetingService.greet());
32.     }
33.
34.     @Test
35.     void testGreetPM() {
36.
37.         when(remoteTimeService.isAM()).thenReturn(false);
38.
39.         assertNotNull(greetingService);
40.         assertNotNull(greetingService.greet());
41.         assertTrue(greetingService.greet().contains("Hello world"));
42.         assertEquals("Hello world, it's a wonderful afternoon!",
43.             greetingService.greet());
44.     }
45.
```

```
44.  }
45. }
```

If you run the test again, you may be surprised to see that both tests have passed successfully. On *line 20*, we declared **TimeService** as a field in the test class and annotated it with **@MockBean**. The **@MockBean** annotation instructs the Spring Framework to automatically create a Mockito mock for the **TimeService** interface and add it as a bean to the Spring application context. It also replaces any existing bean of **TimeService** available in the application context. An important point to highlight here is that the mocking behavior of the mock object resets after each **test** method.

Then, on *lines 25* and *37*, we defined the expected behavior of the mock object using the **when().thenReturn()** API provided by Mockito, according to the requirements of each test method. To clarify, the purpose of **GreetingServiceTest** is to test **GreetingService** in isolation, so it is perfectly acceptable to mock the behavior of **TimeService** in this scenario.

## Writing tests for web applications

As the final topic before concluding our discussion on Spring testing in this chapter, let us explore how to write a test case for a web application. We will begin by adding a test case for the RESTful service we developed in the previous chapter. First, we will write a test using the Spring embedded server environment. This test, named **HelloControllerWebTest**, aims to verify the behavior of the **HelloController** class. The complete code for this test class is provided as follows. You can also find this sample in this book's companion GitHub source repository.

```
1. package com.bpb.hssb.ch3.helloworld.web;
2.
3. import org.junit.jupiter.api.Test;
4. import org.springframework.beans.factory.annotation.Autowired;
5. import org.springframework.boot.test.context.SpringBootTest;
6. import org.springframework.boot.test.context.SpringBootTest.
   WebEnvironment;
7. import org.springframework.boot.test.web.client.TestRestTemplate;
8. import org.springframework.boot.test.web.server.LocalServerPort;
9.
10. import static org.assertj.core.api.Assertions.assertThat;
11.
```

```

12. @SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
13. public class HelloControllerWebTest {
14.
15.     @LocalServerPort
16.     private int port;
17.
18.     @Autowired
19.     private TestRestTemplate restTemplate;
20.
21.     @Test
22.     void testHello() throws Exception {
23.         assertThat(this.restTemplate.getForObject
24.             ("http://localhost:" + port + "/hello",
25.             String.class)).contains("Hello world");
26.
27. }

```

Similar to our previous test cases, here we have used the **@SpringBootTest** annotation, but this time with **webEnvironment=WebEnvironment.RANDOM\_PORT** attribute. By specifying this attribute, Spring Boot initiates an embedded web server like Tomcat during test execution, using a random port to prevent port conflicts. This setup is beneficial when testing the complete HTTP request-response cycle, which includes server-side routing and request handling after loading the entire application context. This test case can be considered as an integration test.

On *line 15*, we injected the randomly assigned port number chosen by the Spring Framework into the test class for use within the **testHello** method using **@LocalServerPort** annotation. Then, on *line 19*, we declared **TestRestTemplate** to be used in our test method and injected it using the **@Autowired** annotation. **TestRestTemplate** is a specialized version of Spring's **RestTemplate** tailored for testing in Spring Boot applications. It simplifies the testing of RESTful web services by sending HTTP requests and handling responses.

The Spring Framework also provides a flexible mock module called **MockMvc** for testing Spring MVC controllers without needing to deploy them on an actual web server. **MockMvc** simulates HTTP requests and provides methods to interact with the responses generated by Spring MVC

controllers. It includes built-in mechanisms for assertions and validation to check status codes, headers, and response messages. Additionally, MockMvc can be configured and extended to suit specific testing requirements as well.

As our final test case in this section, let us write a test using MockMvc. The **HelloControllerMockTest** shown in the following code utilizes MockMvc to test the behavior of **HelloController**:

```
1. package com.bpb.hssb.ch3.helloworld.web;
2.
3. import static org.hamcrest.Matchers.containsString;
4. import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
5. import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.content;
6. import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;
7.
8. import org.junit.jupiter.api.Test;
9. import org.springframework.beans.factory.annotation.Autowired;
10. import org.springframework.boot.test.autoconfigure.web.servlet.MockMvcMvc;
11. import org.springframework.boot.test.context.SpringBootTest;
12. import org.springframework.test.web.servlet.MockMvc;
13.
14. @SpringBootTest
15. @AutoConfigureMockMvc
16. public class HelloControllerMockTest {
17.
18.     @Autowired
19.     private MockMvc mockMvc;
20.
21.     @Test
22.     void testHello() throws Exception {
23.         this.mockMvc.perform(get("/hello")).andExpect(status().isOk())
24.             .andExpect(content().string(containsString("Hello world")));
25.     }
26.
27. }
```

In the above code, the **@AutoConfigureMockMvc** annotation instructs Spring Boot to automatically set up and inject a **MockMvc** instance into the test class. Then, on *line 19*, we inject this automatically configured **MockMvc** instance into the test class using the **@Autowired** annotation.

Finally, on *line 24*, we send a GET request to the **/hello** endpoint and verify

that we receive an **HTTP 200** status code, along with a response containing **Hello, world**. This approach demonstrates how convenient it is to write unit tests for Spring MVC controllers using MockMvc.

## Securing Spring Boot applications

Securing an application is a broad topic, and in this section, we will only look at three important concepts related to securing Spring Boot web applications. Depending on the business use cases and the architecture of the application, you may need to consider more security measures, but these three can be considered fundamental aspects of securing an enterprise application. Their three fundamental concepts are:

- **Authentication:** Authentication is the process of verifying that users are who they claim to be. It is a primary requirement for ensuring that only authorized users can access protected resources or perform specific actions within an application.
- **Authorization:** Authorization is the process of determining what actions or resources a user is allowed to access. It typically occurs after authentication once the user's identity has been verified.
- **Protection against exploits:** Protection against exploits refers to the measures and techniques used to safeguard an application from malicious attempts to exploit vulnerabilities or weaknesses, such as cross-site scripting, SQL injection, and other similar attacks.

When it comes to implementing authentication and authorization, the simplest approach would be to utilize a database table within your application to store user profiles, including hashed credentials, after applying adaptive one-way hash algorithms such as bcrypt, PBKDF2, scrypt, or argon2. Implementing authentication and user management features using Spring Boot can be straightforward, integrating these capabilities seamlessly with other business logic and can result in the utilization of infrastructure resources optimally. This approach keeps the overall application design simple.

However, as security standards evolve continuously, ongoing efforts may be required to enhance passwords and other security mechanisms. Implementing additional features like **multi-factor authentication (MFA)**

or integrating social logins would require additional effort. Moreover, you may need to implement and maintain features such as password reset mechanisms, user profile access, and compliance with privacy regulations like GDPR and CCPA.

Alternatively, another option is to delegate user management and authentication capabilities to a purpose-built **identity and access management (IAM)** solution that is independently deployed and managed. Many organizations have a common identity solution in place, which the application can integrate with using APIs or customizable user interfaces provided by the identity provider.

The primary advantages of this approach include eliminating the need to implement and update security features by yourself and out-of-the-box support for features such as social logins, MFA, passwordless logins, user management capabilities, consent management, and self-care services. You can run a proprietary identity solution or choose an open-source identity provider like Spring Authorization Server, Keycloak, or WSO2 Identity Server. Another option would be to use cloud identity providers such as Okta, Auth0, and Asgardeo, which reduce the complexity of running your own identity provider. Most of the cloud identity providers offer free tiers and usage-based pricing models.

However, this approach may initially seem overkill, requiring additional effort to onboard an identity provider, allocate additional infrastructure resources in case of on-premises deployments, and manage the integration process. The application architecture may become a bit more complex due to the need to connect with the identity provider for authentication and user management capabilities.

In summary, there is no definitive right or wrong answer; the choice between these approaches depends on your initial goals, long-term plans, budget constraints, and the complexity of your application architecture.

## Spring Security

Spring Security is a sub-project of the Spring Framework and is designed to provide highly customizable security for Spring-based applications. It can be used with both Spring MVC servlet-based applications and Spring WebFlux reactive applications. In addition to securing HTTP-based applications,

Spring Security can also be used to secure WebSocket applications as well. User authentication is one of the fundamental functionalities supported by Spring Security. It offers a range of authentication mechanisms, including:

- Username and password login using form-based and HTTP BasicAuth
- OAuth 2.0 and OpenID Connect login
- SAML 2.0 login
- **Central Authentication Server (CAS) login**
- JAAS authentication
- X509 authentication

Spring Security also supports several authorization mechanisms to be used after successful user authentication, including:

- Authorization for HTTP requests
- Method security
- Domain object security ACLs

In addition to authentication and authorization features, Spring Security also provides protection against common attacks such as:

- **Cross-Site Request Forgery (CSRF)**
- Security HTTP Response Headers
- HttpFirewall

## Writing your first Spring Security app

Let us use the RESTful service that was created in the previous chapter to demonstrate Spring Security. The first and most important step in securing a Spring Boot application is to add `spring-boot-starter-security` as a dependency in the Maven POM file of the project, as follows:

1. `<dependency>`
2. `<groupId>org.springframework.boot</groupId>`
3. `<artifactId>spring-boot-starter-security</artifactId>`
4. `</dependency>`

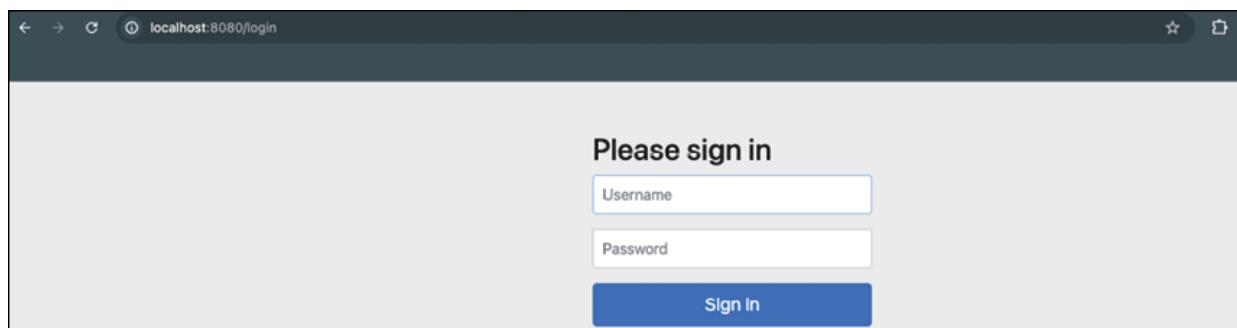
Adding the above starter provides a solid foundation for securing an application, as it automatically includes opinionated default security configurations:

- It configures basic authentication for all HTTP endpoints by default,

ensuring that all endpoints require authentication to access.

- It enables features to render a default login page when accessing the web application from a web browser, leveraging Spring Security's content negotiation capabilities.
- During start-up, it generates a default username and a random password, which are printed in the console logs for development purposes.
- It includes security controls to protect against common exploits such as CSRF attacks, and session fixation attacks, and adds security headers to responses.

If you have already added **spring-boot-starter-security** to your project, you can proceed to run your application using VS Code or Maven. When you visit the <http://localhost:8080/login> URL in your web browser, instead of the expected **Hello World** message, you will see the default login screen generated by Spring Boot, as follows:



**Figure 3.4:** Default login page generated by Spring Security

At this point, if you check the application logs, you will notice that Spring Boot has automatically created a test user named **user** with a random password. Additionally, a warning message will be printed indicating that this user should only be used for development purposes. This is shown in the following figure:



```

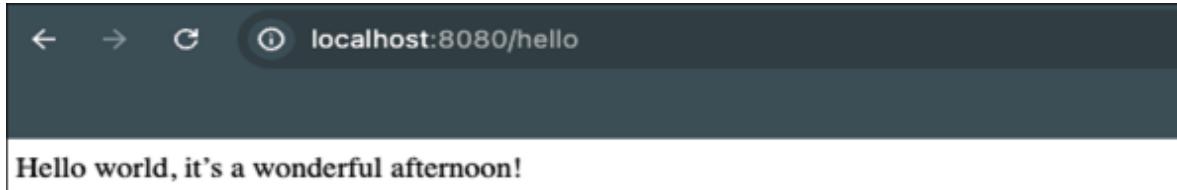
:: Spring Boot ::          (v3.3.0)

2024-06-23T19:29:55.377+12:00  INFO 88817 --- [helloworld-web] [main] c.b.h.c.h.web.HelloworldWebApplication : Starting HelloworldWebApplication using Java 17.0.2 with
PID 88817 (/Users/sagara/Dev/gdrive/Book-Spring6/code/ch3/helloworld-security-basic/target/classes started by sagara in /Users/sagara/Dev/gdrive/Book-Spring6/code/ch3/helloworld-s
ecurity-basic)
2024-06-23T19:29:55.379+12:00  INFO 88817 --- [helloworld-web] [main] c.b.h.c.h.web.HelloworldWebApplication : No active profile set, falling back to 1 default profile
: "default"
2024-06-23T19:29:55.787+12:00  INFO 88817 --- [helloworld-web] [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port 8080 (http)
2024-06-23T19:29:55.798+12:00  INFO 88817 --- [helloworld-web] [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2024-06-23T19:29:55.798+12:00  INFO 88817 --- [helloworld-web] [main] o.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/10.1.24]
2024-06-23T19:29:55.827+12:00  INFO 88817 --- [helloworld-web] [main] o.a.c.c.c.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2024-06-23T19:29:55.828+12:00  INFO 88817 --- [helloworld-web] [main] w.s.c.WebServerApplicationContext : Root WebApplicationContext: initialization completed in
428 ms
2024-06-23T19:29:55.979+12:00  WARN 88817 --- [helloworld-web] [main] .s.s.UserDetailsServiceAutoConfiguration : Using generated security password: abf917dc-a296-4f5c-9da0-d366f9a1441d
This generated password is for development use only. Your security configuration must be updated before running your application in production.

```

**Figure 3.5: Default password generation**

If you enter the *user* as the username and the randomly generated password mentioned earlier, you should be able to see the expected **hello world** message, as shown in the following figure:



**Figure 3.6: hello endpoint**

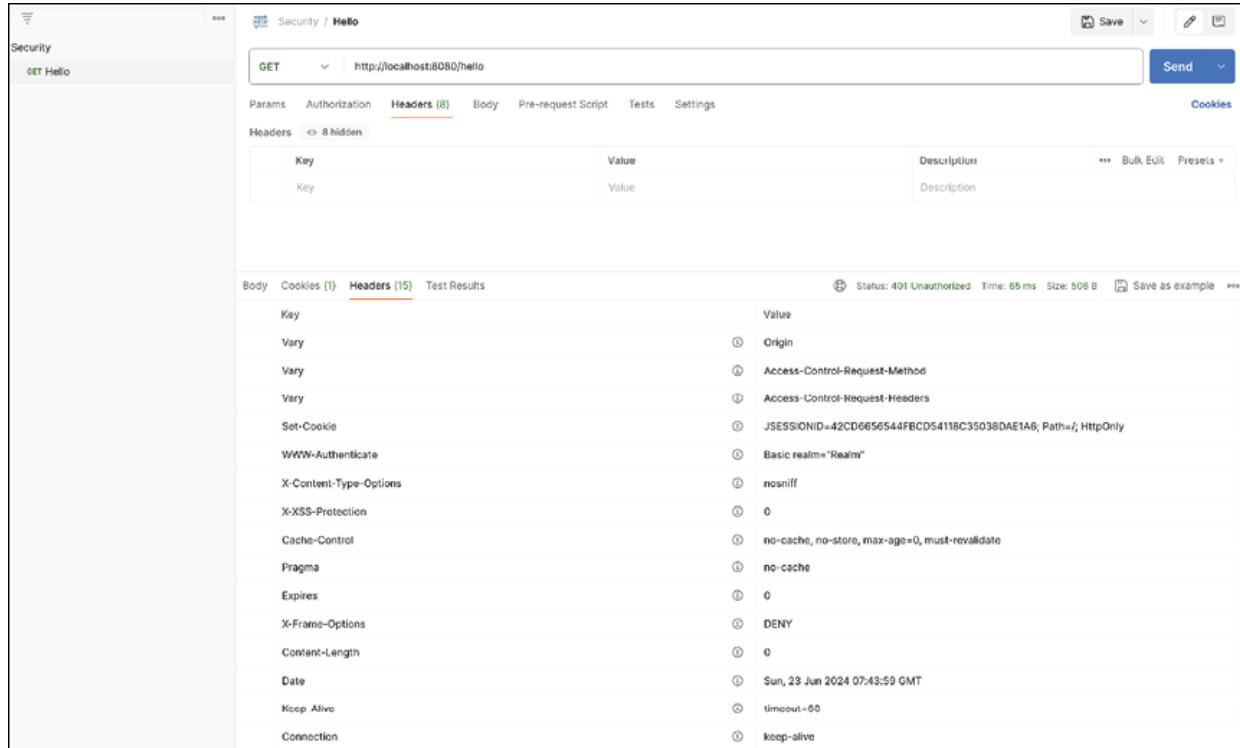
This is how you can enable default security configuration for your Spring Boot web applications without writing any code. If you prefer not to use a randomly generated password each time you run the application, you can easily set your username and password using configuration files. Simply add the following two lines to the **application.properties** file:

1. `spring.security.user.name=testuser`
2. `spring.security.user.password=testpwd`

Now, you will not see the automatic random password generation in the application logs anymore because the above settings disable that feature. Instead, you should be able to log in to the application using the above-specified username and password.

Let us also access the <http://localhost:8080/login> URL from an HTTP client other than a web browser, such as cURL or Postman. For demonstration purposes, we will use Postman, but you are free to use any HTTP client of your choice. This time, when you make this HTTP call, as shown in [Figure 3.7](#), you will receive a **401 Unauthorized** status instead of a login form. This is because Spring Boot detects that you are calling from a non-browser

client and responds accordingly, this is known as content negotiation:

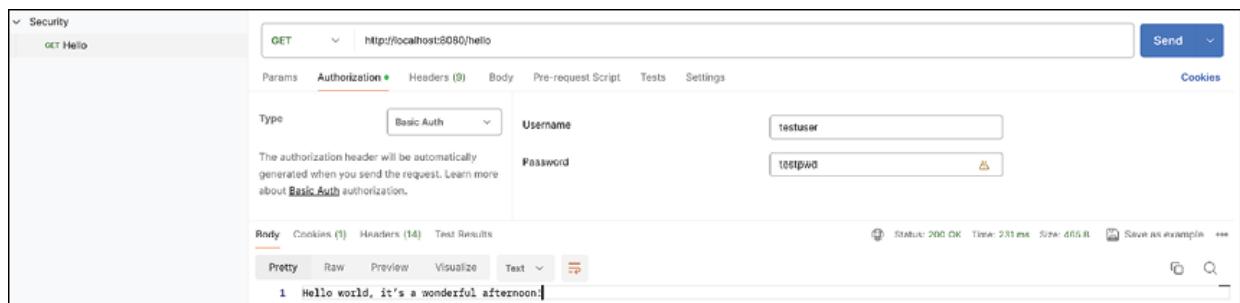


The screenshot shows a Postman request for a GET method to `http://localhost:8080/hello`. The Headers tab is selected, displaying 15 headers. The status bar indicates a 401 Unauthorized response with a 65 ms time and 508 B size. The Headers table includes columns for Key, Value, Description, Bulk Edit, and Presets.

Key	Description	Bulk Edit	Presets
Origin			
Access-Control-Request-Method			
Access-Control-Request-Header			
JSESSIONID	=42CD6656544FBCD54118C35038DAE1A; Path=/; HttpOnly		
Basic realm="Realm"			
nosniff			
0			
no-cache, no-store, max-age=0, must-revalidate			
no-cache			
0			
DENY			
0			
Sun, 23 Jun 2024 07:43:59 GMT			
timeout=60			
keep-alive			

*Figure 3.7: Unauthorized request in Postman*

If you execute the same call after setting the username and password in the **Authorization** tab, as shown in the following figure, you should be able to see the hello world message again:

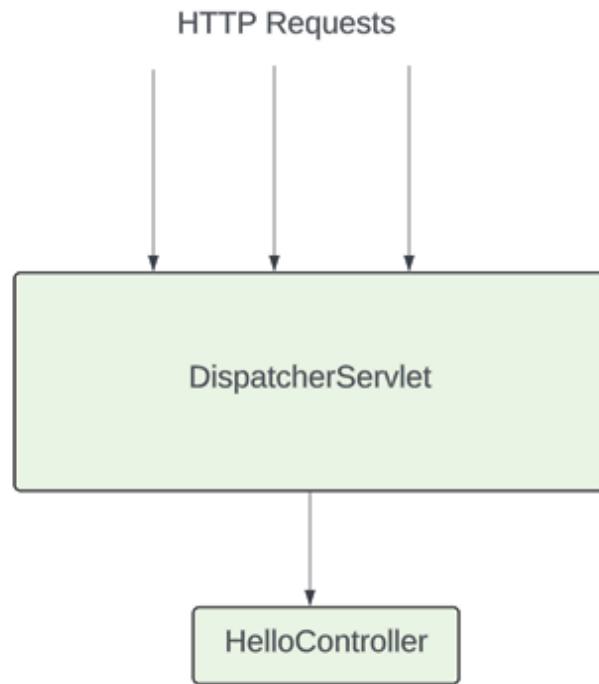


The screenshot shows a Postman request for a GET method to `http://localhost:8080/hello`. The Authorization tab is selected, showing 'Basic Auth' with 'testuser' and '10spwd' entered. The status bar indicates a 200 OK response with a 231 ms time and 476 B size. The Body tab shows the response: 'Hello world, it's a wonderful afternoon.'

*Figure 3.8: Successful request in Postman*

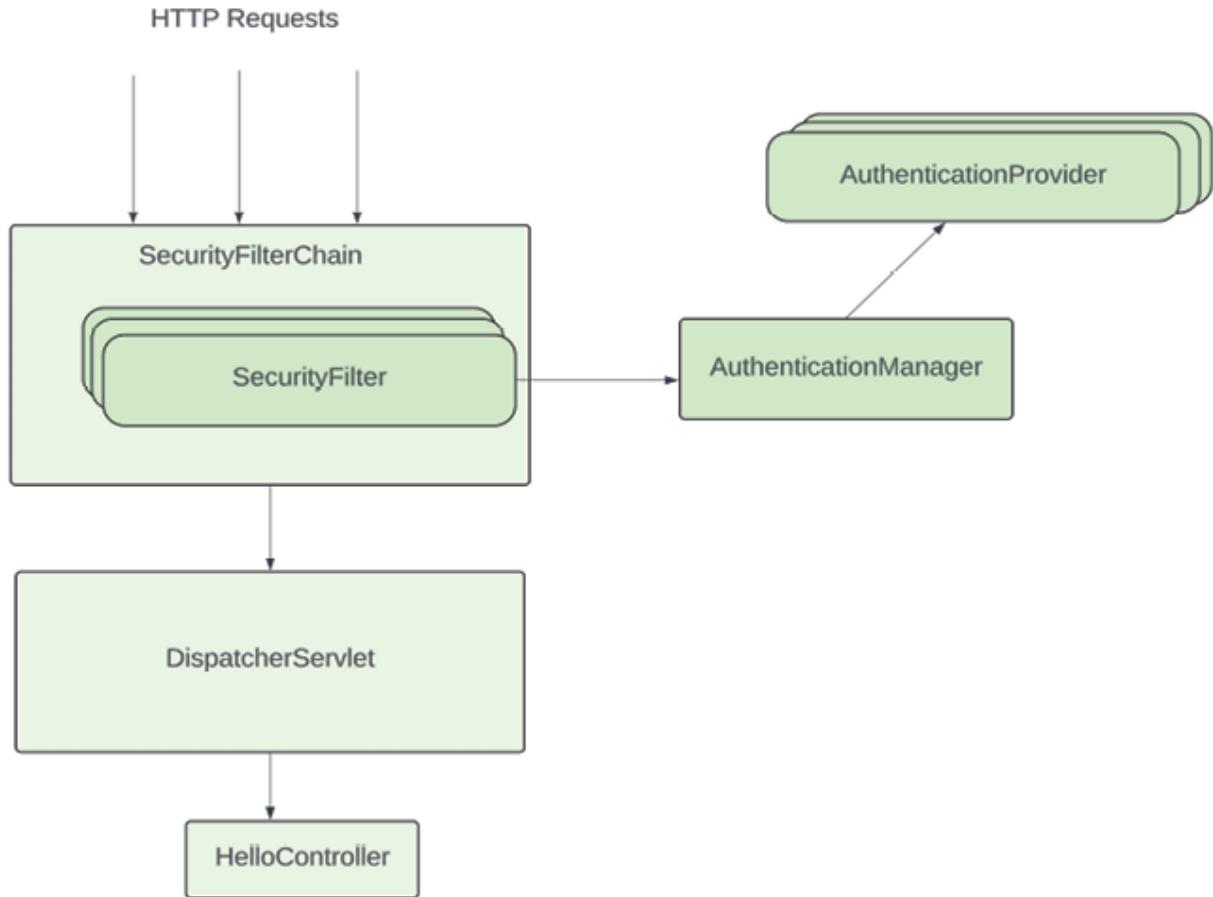
As you have managed to run and test your first Spring Security application, let us look into what exactly happens under the hood. Initially, before adding the Spring Security starter to this project, incoming HTTP requests are first handled by a special Servlet provided by the Spring MVC project known as **DispatcherServlet**. This **DispatcherServlet** delegates the requests to

appropriate handlers, such as our **HelloController**, as shown in the following figure:



*Figure 3.9: Default HTTP request processing in Spring MVC*

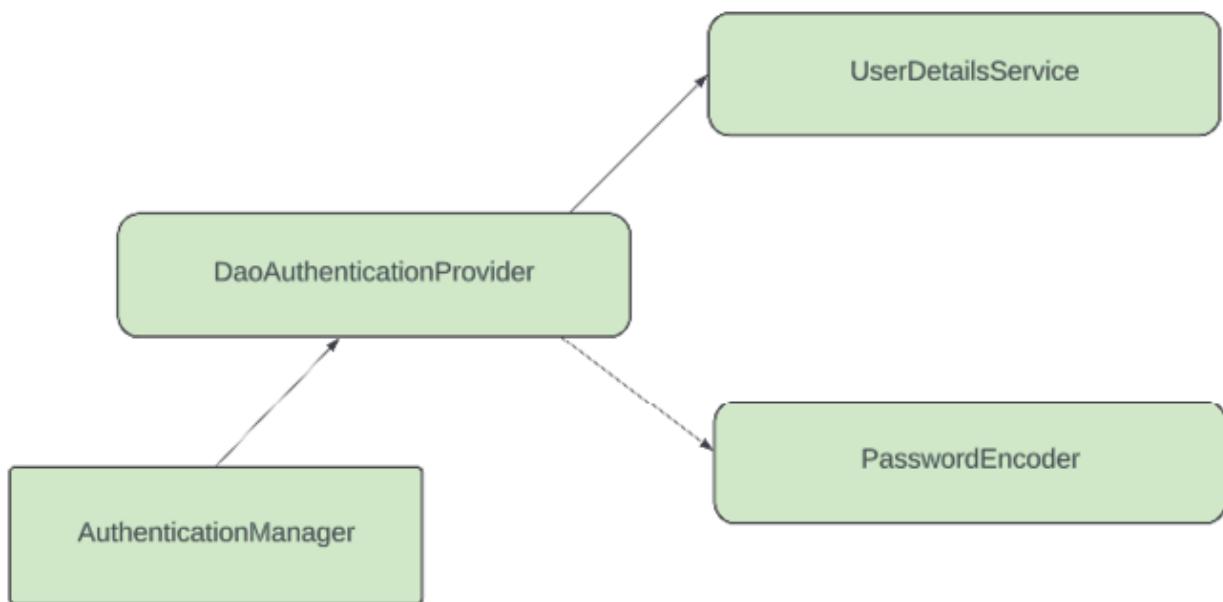
However, once we add the Spring Security starter to the project, this default behavior changes. Now, instead of **DispatcherServlet**, the **SecurityFilterChain** provided by Spring Security handles the incoming HTTP requests before they reach **DispatcherServlet**. The **SecurityFilterChain** applies Spring Security features based on the provided security configuration. This request-handling flow is shown in the following figure:



**Figure 3.10:** HTTP request processing in Spring MVC after adding Spring Security dependencies

As depicted in [Figure 3.10](#), **SecurityFilterChain** does not perform any security tasks by itself. Instead, it delegates responsibility to an ordered list of **SecurityFilters**. Each **SecurityFilter** attempts to extract authentication information from the HTTP request to move the security processing to the next stage. For example, **UsernamePasswordAuthenticationFilter** tries to extract authentication information using form-based POST requests, while **BasicAuthenticationFilter** looks for the presence of an HTTP Authorization header and tries to extract authentication information based on the HTTP BasicAuth schema. When a **SecurityFilter** successfully extracts authentication information from a request, it delegates the processing to the **AuthenticationManager** to authenticate the user. The **AuthenticationManager** implementation includes several **AuthenticationProviders**, each designed to authenticate users using specific methods. For instance, **ActiveDirectoryLdapAuthenticationProvider** authenticates users via Active Directory, while **JwtAuthenticationProvider** verifies users based on JWT tokens.

In our sample application, an **AuthenticationProvider** called **DaoAuthentication Provider** is responsible for performing the actual user authentication process. The **DaoAuthenticationProvider** attempts to authenticate the user using two other components: **UserDetailsService** and **PasswordEncoder**. The **UserDetailsService** retrieves user data stored in-memory or in a database, while the **PasswordEncoder** verifies the submitted password against the stored password hash. These components are shown in the following figure:



*Figure 3.11: Spring Security AuthenticationManager*

If the verification is successful, the current user is authenticated, and a special context object called **SecurityContextHolder** is created to store the authentication information.

Even though we explored a complete Spring Security sample based on default configurations, we must configure Spring Security according to our application's specific requirements. To make our sample realistic, let us add a new class named **SecurityConfiguration.java** with the following code:

```
1. package com.bpb.hssb.ch3.helloworld.web;
2.
3. import static org.springframework.security.config.Customizer.withDefaults;
4.
5. import org.springframework.context.annotation.Bean;
6. import org.springframework.context.annotation.Configuration;
```

```

7. import org.springframework.security.config.annotation.web.builders.HttpSecurity;
8. import org.springframework.security.core.userdetails.User;
9. import org.springframework.security.core.userdetails.UserDetails;
10. import org.springframework.security.provisioning.InMemoryUserDetailsManager;
11. import org.springframework.security.web.SecurityFilterChain;;
12.
13. @Configuration
14. public class SecurityConfiguration {
15.
16.     @Bean
17.     public SecurityFilterChain filterChain
18.         (HttpSecurity http) throws Exception {
19.         http
20.             .authorizeHttpRequests((authz) -> authz
21.                 .anyRequest().hasRole("USER"))
22.             .httpBasic(withDefaults());
23.     }
24.
25.     @Bean
26.     public InMemoryUserDetailsManager userDetailsService() {
27.         UserDetails user = User.withDefaultPasswordEncoder()
28.             .username("user")
29.             .password("password")
30.             .roles("USER")
31.             .build();
32.         return new InMemoryUserDetailsManager(user);
33.     }
34.
35. }

```

In the code, we have annotated the **SecurityConfiguration** class with **@Configuration**. During start-up, Spring picks up this configuration, replacing beans configured by default auto-configuration mechanisms with those defined in **SecurityConfiguration**. This setup is essentially similar to any other Spring bean configuration class.

At *line 17*, we have replaced the default **SecurityFilterChain** with our custom configuration. This configuration attempts to authenticate all incoming HTTP requests, allowing only users with the **USER** role to proceed.

On *line 26*, we have configured an instance of **UserDetailsService** named **InMemoryUserDetailsService**, which stores user data in-memory. While suitable for our example, it is not suitable for real-world applications where production-ready implementations are necessary. Within this setup, we have added a user named **user** with the password **password** and assigned them the role **USER**. Additionally, we have configured a **DefaultPasswordEncoder** for our sample.

If you run the application again, you can now use **user** as the username and **password** as the password to invoke the RESTful endpoint. Just as we have set up our own **SecurityFilterChain** and **UserDetailsService**, you have the flexibility to replace any processing component of Spring Security with your own configuration or custom implementations.

At this stage, we are intentionally postponing the discussion of Spring Security authorization architecture. We will revisit this topic when we look into Spring MVC in the next chapter.

## Observability of an application

Observability of an application refers to the ability to understand the internal state of a running application by examining its external outputs. It provides deep insights into application performance, behavior, and issues. To clarify the concept of observability, we can compare it to the diagnostic process used by healthcare professionals, who monitor vital signs like pulse rate, body temperature, and blood pressure. Similarly, in software systems, observability relies on telemetry data such as events, logs, metrics, and traces. In both fields, proactive and continuous monitoring is essential. In medicine, regular check-ups help detect potential issues early, while in software, observability allows engineers to proactively analyze and optimize systems before problems escalate. Observability consists of three main pillars:

- **Metrics:** Metrics are quantitative data points that indicate the health and performance of an application. They provide real-time visibility into how an application is performing. Applications typically emit metrics as numerical values formatted as timestamped name-value pairs at regular intervals. This allows external metrics analytical tools to collect the data and provide historical and current insights that are easy for humans to

understand. For example, typical metrics of an application include CPU utilization, memory usage, network traffic, and the number of HTTP requests.

- **Logging:** Logs can be files or event streams that record events, warnings, and errors as they occur within an application. Usually, these events are recorded with timestamps. Log entries vary based on the amount of information they contain and the conditions under which they should be logged, known as log levels. Log levels are ordered based on severity to avoid excessive noise and performance degradation. In most situations, log levels can be adjusted at runtime as needed. Logs are useful in real-time for understanding the internal state of an application. For instance, without server logs, it would be difficult to determine if a Tomcat server has started and is ready to serve incoming requests. Additionally, like metrics, logs can be collected by external log analysis tools to gather, store, and analyze data, generating useful insights about the application.
- **Traces:** Traces offer a detailed view of the flow of requests as they move through an application or, more commonly, across multiple applications within a business system. They record the entire journey of a request or action across various services or components. Traces are valuable for understanding the end-to-end behavior of requests, aiding in debugging and monitoring. They typically provide a visual mapping of dependencies and interactions between services.

## Spring Boot Actuators

Spring Boot Actuator is a sub-project of Spring Boot that aims to provide several production-grade services to your application with minimal effort. Actuators enable monitoring and managing your Spring Boot application, both during development and in production, using HTTP endpoints and JMX. The term actuator originated from manufacturing, where it refers to a mechanical device for moving or controlling something that can generate a large amount of motion from a small change. Spring Boot actuators are also designed to enable production-grade features with very little effort.

The best approach to understand how easy it is to use Spring Boot actuators is to use them in our sample application; let us try to add actuators into the

Restful application that we used in the previous sections. In the book's GitHub repository, you can refer to the sample named **helloworld-metrics** to find the full codes.

The very first step to enabling a Spring Boot actuator is to add a **spring-boot-starter-actuator** starter as a dependency in the project's POM file. As you would imagine, adding this dependency to the classpath instructs the auto-configuration mechanisms of Spring Boot to enable several production-grade features in our application.

1. <dependency>
2. <groupId>org.springframework.boot</groupId>
3. <artifactId>spring-boot-starter-actuator</artifactId>
4. </dependency>

Now, if you run the application, you will see the following entry in the application logs because the presence of the actuator dependencies is enough for the Spring Boot auto-configuration mechanism to enable some of the production-grade features. As we will see in the following figure, we can make configuration changes just by using the **application.properties** file alone:

```
2024-06-30T23:00:12.626+13:00 INFO 13278 --- [helloworld-web] [      main]
o.s.b.a.e.web.EndpointLinksResolver : Exposing 5 endpoints beneath base path '/actuator'
2024-06-30T23:00:12.656+13:00 INFO 13278 --- [helloworld-web] [      main]
o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 8080 (http) with context path '/'
2024-06-30T23:00:12.664+13:00 INFO 13278 --- [helloworld-web] [      main]
c.b.h.c.h.web.HelloworldWebApplication : Started HelloworldWebApplication in 1.041 seconds
(process running for 1.38)
2024-06-30T23:00:14.859+13:00 INFO 13278 --- [helloworld-web] [on(2)-127.0.0.1] o.a.c.c.C.
[Tomcat].[localhost]. [/] : Initializing Spring DispatcherServlet 'dispatcherServlet'
2024-06-30T23:00:14.859+13:00 INFO 13278 --- [helloworld-web] [on(2)-127.0.0.1]
o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
2024-06-30T23:00:14.860+13:00 INFO 13278 --- [helloworld-web] [on(2)-127.0.0.1]
o.s.web.servlet.DispatcherServlet : Completed initialization in 0 ms
```

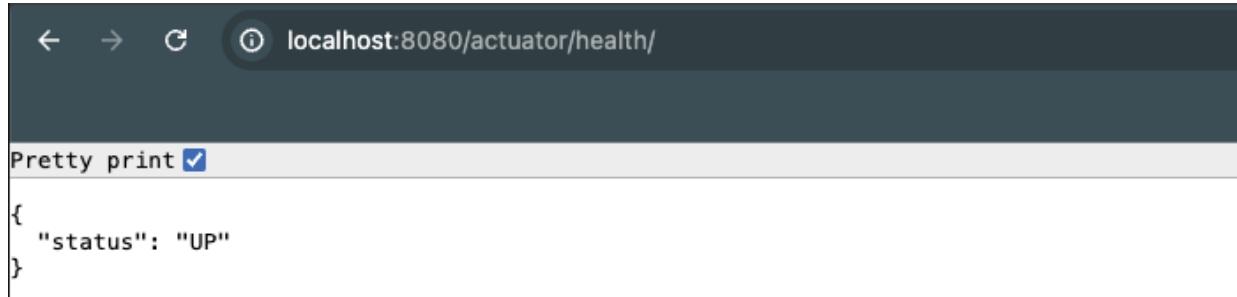
If we hit the above URL in a web browser, we would see the following output, which lists all the default endpoints enabled by the actuator:



```
pretty print
{
  "_links": {
    "self": {
      "href": "http://localhost:8080/actuator",
      "templated": false
    },
    "health": {
      "href": "http://localhost:8080/actuator/health",
      "templated": false
    },
    "health-path": {
      "href": "http://localhost:8080/actuator/health/{*path}",
      "templated": true
    }
  }
}
```

**Figure 3.12: Spring Boot actuator endpoint**

Let us go ahead and hit the health endpoint, as shown in the following figure:

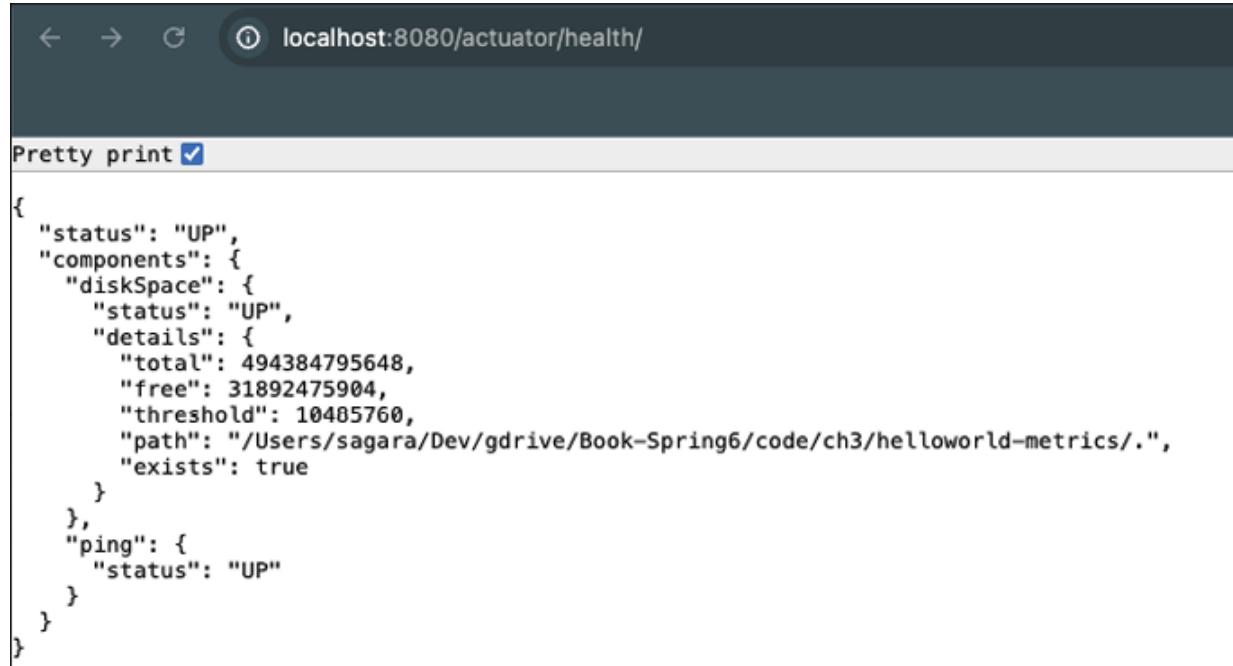


```
pretty print
{
  "status": "UP"
}
```

**Figure 3.13: Spring Boot default health endpoint**

**Figure 3.13** provides an output from the health endpoint, which is a part of the Spring Boot actuator and provides detailed information about the health of an application and can be used to monitor software to alert someone when a production application goes down or when the available free disk space went critically low. By default, only a simple *status* is shown and indicates that the application is up and running. To get more detailed health insight, go ahead and add the following line to the **application.properties** file and rerun the application. Under the hood, a set of beans called **HealthIndicator** configured in the **ApplicationContext** is responsible for rendering the above information. When appropriate dependencies are available in the classpath, the Spring boot provides a set of **HealthIndicators**, which includes

**DataSourceHealthIndicator**, **JmsHealthIndicator**,  
**LdapHealthIndicator**, **MailHealthIndicator**, **MongoHealth Indicator**,  
**MongoHealthIndicator**.



```
Pretty print 
```

```
{  
  "status": "UP",  
  "components": {  
    "diskSpace": {  
      "status": "UP",  
      "details": {  
        "total": 494384795648,  
        "free": 31892475904,  
        "threshold": 10485760,  
        "path": "/Users/sagara/Dev/gdrive/Book-Spring6/code/ch3/helloworld-metrics./",  
        "exists": true  
      }  
    },  
    "ping": {  
      "status": "UP"  
    }  
  }  
}
```

*Figure 3.14: Spring Boot health endpoint with additional configuration*

This time, it showed additional health data related to disk space, such as total and free disk space. You can also add your own **HealthIndicator** to indicate the health of your critical application components quite easily. As a simple exercise, let us go ahead and add the **Health** status for **RemoteTimeServcie** that we discussed in the testing section of this chapter. Create a file called **RemoteTimeServcieHealthIndicator.java** and add the following code:

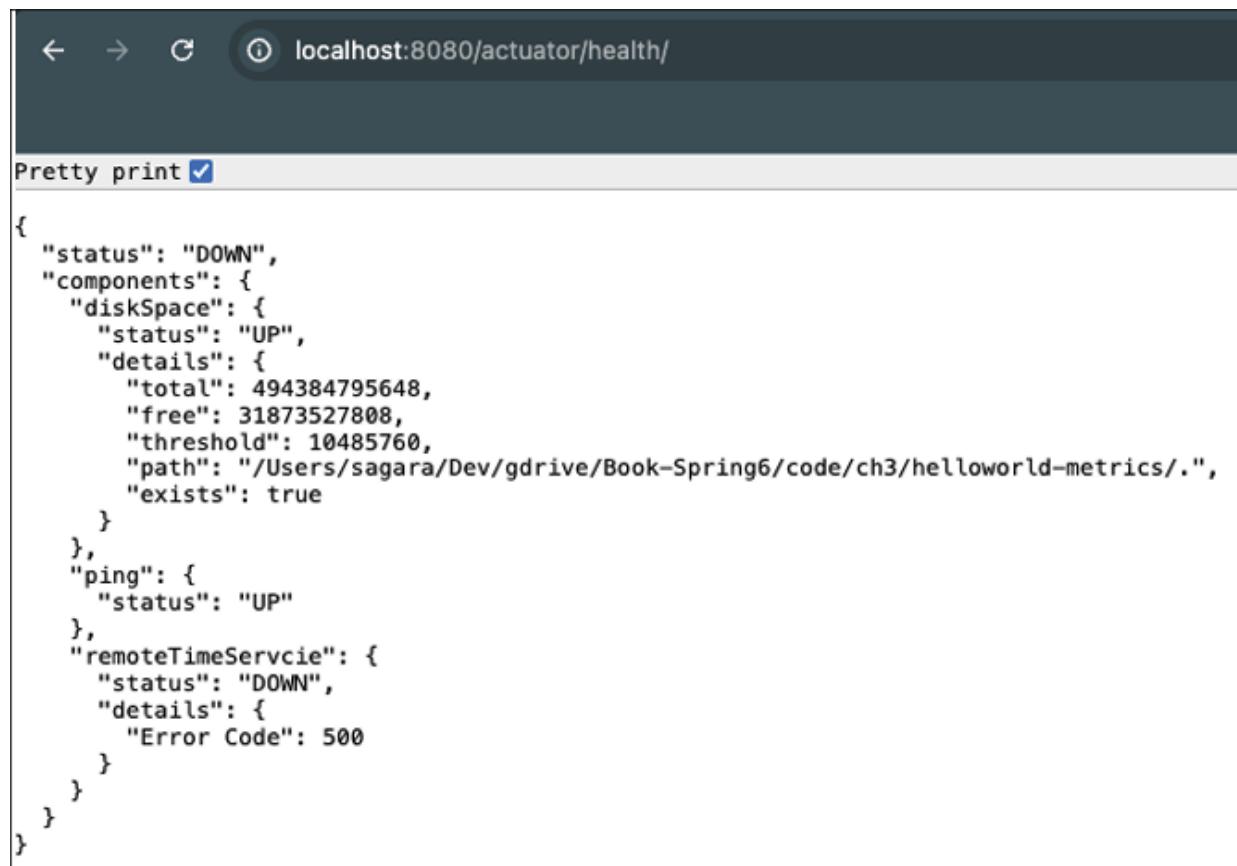
```
1. package com.bpb.hssb.ch3.helloworld.web;  
2.  
3. import org.springframework.boot.actuate.health.Health;  
4. import org.springframework.boot.actuate.health.HealthIndicator;  
5. import org.springframework.stereotype.Component;  
6.  
7. @Component  
8. public class RemoteTimeServcieHealthIndicator  
  implements HealthIndicator {  
9.  
10.  @Override  
11.  public Health health() {
```

```

12.     if (isServiceAvailable() == false) {
13.         return Health.down().withDetail("Error Code", 500).build();
14.     }
15.     return Health.up().build();
16. }
17.
18. private boolean isServiceAvailable() {
19.     return false;
20. }
21.
22. }

```

As we have not implemented the **RemoteTimeServcie** yet, we simply show an error message. After all, the main objective of the above code is to show you how easy it is to implement your own **HealthIndicators**. If you rerun the application and visit the health endpoint, you will now be able to see the following output.



```

{
  "status": "DOWN",
  "components": {
    "diskSpace": {
      "status": "UP",
      "details": {
        "total": 494384795648,
        "free": 31873527808,
        "threshold": 10485760,
        "path": "/Users/sagara/Dev/gdrive/Book-Spring6/code/ch3/helloworld-metrics/.",
        "exists": true
      }
    },
    "ping": {
      "status": "UP"
    }
  },
  "remoteTimeServcie": {
    "status": "DOWN",
    "details": {
      "Error Code": 500
    }
  }
}

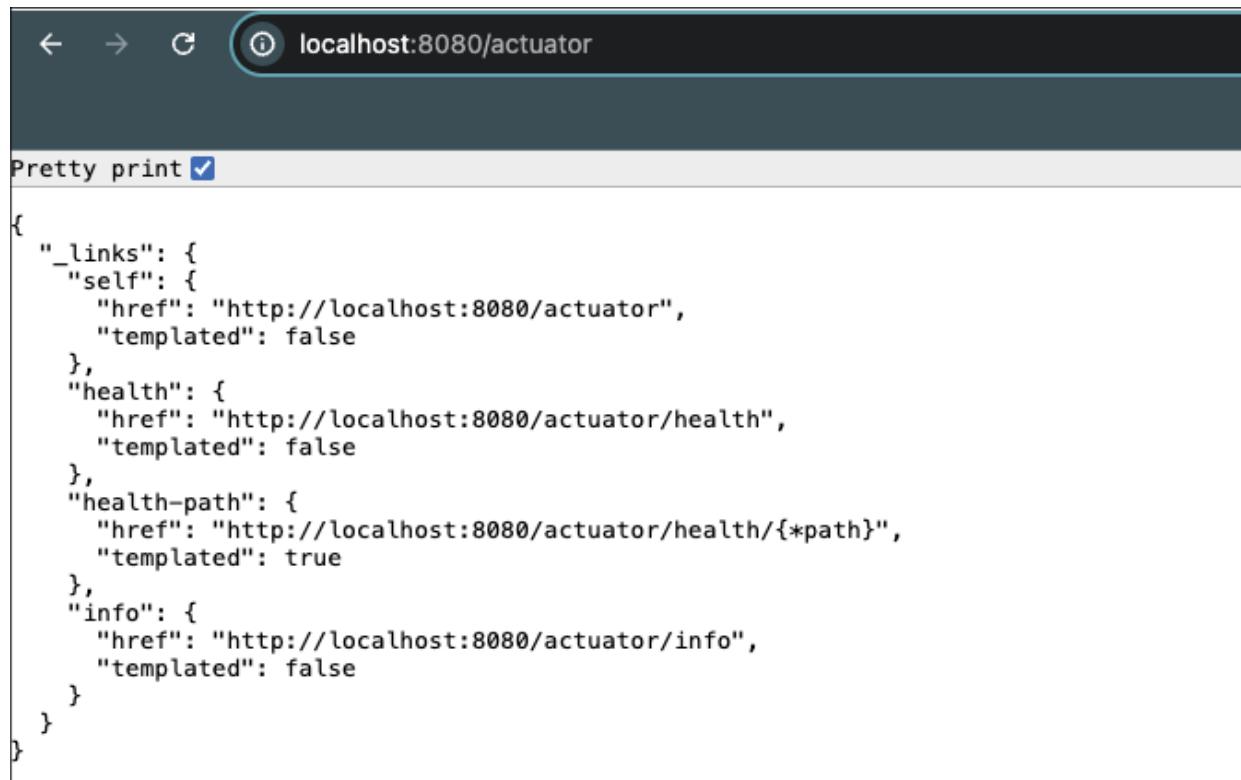
```

**Figure 3.15:** Spring Boot health endpoint with custom HealthIndicator

You can selectively enable required endpoints by using the same configuration parameter. Let us go ahead and add the following configuration to the **application.properties** file. This would only enable health and info endpoints.

```
1. management.endpoints.web.exposure.include=health,info
```

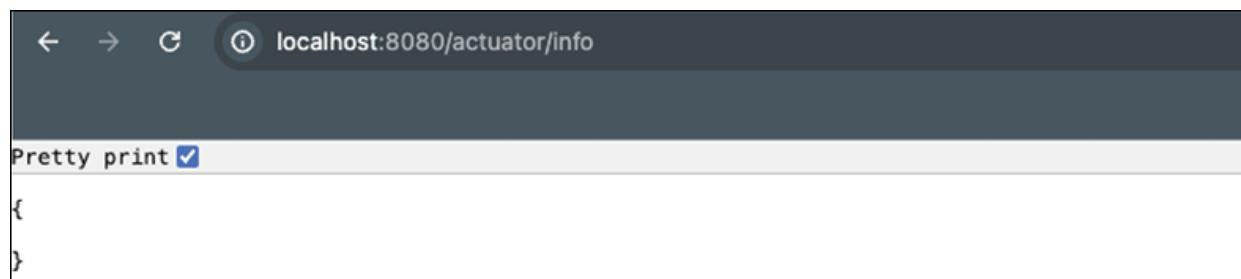
You can verify this configuration by visiting the root level actuator URL as shown in the following figure:



```
Pretty print   
{  
  "_links": {  
    "self": {  
      "href": "http://localhost:8080/actuator",  
      "templated": false  
    },  
    "health": {  
      "href": "http://localhost:8080/actuator/health",  
      "templated": false  
    },  
    "health-path": {  
      "href": "http://localhost:8080/actuator/health/{*path}",  
      "templated": true  
    },  
    "info": {  
      "href": "http://localhost:8080/actuator/info",  
      "templated": false  
    }  
  }  
}
```

*Figure 3.16: Spring Boot actuator endpoint with info*

We can also visit the newly added info URL using your browser:



```
Pretty print   
{  
}
```

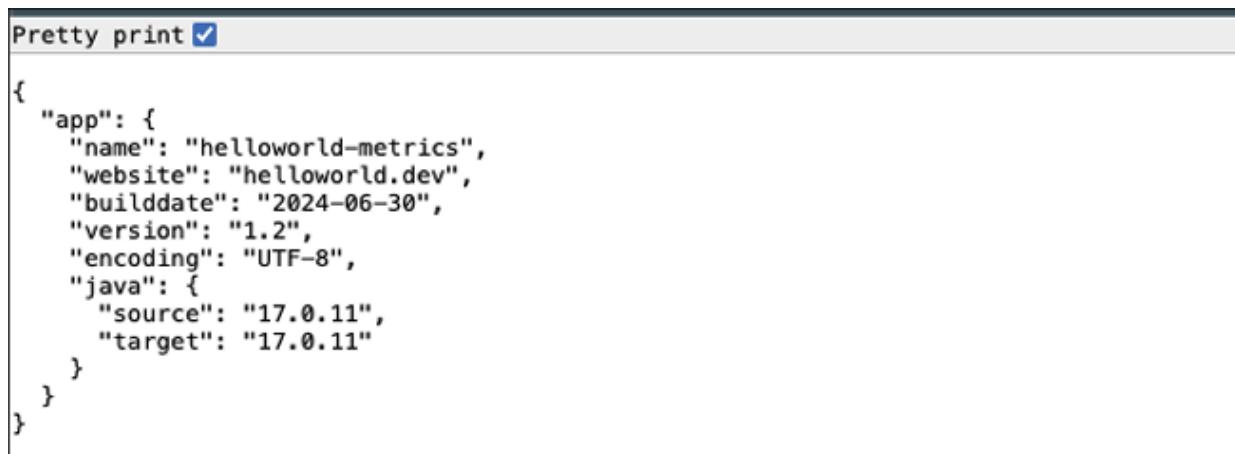
*Figure 3.17: Spring Boot info endpoint with default configuration*

As shown above, this would give you a successful result, but no application

information is available. To change this behavior, add the following content to the **application.properties** file:

1. `management.info.env.enabled=true`
2. `info.app.name=@project.artifactId@`
3. `info.app.website=helloworld.dev`
4. `info.app.builddate=2024-06-30`
5. `info.app.version=1.2`
6. `info.app.encoding=@project.build.sourceEncoding@`
7. `info.app.java.source=@java.version@`
8. `info.app.java.target=@java.version@`

This would show some of the information related to the application as follows:



The screenshot shows a JSON editor with the "Pretty print" checkbox checked. The JSON output is as follows:

```
{  
  "app": {  
    "name": "helloworld-metrics",  
    "website": "helloworld.dev",  
    "builddate": "2024-06-30",  
    "version": "1.2",  
    "encoding": "UTF-8",  
    "java": {  
      "source": "17.0.11",  
      "target": "17.0.11"  
    }  
  }  
}
```

*Figure 3.18: Spring Boot info endpoint with custom configuration*

So far, we have only seen the default health endpoint, but you can enable all the available endpoints by adding the following configuration **application.properties** file:

1. `management.endpoints.web.exposure.include=*`

With the above configuration, you should be able to see all the available endpoints, as shown in the following figure:

Pretty print

```
{
  "_links": {
    "self": {
      "href": "http://localhost:8080/actuator",
      "templated": false
    },
    "beans": {
      "href": "http://localhost:8080/actuator/beans",
      "templated": false
    },
    "caches": {
      "href": "http://localhost:8080/actuator/caches",
      "templated": false
    },
    "caches-cache": {
      "href": "http://localhost:8080/actuator/caches/{cache}",
      "templated": true
    },
    "health": {
      "href": "http://localhost:8080/actuator/health",
      "templated": false
    },
    "health-path": {
      "href": "http://localhost:8080/actuator/health/{*path}",
      "templated": true
    },
    "info": {
      "href": "http://localhost:8080/actuator/info",
      "templated": false
    },
    "conditions": {
      "href": "http://localhost:8080/actuator/conditions",
      "templated": false
    },
    "configprops": {
      "href": "http://localhost:8080/actuator/configprops",
      "templated": false
    },
    "configprops-prefix": {
      "href": "http://localhost:8080/actuator/configprops/{prefix}",
      "templated": true
    },
    "env-toMatch": {
      "href": "http://localhost:8080/actuator/env/{toMatch}",
      "templated": true
    },
    "env": {
      "href": "http://localhost:8080/actuator/env",
      "templated": false
    }
  }
}.
```

*Figure 3.19: Spring Boot actuator endpoint with all the available endpoints*

The following table provides you with a summary of each of the endpoint and their purpose:

Endpoint	Description
<b>info</b>	Provides various pieces of information about the application such as application name and version, build and deployment information.
<b>health</b>	Provides basic health information about the application, such as its status and additional details about its components.

<b>metrics</b>	Provides detailed metrics about the performance and resource usage which can be used for monitoring and diagnostics.
<b>env</b>	Provides the current environment properties of the application, including configuration properties, system properties, and environment variables.
<b>loggers</b>	Provides information about configured loggers, their log levels, and support runtime adjustment of log levels.
<b>beans</b>	Provides a complete list of all the Spring beans and detailed information about each bean, such as name, type, dependencies, and scope.
<b>conditions</b>	Provides detailed information about the configuration conditions that were evaluated during application start-up.
<b>shutdown</b>	Provides gracefully shutting down the application remotely via an HTTP POST request.
<b>configprops</b>	Provides a list of all configuration properties available in the application in the form of both the configured values and default value.
<b>httptrace</b>	Provides HTTP trace information for recent requests to the application. It captures details like HTTP method, URI, timestamps, headers, and response status.
<b>sessions</b>	Provides information about active HTTP sessions in the application. It displays details such as session IDs, creation times, last accessed times, and session attributes.
<b>threaddump</b>	Provides a snapshot of the current state of all threads in the JVM. It displays information such as thread names, states, stack trace.
<b>heapdump</b>	Provides a snapshot of the Java heap memory of the running application. It generates a binary heap dump file that can be analyzed to diagnose memory-related issues.
<b>auditevents</b>	Provides a list of security-related events that have occurred in the application.

**Table 3.1 : Spring Boot actuator endpoints**

Previously, we looked at how we can enrich the health endpoint with our own information; now, let us look at how we can implement our own endpoint. Create a new class called **HelloEndpoint.java** and add the following code segment:

1. `package com.bpb.hssb.ch3.helloworld.web;`
2.
3. `import org.springframework.boot.actuate.endpoint.annotation.Endpoint;`
4. `import org.springframework.boot.actuate.endpoint.annotation.ReadOperation;`
5. `import org.springframework.context.annotation.Bean;`

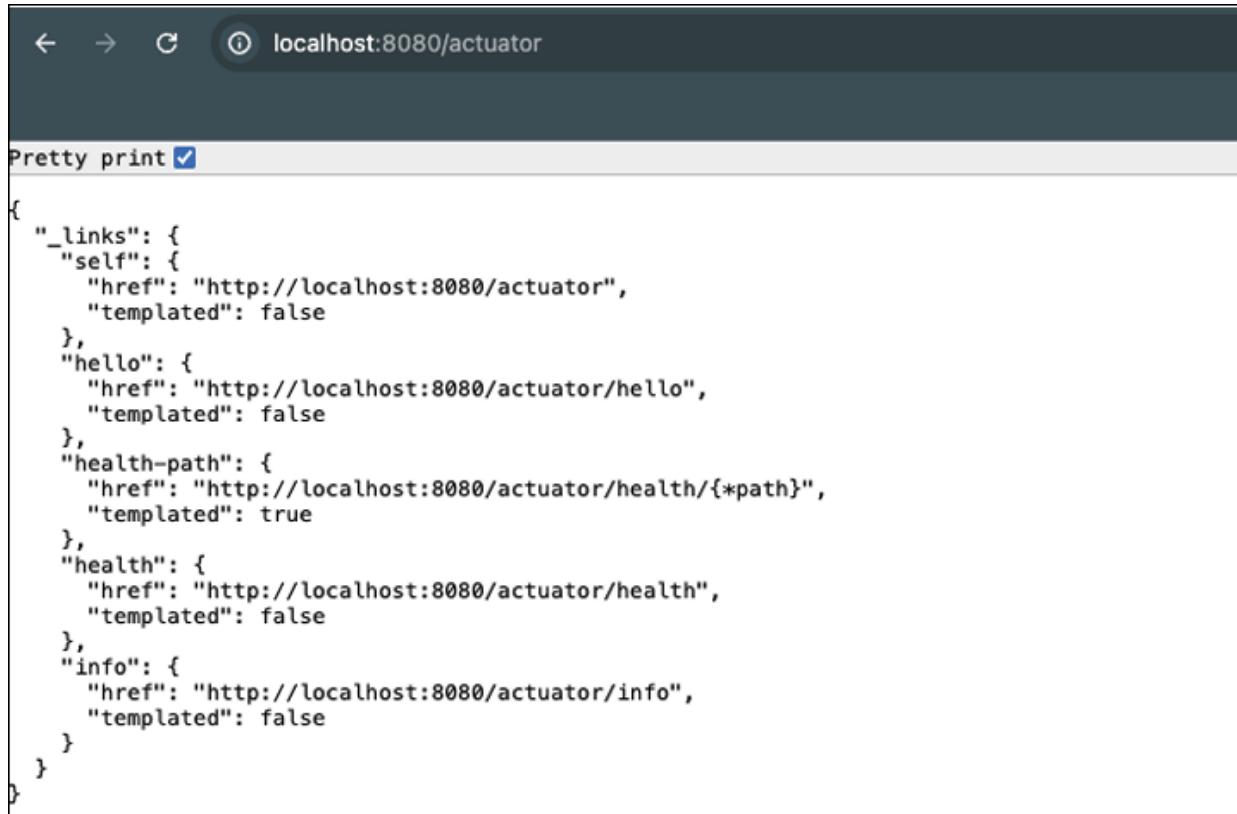
```
6. import org.springframework.stereotype.Component;  
7.  
8. @Endpoint(id = "hello")  
9. @Component  
10. public class HelloEndpoint {  
11.  
12.     @ReadOperation  
13.     @Bean  
14.     public String hello() {  
15.         return «HelloEndpoint is UP»;  
16.     }  
17.  
18. }
```

In the above code, the **HelloEndpoint** class is annotated with **@Endpoint** annotation, which indicates this is an actuator endpoint that provides information. Also, we registered the **hello** as the endpoint name. Then, on *line 12*, we indicated that the hello method is responsible for providing the read operation of the endpoint. The other annotations are already familiar to you at this stage.

After adding the above class, you can enable the above endpoint in the **application.properties** file unless you have enabled all the endpoints. For example, you may use the following configuration:

```
1. management.endpoints.web.exposure.include=health,info,hello
```

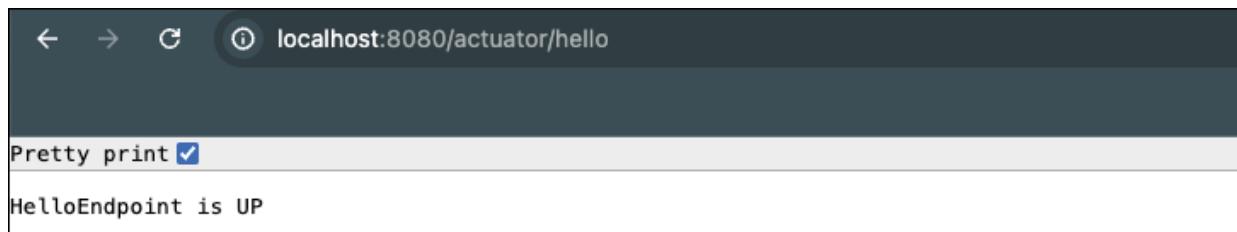
At this point, if you visit the actuator endpoint, you should be able to see that our hello endpoint has been listed as follows.



```
pretty print   
{  
  "_links": {  
    "self": {  
      "href": "http://localhost:8080/actuator",  
      "templated": false  
    },  
    "hello": {  
      "href": "http://localhost:8080/actuator/hello",  
      "templated": false  
    },  
    "health-path": {  
      "href": "http://localhost:8080/actuator/health/{*path}",  
      "templated": true  
    },  
    "health": {  
      "href": "http://localhost:8080/actuator/health",  
      "templated": false  
    },  
    "info": {  
      "href": "http://localhost:8080/actuator/info",  
      "templated": false  
    }  
  }  
}
```

Figure 3.20: Spring Boot actuator endpoint with a custom endpoint

We can also visit our **hello** endpoint and should be able to see the following output:



```
pretty print   
HelloEndpoint is UP
```

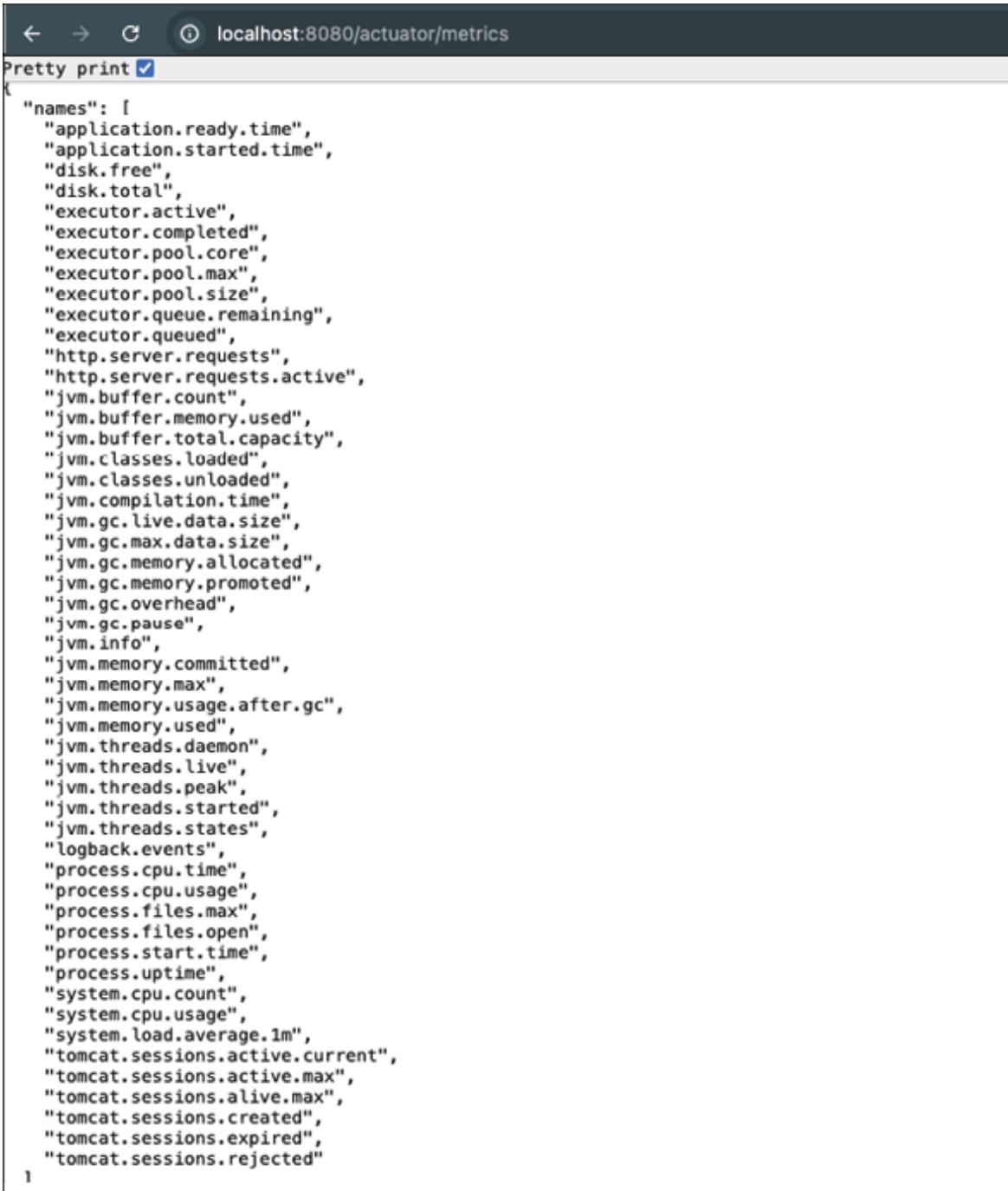
Figure 3.21: The hello custom endpoint

## Spring Boot metrics

So far, we have seen how to add and configure Spring Boot actuators to receive health and application information; in this section, let us look at how to enable and use metrics. First, add the following configuration to the **application.properties** files to enable the metrics endpoint:

1. `management.endpoints.web.exposure.include=health,info,metrics`

Now, if you visit the metrics endpoint, you should be able to see all the available metrics, as shown in the following figure:

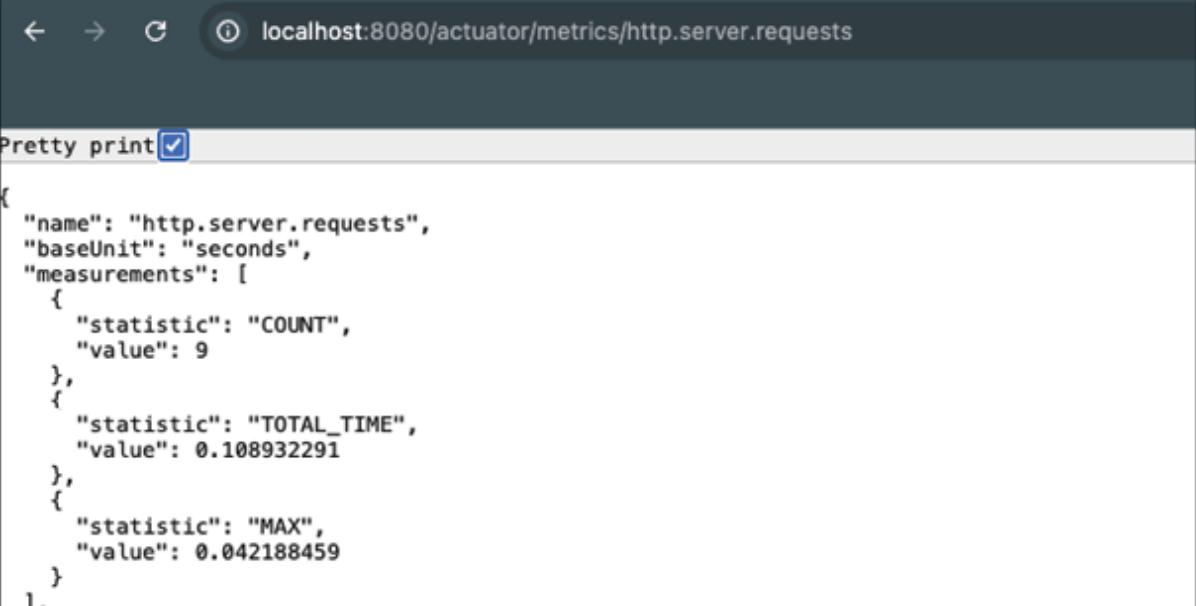


```
localhost:8080/actuator/metrics
Pretty print 
{
  "names": [
    "application.ready.time",
    "application.started.time",
    "disk.free",
    "disk.total",
    "executor.active",
    "executor.completed",
    "executor.pool.core",
    "executor.pool.max",
    "executor.pool.size",
    "executor.queue.remaining",
    "executor.queued",
    "http.server.requests",
    "http.server.requests.active",
    "jvm.buffer.count",
    "jvm.buffer.memory.used",
    "jvm.buffer.total.capacity",
    "jvm.classes.loaded",
    "jvm.classes.unloaded",
    "jvm.compilation.time",
    "jvm.gc.live.data.size",
    "jvm.gc.max.data.size",
    "jvm.gc.memory.allocated",
    "jvm.gc.memory.promoted",
    "jvm.gc.overhead",
    "jvm.gc.pause",
    "jvm.info",
    "jvm.memory.committed",
    "jvm.memory.max",
    "jvm.memory.usage.after.gc",
    "jvm.memory.used",
    "jvm.threads.daemon",
    "jvm.threads.live",
    "jvm.threads.peak",
    "jvm.threads.started",
    "jvm.threads.states",
    "logback.events",
    "process.cpu.time",
    "process.cpu.usage",
    "process.files.max",
    "process.files.open",
    "process.start.time",
    "process.uptime",
    "system.cpu.count",
    "system.cpu.usage",
    "system.load.average.1m",
    "tomcat.sessions.active.current",
    "tomcat.sessions.active.max",
    "tomcat.sessions.alive.max",
    "tomcat.sessions.created",
    "tomcat.sessions.expired",
    "tomcat.sessions.rejected"
  ]
}
```

*Figure 3.22: The Spring Boot metrics endpoint*

We are not planning to discuss each of the metrics listed above, but for an example, let us visit the matrix, and you will see something similar to the

one shown in the following figure:



```
pretty print  
{  
  "name": "http.server.requests",  
  "baseUnit": "seconds",  
  "measurements": [  
    {  
      "statistic": "COUNT",  
      "value": 9  
    },  
    {  
      "statistic": "TOTAL_TIME",  
      "value": 0.108932291  
    },  
    {  
      "statistic": "MAX",  
      "value": 0.042188459  
    }  
  ],  
}
```

*Figure 3.23: The HTTP requests data from the metrics endpoint*

The above results show the total number of HTTP requests served by the server. You can experiment by sending a couple of more HTTP requests and see how this count varies. The metric analytics tools would use these metrics to show the historical trends and the current status of the application. In addition, these types of metrics can be used to trigger alerts when reaching a predefined threshold limit.

Under the hood, Spring Boot uses Micrometer, which acts as a vendor-neutral application metrics facade, allowing Spring Boot to collect metrics from the JVM and send them to various monitoring systems. When you add the **spring-boot-starter-actuator** dependency, Spring Boot automatically configures Micrometer for you. The main advantage of using Micrometer is that you can switch between monitoring systems such as Prometheus, Graphite, and DataDog without changing the metrics code.

At this stage, let us look at how we can support Prometheus metrics in our application. Prometheus is a widely used open-source metrics monitoring and alerting solution. In order to enable necessary auto-configuration, you need to add the following dependency into the application:

1. <dependency>
2. <groupId>io.micrometer</groupId>

3. <artifactId>micrometer-registry-prometheus</artifactId>
4. </dependency>

The next step is to enable the Prometheus endpoint by adding the following configuration to the **application.properties** file:

1. `management.endpoints.web.exposure.include=health,info,prometheus`

Now, if you visit the newly added Prometheus endpoint, you will be able to see the metrics in the format supported by Prometheus. This is just a small portion of the results set generated by the Prometheus endpoint:

```
# HELP application_ready_time_seconds Time taken for the application to be ready to service requests
# TYPE application_ready_time_seconds gauge
application_ready_time_seconds{main_application_class="com.bpb.hssb.ch3.helloworld.web.HelloworldWebApplication"} 1.174
# HELP application_started_time_seconds Time taken to start the application
# TYPE application_started_time_seconds gauge
application_started_time_seconds{main_application_class="com.bpb.hssb.ch3.helloworld.web.HelloworldWebApplication"} 1.169
# HELP disk_free_bytes Usable space for path
# TYPE disk_free_bytes gauge
disk_free_bytes{path="/Users/sagara/Dev/gdrive/Book-Spring6/code/ch3/helloworld-metrics/."} 3.2909176832E10
# HELP disk_total_bytes Total space for path
# TYPE disk_total_bytes gauge
disk_total_bytes{path="/Users/sagara/Dev/gdrive/Book-Spring6/code/ch3/helloworld-metrics/."} 4.94384795648E11
# HELP executor_active_threads The approximate number of threads that are actively executing tasks
# TYPE executor_active_threads gauge
executor_active_threads{name="applicationTaskExecutor"} 0.0
# HELP executor_completed_tasks_total The approximate total number of tasks that have completed execution
# TYPE executor_completed_tasks_total counter
executor_completed_tasks_total{name="applicationTaskExecutor"} 0.0
# HELP executor_pool_core_threads The core number of threads for the pool
# TYPE executor_pool_core_threads gauge
executor_pool_core_threads{name="applicationTaskExecutor"} 8.0
# HELP executor_pool_max_threads The maximum allowed number of threads in the pool
# TYPE executor_pool_max_threads gauge
executor_pool_max_threads{name="applicationTaskExecutor"} 2.147483647E9
# HELP executor_pool_size_threads The current number of threads in the pool
# TYPE executor_pool_size_threads gauge
executor_pool_size_threads{name="applicationTaskExecutor"} 0.0
# HELP executor_queue_remaining_tasks The number of additional elements that this queue can ideally accept without blocking
# TYPE executor_queue_remaining_tasks gauge
executor_queue_remaining_tasks{name="applicationTaskExecutor"} 2.147483647E9
```

*Figure 3.24: Spring Boot logs printed in the console*

## Spring Boot logging

Spring Boot uses **commons logging** for all internal logging; commons logging acts as an abstraction layer, providing a common interface for various logging implementations such as Log4j, Java Util Logging, and Logback. Default configurations are provided for each logging implementation. In each case, loggers are pre-configured to use console output with optional file output also available. You have already seen logging-in action starting from our first application. Unless you have specific requirements, such as integrating with an existing application that uses a different logging framework, most of the time, the default logging features enabled by Spring Boot are enough for your applications. You can control the logging behavior using the **application.properties** file.

To refresh our memory, let us look at the following logs printed to the console:



```
:: Spring Boot :: (v3.3.0)
2024-06-30T16:27:16.425+12:00 INFO 27795 --- [helloworld-web] [main] c.b.h.c.h.web.HelloworldWebApplication : Starting HelloworldWebApplication using Java 17.0.2 with PID 27795 (/Users/sagara/Dev/gdrive/Book-Spring6/code/ch3/helloworld-logging)
2024-06-30T16:27:16.427+12:00 INFO 27795 --- [helloworld-web] [main] c.b.h.c.h.web.HelloworldWebApplication : No active profile set, falling back to 1 default profile: "default"
2024-06-30T16:27:16.428+12:00 INFO 27795 --- [helloworld-web] [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port 8080 (http)
2024-06-30T16:27:16.429+12:00 INFO 27795 --- [helloworld-web] [main] o.s.b.w.embedded.tomcat.TomcatWebServer : StandardService[Tomcat]: Starting
2024-06-30T16:27:16.430+12:00 INFO 27795 --- [helloworld-web] [main] o.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/10.1.24]
2024-06-30T16:27:16.431+12:00 INFO 27795 --- [helloworld-web] [main] o.a.c.c.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2024-06-30T16:27:16.432+12:00 INFO 27795 --- [helloworld-web] [main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 461 ms
2024-06-30T16:27:16.433+12:00 INFO 27795 --- [helloworld-web] [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 8080 (http) with context path '/'
2024-06-30T16:27:17.059+12:00 INFO 27795 --- [helloworld-web] [main] c.b.h.c.h.web.HelloworldWebApplication : Started HelloworldWebApplication in 0.86 seconds (process running for 1.119)
```

*Figure 3.25: Spring Boot logs printed in the console*

This is what we can see in *Figure 3.25*:

- Log entries are sorted based on date and time
- Log levels, which are set to INFO at the moment
- Process IDs
- Application name as defined in `spring.application.name` property
- Thread name which main in our sample
- The content of the log message

Now, let us play around with the logging configuration a bit so that you can understand how you can configure the logging configuration easily. For instance, adding the following configuration into the **application.properties** file creates a log file in the log directory in addition to console logs and configures the log level for **com.bpb.hssb.ch3.helloworld.web** package into **debug**, in practical scenarios, you would usually set your application packages to debug level from time to time in order to debug the issues.

1. `logging.file.name=./log/app.log`
2. `logging.level.root=info`
3. `logging.level. com.bpb.hssb.ch3.helloworld.web =debug`

To see some debug outputs in the console, we need to add a couple of debug log messages into our code as shown in the following code listing:

1. `package com.bpb.hssb.ch3.helloworld.web;`
- 2.
3. `import java.time.LocalDateTime;`
- 4.
5. `import org.slf4j.Logger;`
6. `import org.slf4j.LoggerFactory;`
7. `import org.springframework.stereotype.Component;`

```

8.
9. @Component
10. public class GreetingService {
11.
12.     Logger logger = LoggerFactory.getLogger(GreetingService.class);
13.
14.     public String greet() {
15.         LocalTime now = LocalTime.now();
16.         int currentHour = now.getHour();
17.         String msg = "Hello world";
18.         logger.debug("The currnt hour of the day : {}", currentHour);
19.         logger.debug("Message prefix : {}", msg);
20.         if (currentHour < 12) {
21.             return msg + ", it's a wonderful morning!";
22.         } else {
23.             return msg + ", it's a wonderful afternoon!";
24.         }
25.     }
26. }
27.

```

The following points provide a summary of the log management capabilities supported by Spring Boot:

- Logback is the default logging framework, and the ability to configure Log4J and Java Util Logging as alternative logging implantations.
- Auto-configuration support for logging implantations.
- Ability to configure log levels using configuration files and system properties.
- Ability to define custom log patterns.
- Profile-specific logging.
- Log file management features such as log file name patterns and rotation configurations.
- Integration with Spring Boot actuator to control log levels during the runtime.

## Spring Boot tracing

Similar to metrics, the Spring Boot actuator provides dependency management and auto-configuration for Micrometer Tracing, which is a facade for popular tracer libraries, and Spring Boot ships auto-configuration and supports OpenTelemetry and OpenZipkin Brave as tracers. The tracing platforms such as Zipkin, Wavefront, and OTLP can be used with Spring Boot quite easily. In the chapters, we are not trying to discuss end-to-end distributed tracing; instead, we are only discussing the basics of supporting tracing in our samples.

We will use the same application we used before because, as you might remember, we introduced some logging messages into the application code, which would be useful in this section. Before we go further, please pay attention to the following log messages printed in the console when invoking the hello endpoint of this application:

```
2024-06-30T23:05:13.923+13:00 DEBUG 14941 --- [helloworld-web] [nio-8080-exec-6]
[67bc4459d200ecdcfde61e6cf50b94a9-fde61e6cf50b94a9] c.b.h.c.helloworld.web.HelloController :
processing an incoming request
2024-06-30T23:05:13.923+13:00 DEBUG 14941 --- [helloworld-web] [nio-8080-exec-6]
[67bc4459d200ecdcfde61e6cf50b94a9-fde61e6cf50b94a9] c.b.h.c.helloworld.web.GreetingService :
The currnt hour of the day : 23
2024-06-30T23:05:13.924+13:00 DEBUG 14941 --- [helloworld-web] [nio-8080-exec-6]
[67bc4459d200ecdcfde61e6cf50b94a9-fde61e6cf50b94a9] c.b.h.c.helloworld.web.GreetingService :
Message prefix : Hello world
```

As we did in the metrics section, the first step would be to add the **spring-boot-starter-actuator** as a dependency in our application. That can be done by adding the following entry to the POM file of our application:

1. <dependency>
2. <groupId>org.springframework.boot</groupId>
3. <artifactId>spring-boot-starter-actuator</artifactId>
4. </dependency>

Then, let us go ahead and add micrometer-tracing-bridge-brave as a dependency by adding these lines.

Now, if you rerun the application and invoke the hello endpoint, you will still see the same debug entries in the application console but with one additional data element, as shown in the following figure:

```
2024-06-30T23:11:29.942+13:00 INFO 18239 --- [helloworld-web] [nio-8080-exec-1]
[67bc45d1da809fbfdf8d0fefd3a0aa1-def8d0fefd3a0aa1] c.b.h.c.helloworld.web.HelloController :
```

Say Hello along with trace Id

```
2024-06-30T23:11:29.943+13:00 INFO 18239 --- [helloworld-web] [nio-8080-exec-1]
[67bc45d1da809fbfdef8d0fefd3a0aa1-def8d0fefd3a0aa1] c.b.h.c.helloworld.web.HelloController :
127.0.0.1
```

The additional randomly generated string printed in the console is known as the correlation ID, and that is the fundamental building block of tracing. If you closely observe the above console, you will be able to see that this correlation ID remains the same even though we have printed debug messages from several components of our application, under the hood, Spring Boot auto-configuration makes sure to constantly propagate the same correlation ID across internal components. If you send another request to the hello endpoint, you will be able to see a new correlation ID uniquely associated with the new request processing. In tracing tools, this correlation ID is used to uniquely identify each transaction.

Now, you may have a valid question: we managed to maintain a correlation ID within the same application, so how could we propagate the same ID to the downstream services or applications? For this purpose, Spring Boot uses the transparent HTTP header defined by Micrometer to propagate trace IDs across services. When you use client utilities such as **RestTemplateBuilder**, you can automatically pass the transparent HTTP header to the other applications. We conclude our discussion related to the tracing at this stage, and we will revisit it again later.

## Conclusion

In this chapter, we looked at TDD and how to use the Spring test module and mocking capabilities. We learned the fundamentals of Spring Security and applied security for an application. We also looked at enabling and using actuators and other observability features in your applications.

In the next chapter, we will explore the Spring MVC framework and try to build a web application using Spring MVC.

## Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



*OceanofPDF.com*

# CHAPTER 4

# Building Spring MVC Web Applications

## Introduction

This chapter will introduce MVC architecture and the Spring MVC, explaining the concepts of Model, View, and Controller separately. We will briefly discuss templating technologies such as Thymeleaf. We will also introduce the sample use case that we will use throughout this book and will move into implementing the use case using Spring MVC. Following this, we will look into the practical aspects of handling errors and error pages and input validation. Finally, we will discuss how to apply security for the sample application using Spring Security.

## Structure

The chapter covers the following topics:

- Introduction to Spring MVC
- Writing your first Spring MVC web application
- BookClub use case
- Data validation
- Error handling
- Securing Spring MVC applications

- Observability

## Objectives

By the end of this chapter, you will understand the key features of Spring MVC. You will also learn about the fundamentals required to implement a web application using Spring MVC. In addition to that, we will discuss how to secure your web application using Spring Security and apply observability features.

## Introduction to Spring MVC

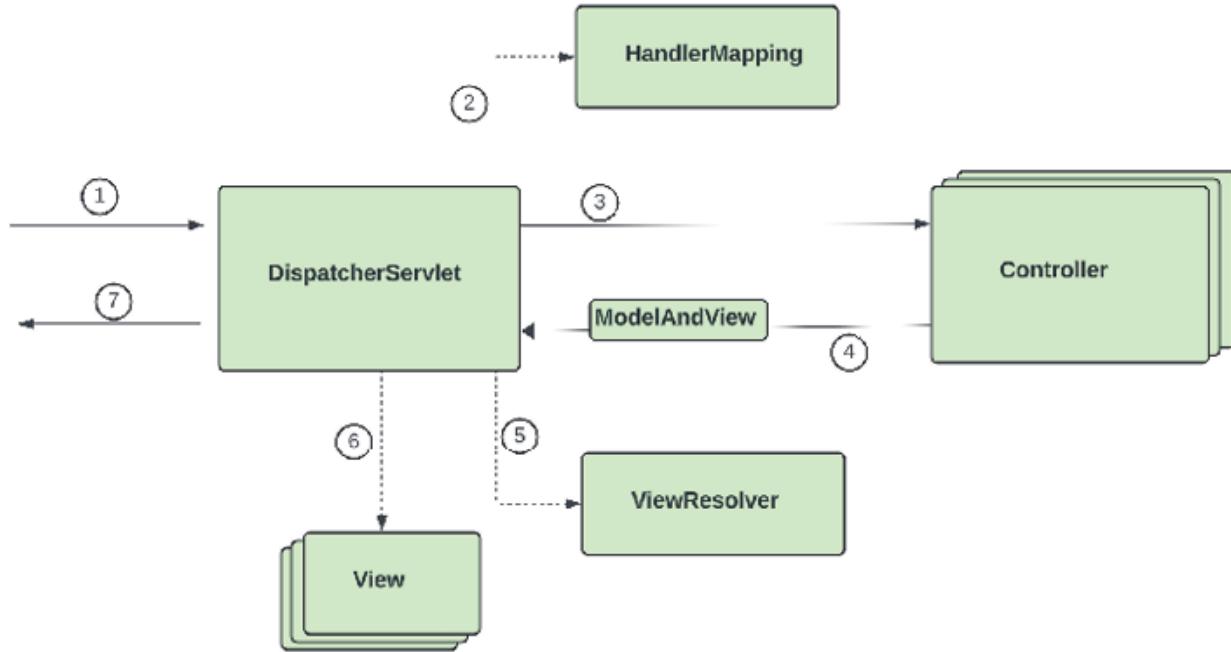
The Spring MVC, or Spring Web MVC, is a component of the Spring Framework designed for building web applications using the traditional MVC design pattern. This pattern divides an application into three logical components: Model, View, and Controller, promoting separation of concerns to simplify application complexity.

In the MVC design pattern:

- The Model represents the data and encapsulates the application's state.
- The View is responsible for presenting data to the user by rendering the user interfaces according to changes in the model.
- The Controller acts as an intermediary between the model and the view, handling data processing and business logic.

Beyond developing web applications with user interfaces, Spring MVC also supports creating RESTful web services. Each request in Spring MVC is handled by a dedicated thread based on the Java Servlet architecture. Spring MVC applications require a Servlet container to run, which can be either an embedded container or an external server. Additionally, Spring MVC supports internationalization and localization, providing core web application development capabilities.

The architecture of Spring MVC is quite simple. Before we write our first Spring MVC application, let us look at the high-level architecture shown in the following figure and try to understand the request processing:



*Figure 4.1: Spring MVC architecture*

The following steps describe request processing flows shown in [Figure 4.1](#):

1. During the first step, all incoming requests are intercepted by **DispatcherServlet**, acting as the front controller.
2. The **DispatcherServlet** looks for **HandlerMapping** to identify the correct Controller to handle the current request.
3. Then, the **DispatcherServlet** forwards the request to the correct Controller identified in the previous step for request processing
4. The specific Controller processes the incoming message and executes the business logic, such as querying a database or calling an external API. Once it is ready to return, it creates and returns a **ModelAndView** to the **DispatcherServlet**.
5. During this step, the **DispatcherServlet** contacts the **ViewResolver** to determine the appropriate view based on the **ModelAndView** returned by the Controller in the previous step.
6. Once the appropriate view has been identified, the **DispatcherServlet** renders the view with the data.
7. As the final step, the response message is returned to the user's browser, and the view addresses the preparation of data before handing it over to a specific view technology.

The **DispatcherServlet** is associated with a special **ApplicationContext** known as **WebApplicationContext**, which contains all the beans required for processing, including the special types of beans we looked at earlier, such as **HandlerMapping** and **ViewResolver**, as well as the actual Controllers responsible for request processing.

One important point to highlight here is that compared to other building blocks of Spring MVC, the ViewResolver and View interfaces are designed in a pluggable manner without being specifically bound to any viewing technology. Spring MVC supports many viewing technologies, some of which are briefly discussed in the following list. You can configure your preferred viewing technology using the Spring configuration.

- **Thymeleaf:** Thymeleaf is a popular server-side template engine for Java. Thymeleaf uses HTML that can be correctly displayed in browsers while also functioning as a static prototype. In addition to HTML, Thymeleaf can process HTML, XML, JavaScript, and CSS. If you are familiar with basic HTML syntax, it is not hard to learn Thymeleaf; due to this reason, Thymeleaf is quite popular among Spring developers, and we will use Thymeleaf as the viewing technology in the samples of this book.
- **FreeMarker:** FreeMarker is another popular template engine for Java and supports HTML output format.
- **Groovy Markup:** Spring MVC also supports a template engine called Groovy Markup Template Engine by Groovy. This would be an excellent choice if you are familiar with Groovy.
- **Java scripting engine (JSR 223):** Spring MVC supports templating libraries that can run on top of the Java scripting engine (JSR-223), such as Handlebars, Mustache, and Kotlin.
- **JSP and JSTL:** You can use JSP and JSTL as viewing technologies. This was a popular choice in the past.

In addition to these viewing technologies, you can also use Spring MVC to generate RSS, ATOM, PDF, and Excel content.

## Writing your first Spring MVC web application

As we have covered the Spring MVC architecture, let us proceed to build

our first Spring MVC application. Begin by navigating to Spring Initializr either through your browser or IDE, and generate an application template using the following details:

**Project : Maven**

**Language : Java**

**Spring Boot : Latest stable version ( e.g. – 3.3.0)**

**Project Metadata :**

**Group : com.bpb.hssb.ch4**

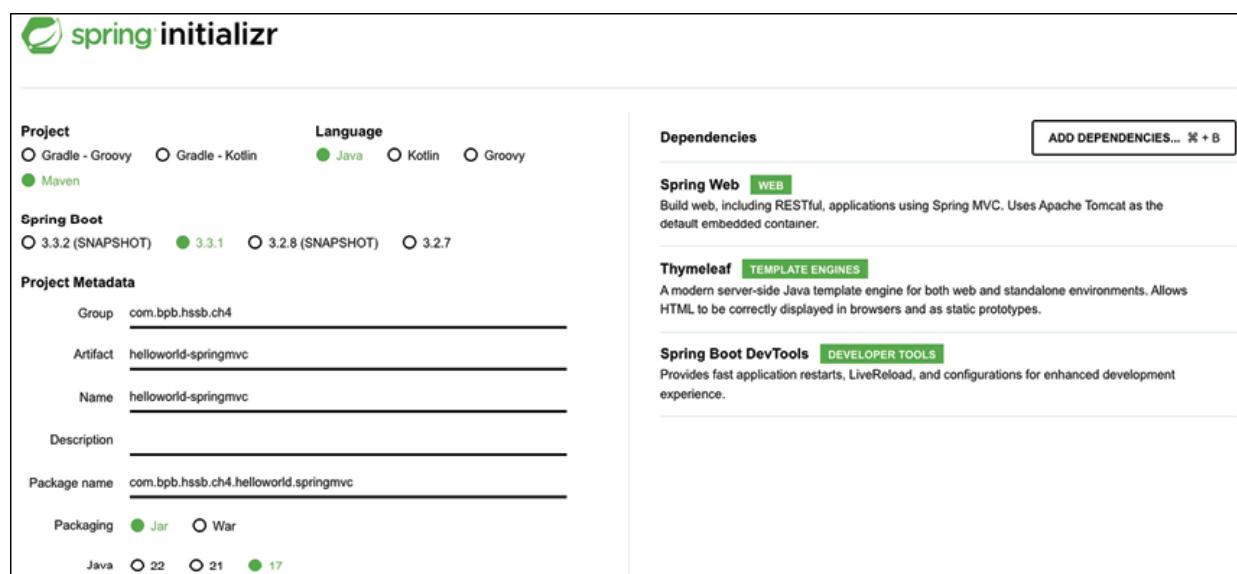
**Artifact : helloworld- springmvc**

**Packaging : Jar (Default)**

**Java : 17 (Default)**

**Dependencies: Spring Web, Thymeleaf, Spring Boot DevTools**

You should see a Spring Initializr configuration similar to the following figure:



*Figure 4.2: Spring Initializr*

If you open the POM file of the generated project, you will notice the following dependencies:

1. <dependencies>
2. <dependency>
3. <groupId>org.springframework.boot</groupId>
4. <artifactId>spring-boot-starter-thymeleaf</artifactId>
5. </dependency>

```

6. <dependency>
7.   <groupId>org.springframework.boot</groupId>
8.   <artifactId>spring-boot-starter-web</artifactId>
9. </dependency>
10.
11. <dependency>
12.   <groupId>org.springframework.boot</groupId>
13.   <artifactId>spring-boot-devtools</artifactId>
14.   <scope>runtime</scope>
15.   <optional>true</optional>
16. </dependency>
17. <dependency>
18.   <groupId>org.springframework.boot</groupId>
19.   <artifactId>spring-boot-starter-test</artifactId>
20.   <scope>test</scope>
21. </dependency>
22. </dependencies>

```

**Tip:** Spring Boot DevTools is a set of tools designed to enhance the development experience. It includes features such as automatic restarts whenever files on the classpath change and live reload, which triggers a browser refresh when a resource is updated. Additionally, it sets sensible development-time defaults and disables caching. Spring Boot DevTools aims to speed up the development cycle by reducing the need for manual restarts and enabling immediate reflection of code changes.

Then, create a new Java class called **HelloController.java** and add the following code:

```

1. package com.bpb.hssb.ch4.helloworld.springmvc;
2.
3. import org.springframework.stereotype.Controller;
4. import org.springframework.ui.Model;
5. import org.springframework.web.bind.annotation.GetMapping;
6. import org.springframework.web.bind.annotation.RequestParam;
7.
8. @Controller
9. public class HelloController {
10.
11.   @GetMapping("/hello")
12.   public String hello(@RequestParam(name = "name", defaultValue = "World") String name,
13.                      Model model) {
14.
15.     model.addAttribute("name", name);
16.     return <<hello></>;
17.   }

```

18. }

On *line 8*, we have annotated our **HelloController** class with the **@Controller** annotation. The **@Controller** annotation is a specialization of the **@Component** annotation, indicating that the annotated class acts as a controller in Spring MVC. During the Spring auto-scanning process, classes annotated with **@Controller** are detected and registered with the **WebApplicationContext**. These annotated classes contain one or more handling methods associated with request paths and HTTP methods.

Then, on *line 11*, we annotated the hello world method with the **@GetMapping** annotation. The **@GetMapping** annotation indicates that this method is responsible for handling HTTP GET requests with the specified path or path pattern, in our case, HTTP GET requests to the **/hello** path.

On *line 12*, we have annotated a method parameter with the **@RequestParam** annotation. This annotation ensures that if the request URL contains a query string called **name**, it is passed as the value of the name parameter. If no query string named "**name**" is present in the request, the default value specified in the annotation will be used.

Additionally, we have added a special parameter called **model** to this method. Spring automatically injects a **model** object into the method. On *line 15*, we use this **model** object to set the value we received from the request query string as an attribute called **name**. These attributes will be used during view rendering. Finally, on *line 16*, we return a **String** value, which we call the view name.

Before discussing view aspects, let us run our application to see the response when calling the **HelloController**. You can run **HelloworldSpringmvcApplication** from your IDE or using Maven. If everything is correct so far, you should see the result as shown in the following figure:

## Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

```
Sun Jul 14 20:05:49 NZST 2024
There was an unexpected error (type=Internal Server Error, status=500).
Error resolving template [hello], template might not exist or might not be accessible by any of the configured Template Resolvers
org.thymeleaf.exceptions.TemplateInputException: Error resolving template [hello], template might not exist or might not be accessible by any of the configured Template Resolvers
at org.thymeleaf.engine.TemplateManager.resolveTemplate(TemplateManager.java:869)
at org.thymeleaf.engine.TemplateManager.parseAndProcess(TemplateManager.java:607)
at org.thymeleaf.TemplateEngine.process(TemplateEngine.java:1103)
at org.thymeleaf.TemplateEngine.process(TemplateEngine.java:1077)
at org.thymeleaf.spring6.view.ThymeleafView.renderFragment(ThymeleafView.java:372)
```

*Figure 4.3: Error page*

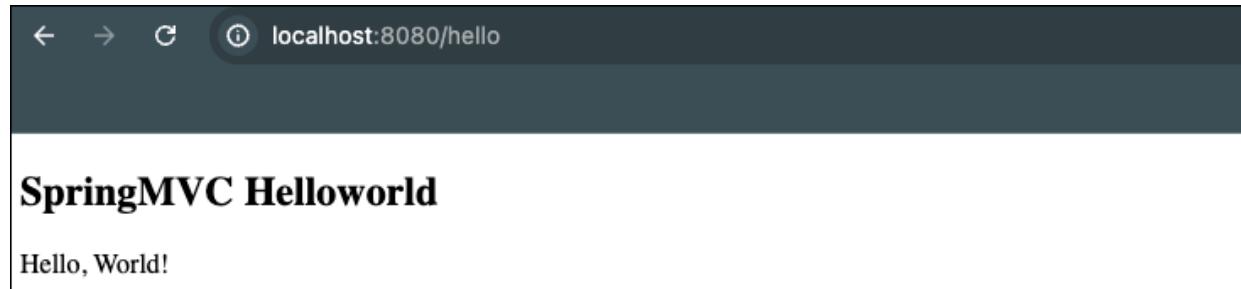
This result is not unexpected; we have defined a controller called **HelloController** that returns a view called **hello**, but we have not defined any view yet. The error message highlights that Spring MVC faced an issue resolving the template associated with the **hello** view.

As the next step, create a file called **hello.html** in the **src/main/resources/templates/** directory, which should already exist. In this example, we use Thymeleaf as the viewing technology. The initially added Thymeleaf Spring Boot starter takes care of configuring Thymeleaf as the viewing technology, so we only need to define the views. Add the following code to the **hello.html** file:

```
1. <!DOCTYPE HTML>
2. <html xmlns:th="http://www.thymeleaf.org">
3.
4. <head>
5.   <title>SpringMVC Helloworld</title>
6.   <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
7. </head>
8.
9. <body>
10.  <h2>SpringMVC Helloworld</h2>
11.  <p th:text="|Hello, ${name}!|" />
12. </body>
13.
14. </html>
```

As you can see, this is a very simple HTML file with some Thymeleaf elements. On *line 11*, we use a couple of Thymeleaf syntaxes to set the model attribute from the **HelloController** to the view. The **th:text** is a Thymeleaf attribute used to set the text content of an HTML element, and  **\${name}** is a variable expression in Thymeleaf, while **|** denotes a literal template in Thymeleaf.

Now, if you revisit the same URL, you will see the results as follows:



*Figure 4.4: Helloworld application*

At this stage, we have developed our first Spring MVC application to understand the architectural concepts discussed at the beginning of this section. Before proceeding further, let us look at the sample use case that we will use in the book.

## BookClub use case

The concept of BookClub originally started with a couple of developers sharing technical books among themselves. As the number of members grew, they decided to turn it into a start-up business called BookClub. One of their ongoing efforts is to create a book management system for the business.

We will use this use case throughout the book and build a part of the book management system in each chapter. To start with a simple system, as depicted in the following figure, they have defined their domain with two classes: **Book** and **BookCopy**.



*Figure 4.5: Main domain classes of the use case*

In the next section, we will implement BookClub as a Spring MVC web application. To maintain readability, we will only provide the most important code segments in the book, especially for Thymeleaf templates. You can

refer to the complete source code of the sample application in the book's companion GitHub repository.

## Building BookClub application

First, as we did with the previous example, go to Spring Initializr through your browser or IDE and generate an application template using the following details:

**Project : Maven**

**Language : Java**

**Spring Boot : Latest stable version ( e.g. – 3.3.0)**

**Project Metadata :**

**Group : com.bpb.hssb.ch4**

**Artifact : bookclub**

**Packaging : Jar (Default)**

**Java : 17 (Default)**

**Dependencies: Spring Web, Thymeleaf, Lombok, Spring Boot DevTools**

**Tip:** Lombok is a Java library designed to reduce boilerplate code by automatically generating common methods like getters, setters, and constructors through annotations. It can also implement the builder pattern and create full data classes with a single annotation. Lombok generates this boilerplate code during the compilation process by modifying the Abstract Syntax Tree (AST). Most Java IDEs, including VS Code, support Lombok.

Next, open the generated project in the VS Code IDE and start adding our domain classes. Following a convention used by Spring developers, create a package called **domain**. Create a **Book.java** file and add the following code:

```
1. package com.bpb.hssb.ch4.bookclub.domain;  
2.  
3. import java.util.List;  
4.  
5. import lombok.AllArgsConstructor;  
6. import lombok.Builder;  
7. import lombok.Getter;  
8. import lombok.NoArgsConstructor;  
9. import lombok.Setter;  
10.  
11. @Getter  
12. @Setter
```

```
13. @AllArgsConstructor
14. @NoArgsConstructor
15. @Builder
16. public class Book {
17.
18.     private String bookId;
19.     private String isbn;
20.     private String title;
21.     private String author;
22.     private int publicationYear;
23.     private List<BookCopy> copies;
24.
25. }
26.
```

The Book class consists of fields such as **title**, **author**, **isbn**, and **bookId** to uniquely track each book within the system, and also contains references to the copies of a book. We use Lombok annotations instead of explicitly defining getters, setters, and constructors to reduce the amount of code in the domain classes, but it is perfectly fine if you prefer to define them explicitly instead.

Next, create another class called **BookCopy.java** in the same domain package and add the following code:

```
1. package com.bpb.hssb.ch4.bookclub.domain;
2.
3. import lombok.AllArgsConstructor;
4. import lombok.Builder;
5. import lombok.Getter;
6. import lombok.NoArgsConstructor;
7. import lombok.Setter;
8. import lombok.ToString;
9.
10. @Getter
11. @Setter
12. @AllArgsConstructor
13. @NoArgsConstructor
14. @Builder
15. @ToString
16. public class BookCopy {
```

```
17.  
18. private String copyId;  
19. private Book book;  
20. private boolean isAvailable;  
21.  
22. }
```

The **BookCopy** class consists of **copyId**, **isAvailable** fields, and a reference to the particular book. Here, we also use Lombok annotations to define the getters, setters, and constructors.

The next step is to define a repository interface for our domain model. In Spring, the concept of a repository is an abstraction that provides a consistent interface for data access operations such as persisting and retrieving data records. In this chapter, we use an in-memory repository implementation, and in the next chapter, we will use the same repository interface to implement a repository backed by relational databases. Create a **BookRepository** interface with the following code:

```
1. package com.bpb.hssb.ch4.bookclub.repository;  
2.  
3. import java.util.Collection;  
4.  
5. import com.bpb.hssb.ch4.bookclub.domain.Book;  
6. import com.bpb.hssb.ch4.bookclub.domain.BookCopy;  
7.  
8. public interface BookRepository {  
9.  
10.    public Book createBook(Book book);  
11.  
12.    public Book getBook(String bookId);  
13.  
14.    public Collection<Book> getAllBooks();  
15.  
16.    public Book deleteBook(String bookId);  
17.  
18.    public BookCopy createBookCopy(Book book);  
19.  
20.    public BookCopy getBookCopy(String bookCopyId);  
21.  
22.    public BookCopy deleteBookCopy(String bookId, String bookCopy);  
23. }
```

Now, create an in-memory implementation of the above interface called **InMemoryBookRepository.java** with the following code:

```
1. package com.bpb.hssb.ch4.bookclub.repository;  
2.  
3. public class InMemoryBookRepository implements BookRepository {  
4.  
5. }
```

You can refer to the complete code of **InMemoryBookRepository** from the *Chapter 4* directory of the companion GitHub repository of this book. The most important implementation details related to the **InMemoryBookRepository** class are provided as a code segment as follows:

```
1.     private Map<String, Book> books = new HashMap<>();  
2.  
3.     public InMemoryBookRepository() {  
4.         populateBookData();  
5.     }  
6.  
7.     @Override  
8.     public Book createBook(Book book) {  
9.         String bookId = getBookId();  
10.        book.setBookId(bookId);  
11.        book.setCopies(new ArrayList<>());  
12.        return books.put(bookId, book);  
13.    }  
14.  
15.    @Override  
16.    public Book getBook(String bookId) {  
17.        return books.get(bookId);  
18.    }  
19.  
20.    @Override  
21.    public Collection<Book> getAllBooks() {  
22.        return books.values();  
23.    }  
24.  
25.    @Override  
26.    public Book deleteBook(String bookId) {
```

```

27.     return books.remove(bookId);
28. }
29.
30. @Override
31. public BookCopy createBookCopy(Book book) {
32.     BookCopy copy = BookCopy.builder().book(book).copyId(getBookCopyId(book)).isAvailable(true).build();
33.     book.getCopies().add(copy);
34.     return copy;
35. }
36.
37. @Override
38. public BookCopy deleteBookCopy(String bookId, String copyId) {
39.     Book book = books.get(bookId);
40.     if (book != null) {
41.         book.getCopies().stream()
42.             .filter(copy -> copy.getCopyId().equals(copyId))
43.             .findFirst()
44.             .ifPresent(book.getCopies()::remove);
45.     }
46.     return null;
47. }

```

In the code segment, we use a Java Map to keep the books and their copies in memory, with each book added to the map using the book ID as the key. Additionally, we call a method called **populateBookData** to load an initial set of books into the map.

For our sample application, implementing the **BookRepository** is sufficient as we do not have any separate business logic. However, considering future requirements, it is a good idea to implement a service layer for our application. To do this, create a **BookService.java** class with the following code:

```

1. package com.bpb.hssb.ch4.bookclub.service;
2.
3. import java.util.Collection;
4.
5. import com.bpb.hssb.ch4.bookclub.domain.Book;
6. import com.bpb.hssb.ch4.bookclub.domain.BookCopy;
7.
8. public interface BookService {

```

```
9.
10. public Book createBook(Book book) throws BookServiceException;
11.
12. public Book getBook(String bookId) throws BookServiceException;
13.
14. public Collection<Book> getAllBooks() throws BookServiceException;
15.
16. public void removeBook(String bookId) throws BookServiceException;
17.
18. public BookCopy createBookCopy(Book book)
   throws BookServiceException;
19.
20. public BookCopy getBookCopy(String bookCopyId)
   throws BookServiceException;
21.
22. public void removeBookCopy(String bookId, String copyId)
   throws BookServiceException;
23.
24. }
```

The definition of the **BookService** interface is similar to the **BookRepository** interface, with the notable difference being that **BookService** uses **BookServiceException** to manage exceptions. At this point, defining the **BookServiceException** as follows is sufficient:

```
1. package com.bpb.hssb.ch4.bookclub.service;
2.
3. public class BookServiceException extends Exception {
4.
5.     public BookServiceException() {
6.         super();
7.     }
8.
9.     public BookServiceException(String message) {
10.         super(message);
11.     }
12.
13. }
```

Now, we can add the **BookServiceImpl.java** class with the following code:

```
1. package com.bpb.hssb.ch4.bookclub.service;
2.
3. import java.util.Collection;
```

```
4.
5. import com.bpb.hssb.ch4.bookclub.domain.Book;
6. import com.bpb.hssb.ch4.bookclub.domain.BookCopy;
7. import com.bpb.hssb.ch4.bookclub.repository.BookRepository;
8.
9. public class BookServiceImpl implements BookService {
10.
11.     private BookRepository bookRepository;
12.
13.     public BookServiceImpl(BookRepository bookRepository) {
14.         this.bookRepository = bookRepository;
15.     }
16.
17.     @Override
18.     public Book createBook(Book book) throws BookServiceException {
19.         return bookRepository.createBook(book);
20.     }
21.
22.     @Override
23.     public Book getBook(String bookID) throws BookServiceException {
24.         return bookRepository.getBook(bookID);
25.     }
26.
27.     public Collection<Book> getAllBooks()
28.         throws BookServiceException {
29.         return bookRepository.getAllBooks();
30.     }
31.     @Override
32.     public void removeBook(String bookID)
33.         throws BookServiceException {
34.         bookRepository.deleteBook(bookID);
35.     }
36.     @Override
37.     public BookCopy createBookCopy(Book book)
38.         throws BookServiceException {
39.         return bookRepository.createBookCopy(book);
40.
41.     @Override
```

```

42. public void removeBookCopy(String bookId, String copyId)
43.     throws BookServiceException {
44.     bookRepository.deleteBookCopy(bookId, copyId);
45. }
46.
47. @Override
48. public BookCopy getBookCopy(String bookCopyID)
49.     throws BookServiceException {
50.     throw new UnsupportedOperationException
51.         ("Unimplemented method 'getBookCopy'");
52. }

```

To recap, so far, we have implemented our domain classes, defined service and repository interfaces, and implemented them. We have not used any Spring or Spring MVC features in this application yet. Now, let us implement our controller class, **BookController**, in several steps. First, create a package called the controller and create the **BookController** class with the following code.

As the first step, create a package called the controller and create the **BookController** class with the following code:

```

1. package com.bpb.hssb.ch4.bookclub.controller;
2.
3. import org.springframework.stereotype.Controller;
4. import org.springframework.web.bind.annotation.RequestMapping;
5.
6. @Controller
7. @RequestMapping("/books")
8. public class BookController {
9.
10.    private BookService bookService;
11.
12.    public BookController(BookService bookService) {
13.        this.bookService = bookService;
14.    }

```

In the above code, we have annotated the **BookController** classes with **@Controller**, indicating that these classes are Spring MVC controllers. Additionally, we have used the **@RequestMapping** annotation, which

specifies a common path prefix or a common base URI for all the request mappings defined in the controller classes.

The constructor of the **BookController** class takes **BookService** as a parameter. If you revisit the **BookService** implementation classes, you will notice that we defined a constructor that takes **BookRepository** as a parameter. As you might have guessed, we are using Spring constructor injection in this application, which is why we have defined the dependencies of each class as constructor parameters. To clarify further, let us create a Spring Java configuration for our application before moving to the next step. Create a class called **BookclubConfiguration** and add the following code:

```
1. package com.bpb.hssb.ch4.bookclub;
2.
3. import org.springframework.context.annotation.Bean;
4. import org.springframework.context.annotation.Configuration;
5.
6. import com.bpb.hssb.ch4.bookclub.repository.BookRepository;
7. import com.bpb.hssb.ch4.bookclub.repository.InMemoryBookRepository;
8. import com.bpb.hssb.ch4.bookclub.service.BookService;
9. import com.bpb.hssb.ch4.bookclub.service.BookServiceImpl;
10.
11. @Configuration
12. public class BookclubConfiguration {
13.
14.     @Bean
15.     public BookRepository bookRepository() {
16.         return new InMemoryBookRepository();
17.     }
18.
19.     @Bean
20.     public BookService bookService(BookRepository bookRepository) {
21.         return new BookServiceImpl(bookRepository);
22.     }
23. }
```

Now, we can proceed by adding two methods to our controller classes to list all the books and to list the details of each book. Modify the class with the following code:

```
1. package com.bpb.hssb.ch4.bookclub.controller;
2.
```

```

3. import org.springframework.stereotype.Controller;
4. import org.springframework.ui.Model;
5.
6. import com.bpb.hssb.ch4.bookclub.service.BookService;
7. import com.bpb.hssb.ch4.bookclub.service.BookServiceException;
8.
9. import org.springframework.web.bind.annotation.GetMapping;
10. import org.springframework.web.bind.annotation.PathVariable;
11. import org.springframework.web.bind.annotation.RequestMapping;
12.
13. @Controller
14. @RequestMapping("/books")
15. public class BookController {
16.
17.     private BookService bookService;
18.
19.     public BookController(BookService bookService) {
20.         this.bookService = bookService;
21.     }
22.
23.     @GetMapping
24.     public String listAllBooks(Model model)
25.         throws BookServiceException {
26.         model.addAttribute("books", bookService.getAllBooks());
27.         return "books";
28.     }
29.     @GetMapping("/{bookId}")
30.     public String listBook(@PathVariable("bookId") String bookId,
31.         Model model) throws BookServiceException {
32.         model.addAttribute("book", bookService.getBook(bookId));
33.         return "book";
34.     }
35. }

```

On *line 25*, we annotated the **listAllBooks** method with the **@GetMapping** annotation, meaning this method will process the GET request with the **/books** path and return a view called **books**.

On *line 31*, we annotated the **listBook** method again with **@GetMapping** with a path parameter called **bookId**. This path parameter value can be

accessed using the **bookId** parameter, achieved by annotating the **bookId** parameter with the **@PathVariable** annotation. After processing the requests, the **listBook** method returns a view called **book**.

**Tip:** Bootstrap is a widely used open-source front-end framework that simplifies web development by offering pre-designed HTML, CSS, and JavaScript components for creating responsive and mobile-first designs. It allows developers to build visually appealing and consistent user interfaces quickly.

The final step is to create two views for books as defined in the controller classes. Create a **books.html** file in the **src/main/resources/templates/** directory with the following code:

```
1. <!DOCTYPE HTML>
2. <html xmlns:th="http://www.thymeleaf.org">
3.
4. <head>
5.   <title>BookClub</title>
6.   <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
7.   <meta name="viewport" content="width=device-width, initial-scale=1">
8.   <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css" rel="stylesheet">
9. </head>
10.
11. <body class="bg-light">
12.   <div class="container mt-5">
13.     <div class="row justify-content-center">
14.       <div class="col-md-8">
15.         <div class="card shadow">
16.           <div class="card-header bg-success text-white">
17.             <h2 class="mb-0 bg-success">BookClub</h2>
18.           </div>
19.           <div class="card-body">
20.             <table class="table table-striped table-hover">
21.               <thead class="table-dark">
22.                 <tr>
23.                   <th>Book ID</th>
24.                   <th>Title</th>
25.                   <th>Author</th>
26.                   <th>Publication</th>
27.                   <th>Copies</th>
28.             </tr>
```

```

29.          </thead>
30.          <tbody>
31.          <tr th:each="book : ${books}">
32.              <td>
33.                  <a th:href="@{/books/{id}
34. (id=${book.bookId})}" th:text="${book.bookId}"></a>
35.              </td>
36.              <td th:text="${book.title}"></td>
37.              <td th:text="${book.author}"></td>
38.              <td th:text="${book.publicationYear}"></td>
39.              <td th:text="${book.copies.size()}"></td>
40.          </tr>
41.      </tbody>
42.  </table>
43. </div>
44. </div>
45. </div>
46. </div>
47. </body>
48.
49. </html>

```

In this code, we used the Thymeleaf **th:each** attribute to iterate over the collection of books returned by the controller and render their details. For **bookId**, we used the Thymeleaf **th:href** attribute to construct a link to navigate to the individual book listing. Now, create the book.html view with the following code:

```

1. <!DOCTYPE HTML>
2. <html xmlns:th="http://www.thymeleaf.org">
3.
4. <head>
5. .....
6.
7. </head>
8.
9. <body class="bg-light">
10. <div class="container mt-5">
11.     <div class="row justify-content-center">
12.         <div class="col-md-8">
13.             <div class="card shadow">

```

```

14.
15.      <div class="card-header bg-success text-white d-flex justify-content-between align-items-center">
16.          <h2 class="mb-0 bg-success" th:text="${book.title}">Book Title</h2>
17.          <a href="/books" class="text-white">Home</a>
18.      </div>
19.      <div class="card-body">
20.          <dl class="row">
21.              <dt class="col-sm-3">Book ID:</dt>
22.              <dd class="col-sm-9" th:text="${book.bookId}">Book ID</dd>
23.
24.              <dt class="col-sm-3">ISBN:</dt>
25.              <dd class="col-sm-9" th:text="${book.isbn}">ISBN</dd>
26.
27.              <dt class="col-sm-3">Author:</dt>
28.              <dd class="col-sm-9" th:text="${book.author}">Author Name</dd>
29.
30.              <dt class="col-sm-3">Publication Year:</dt>
31.              <dd class="col-sm-9" th:text="${book.publicationYear}">Year</dd>
32.
33.              <dt class="col-sm-3">Number of Copies:</dt>
34.              <dd class="col-sm-9" th:text="${book.copies.size()}">0</dd>
35.          </dl>
36.
37.
38.      <h3 class="mt-4 mb-3">Book Copies</h3>
39.      <table class="table table-striped table-hover" th:if="${!book.copies.empty}">
40.          <thead class="table-dark">
41.              <tr>
42.                  <th>Copy ID</th>
43.                  <th>Status</th>
44.                  <th> </th>
45.              </tr>
46.          </thead>
47.          <tbody>
48.              <tr th:each="copy : ${book.copies}">
49.                  <td th:text="${copy.copyId}">Copy ID</td>
50.                  <td>
51.                      <span th:if="${copy.available}" class="badge bg-success">Available</span>
52.                      <span th:unless="${copy.available}" class="badge bg-

```

```

    danger">Not Available</span>
53.      </td>
54.      </tr>
55.      </tbody>
56.      </table>
57.      <p th:if="${book.copies.empty}" class="text-
    muted">No copies available for this book.</p>
58.      </div>
59.      </div>
60.      </div>
61.      </div>
62.      </div>
63. </body>
64.
65. </html>

```

Here, we also used the **th:each** attribute to iterate over the collection of book copies and the **th:if** attribute to conditionally render a view element. The particular conditional code segment is as follows:

```

1. <td>
2.   <span th:if="${copy.available}" class="badge bg-success">Available</span>
3.
4.   <span th:unless="${copy.available}" class="badge bg-danger">Not Available</span>
5. </td>

```

Additionally, we used Bootstrap as the CSS library to render the user interfaces nicely without developing our own CSS files.

Now, if you run the application again, you can access the books page using the URL **http://localhost:8080/books** and you should be able to see the home page as follows:

Book ID	Title	Author	Publication	Copies
BBB1	Mastering Java Persistence API	Nisha Parameswaran	2022	3
BBB6	Selenium with Java	Pallavi Sharma	2022	0
BBB5	Fundamentals of Android App Developmen	Sujit Kumar	2021	0
BBB4	JavaScript for Gurus	Ockert Preez	2020	0
BBB3	Software Design Patterns for Java Developers	Lalit Mehra	2021	0
BBB2	JAVA Programming Simplified	Muneer Ahmad	2020	0

*Figure 4.6: Book listing page*

Now, by clicking any of the **bookId**, you can view the details of the specific book as follows:

Book ID:	BBB1
ISBN:	9789355511263
Author:	Nisha Parameswaran
Publication Year:	2022
Number of Copies:	3

 Below this is a 'Book Copies' section with a table:
 

Copy ID	Status
BBB1-1	Not Available
BBB1-2	Available
BBB1-3	Available

*Figure 4.7: Books detail page*

Next, let us implement the add book functionality to the application. Add the following two methods to the **BookController** classes:

1. `@GetMapping("/add")`
2. `public String showAddBookForm(Model model)`  
`throws BookServiceException {`
3. `model.addAttribute("book", new Book());`
4. `return "add";`
5. `}`
6.

```

7. @PostMapping("/add")
8. public String addBook(Book book, RedirectAttributes
redirectAttributes) throws BookServiceException {
9.     bookService.createBook(book);
10.    redirectAttributes.addFlashAttribute("message",
11.        "The book " + book.getTitle() +
" by " + book.getAuthor() + " has been added");
12.    return "redirect:/books";
13. }

```

On line 1, we annotated the **showAddBookForm** method with **@GetMapping**, meaning that when a request comes through to the **/books/add** path using HTTP GET, the controller will return a view called **add** containing an HTML form. When this form is submitted using HTTP POST, the request is processed by the **addBook** method. The **addBook** method adds the book to the book repository through the service class and returns a redirect to the books listing page. It is also important to note that we are using **RedirectAttributes** with the **addFlashAttribute** method to pass a success message to the book page. The information passed through **addFlashAttribute** is retained during the redirects.

Next, create an **add.html** page and add the following code:

```

1. <!DOCTYPE HTML>
2. <html xmlns:th="http://www.thymeleaf.org">
3.
4. <head>
5.   <title>BookClub</title>
6.   <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
7.   <meta name="viewport" content="width=device-width, initial-scale=1">
8.   <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css" rel="stylesheet">
9. </head>
10.
11. <body class="bg-light">
12.   <div class="container mt-5">
13.     <div class="row justify-content-center">
14.       <div class="col-md-8">
15.         <div class="card shadow">
16.
17.           <div class="card-header bg-success text-white d-flex justify-content-between align-items-center">

```

```

18.      <h2 class="mb-0 bg-success" th:text="${book.title}">Book Title</h2>
19.      <a href="/books" class="text-white">Home</a>
20.    </div>
21.
22.    <div class="container mt-5">
23.      <h2>Add New Book</h2>
24.      <form th:action="@{/books/add}" th:object="${book}" method="post">
25.        <div class="mb-3">
26.          <label for="isbn" class="form-label">ISBN</label>
27.          <input type="text" class="form-control" id="isbn" th:field="*{isbn}" required>
28.        </div>
29.        <div class="mb-3">
30.          <label for="title" class="form-label">Title</label>
31.          <input type="text" class="form-control" id="title" th:field="*{title}" required>
32.        </div>
33.        <div class="mb-3">
34.          <label for="author" class="form-label">Author</label>
35.          <input type="text" class="form-control" id="author" th:field="*{author}" required>
36.        </div>
37.        <div class="mb-3">
38.          <label for="publicationYear" class="form-label">Publication Year</label>
39.          <input type="number" class="form-control" id="publicationYear" th:field="*{publicationYear}" required>
40.        </div>
41.      </div>
42.      <button type="submit" class="btn btn-primary">Add Book</button>
43.    </form>
44.  </div>
45.
46.  </div>
47. </div>
48. </div>
49. </div>
50. </body>
51.
52. </html>

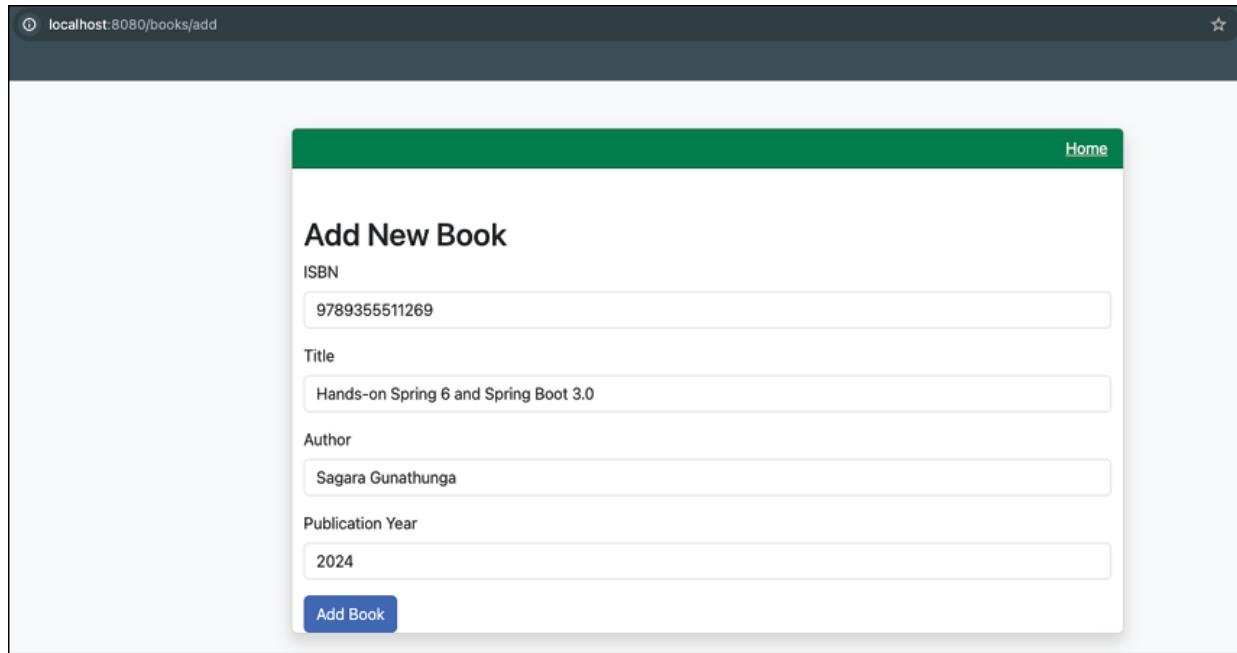
```

In the above code, we added the **th:action** and **th:object** attributes to the

HTML form element to set the form submission correctly. To make it clear, the specific code segments are as follows:

```
1. <form th:action="@{/books/add}" th:object="${book}" method="post">
2.
3.   <div class="mb-3">
4.     <label for="isbn" class="form-label">ISBN</label>
5.     <input type="text" class="form-control" id="isbn" th:field="*{isbn}" required>
6.   </div>
7.   <div class="mb-3">
8.     <label for="title" class="form-label">Title</label>
9.     <input type="text" class="form-control"
10.    id="title" th:field="*{title}" required>
11.   </div>
12.   <div class="mb-3">
13.     <label for="author" class="form-label">Author</label>
14.     <input type="text" class="form-control" id="author"
15.       th:field="*{author}" required>
16.   </div>
17.   <div class="mb-3">
18.     <label for="publicationYear"
19.       class="form-label">Publication Year</label>
20.     <input type="number" class="form-control"
21.       id="publicationYear" th:field="*{publicationYear}" required>
22.   </div>
23.   <button type="submit" class="btn btn-primary">Add Book</button>
24. </form>
```

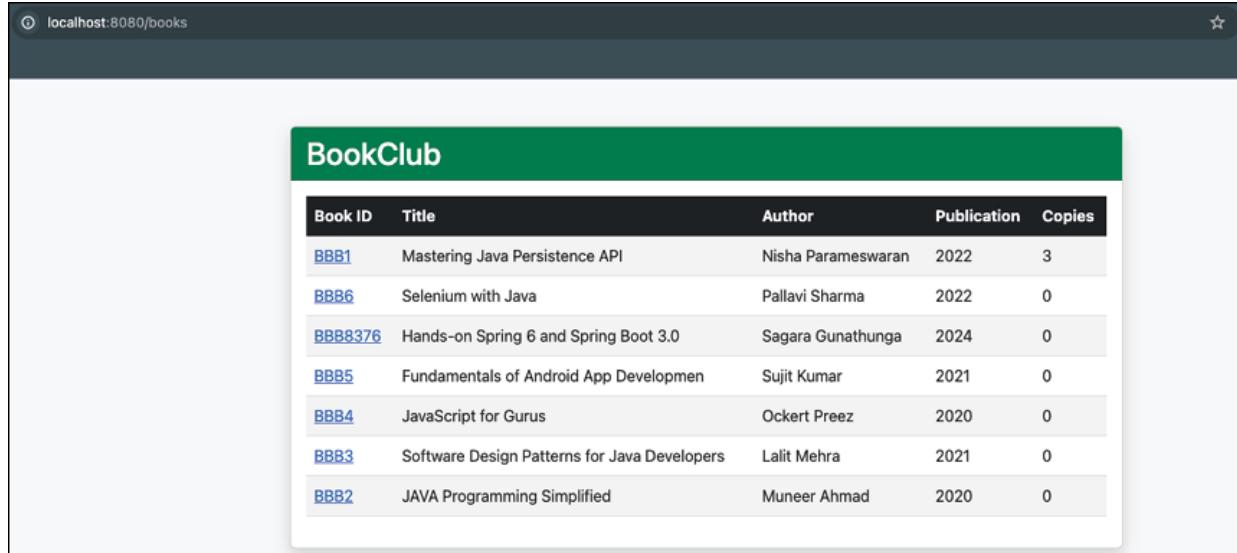
Now, if you visit the URL <http://localhost:8080/books/add>, you should see the following HTML form:



The screenshot shows a web browser window with the URL `localhost:8080/books/add` in the address bar. The page has a green header bar with a 'Home' link. The main content area is titled 'Add New Book'. It contains four input fields: 'ISBN' (value: 9789355511269), 'Title' (value: Hands-on Spring 6 and Spring Boot 3.0), 'Author' (value: Sagara Gunathunga), and 'Publication Year' (value: 2024). Below these fields is a blue 'Add Book' button.

**Figure 4.8:** Add new book form

Once you submit the form, you should see the book listing page with the newly added book, as shown in the following figure:

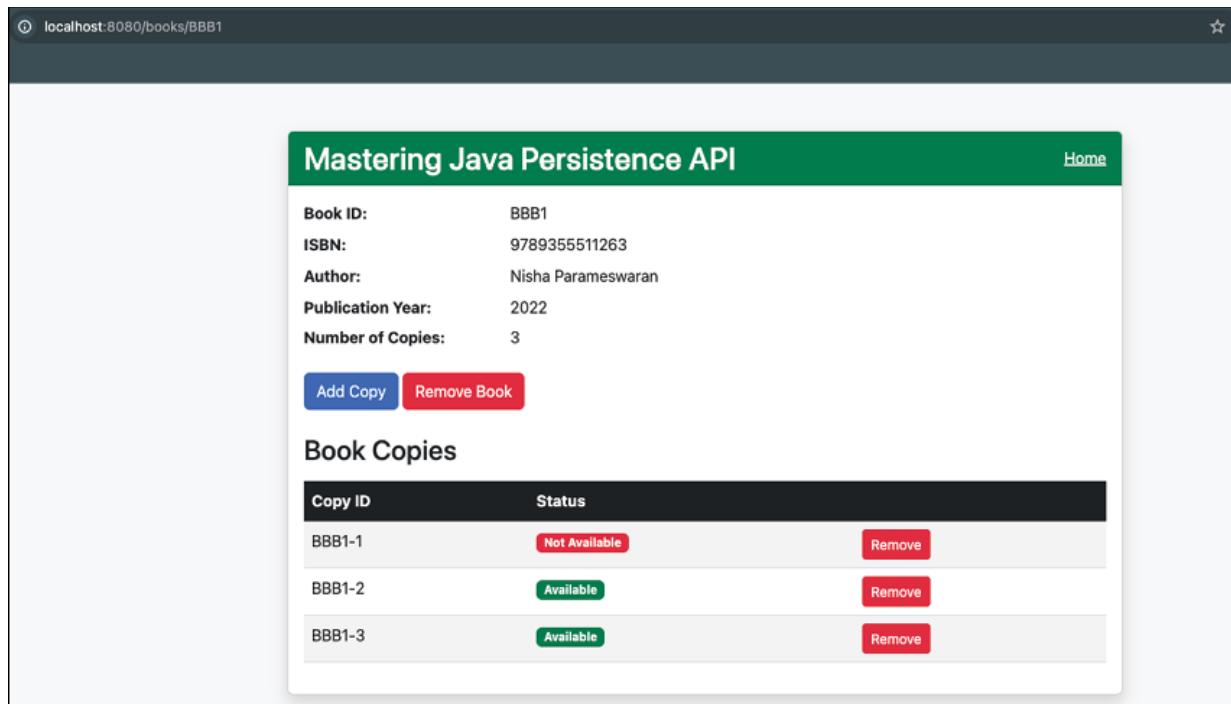


The screenshot shows a web browser window with the URL `localhost:8080/books` in the address bar. The page has a green header bar with the title 'BookClub'. Below the header is a table with the following data:

Book ID	Title	Author	Publication	Copies
BBB1	Mastering Java Persistence API	Nisha Parameswaran	2022	3
BBB6	Selenium with Java	Pallavi Sharma	2022	0
BBB8376	Hands-on Spring 6 and Spring Boot 3.0	Sagara Gunathunga	2024	0
BBB5	Fundamentals of Android App Developmen	Sujit Kumar	2021	0
BBB4	JavaScript for Gurus	Ockert Preez	2020	0
BBB3	Software Design Patterns for Java Developers	Lalit Mehra	2021	0
BBB2	JAVA Programming Simplified	Muneer Ahmad	2020	0

**Figure 4.9:** Book listing page with the newly added book

At this stage, we have covered the basics of rendering a view with data, form submission, and redirections. The full application code available in the companion GitHub repository contains additional methods such as remove book, add book copy, and remove book copy to make the application more realistic. Here is a screenshot of the list book page with all the modifications:



*Figure 4.10: Books detail page with add and remove buttons*

## Data validation

Spring MVC offers robust validation capabilities for form handling and data processing, supporting both server-side and client-side validation. It integrates with the Java Bean Validation API (JSR-303) and its reference implementation, Hibernate Validator. Spring MVC automatically binds form data to model objects and applies the specified validations. Validation errors are captured in a **BindingResult** object, which can be easily accessed in controller methods to handle error scenarios. In addition to built-in validation annotations, you can develop your own custom validators as well. In this section, we will apply validation to the add book form that we developed in the previous section.

First, you need to add **spring-boot-starter-validation** as a dependency to your project by adding the following dependency to the POM file of your project:

1. <dependency>
2. <groupId>org.springframework.boot</groupId>
3. <artifactId>spring-boot-starter-validation</artifactId>
4. </dependency>

This dependency adds the Bean Validation API and its reference implementation, Hibernate Validator, to the classpath, and automatically configures and enables validation support in the application.

Now, let us go ahead and declare validation rules for our domain classes using the annotations provided by the Bean Validation API. Here is the **Book** class after applying the validation annotations:

```
1. package com.bpb.hssb.ch4.bookclub.domain;
2.
3. import java.util.List;
4.
5. import jakarta.validation.constraints.Max;
6. import jakarta.validation.constraints.Min;
7. import jakarta.validation.constraints.NotBlank;
8. import jakarta.validation.constraints.NotNull;
9. import jakarta.validation.constraints.Pattern;
10. import jakarta.validation.constraints.Size;
11. import lombok.AllArgsConstructor;
12. import lombok.Builder;
13. import lombok.Getter;
14. import lombok.NoArgsConstructor;
15. import lombok.Setter;
16.
17. @Getter
18. @Setter
19. @NoArgsConstructor
20. @NoArgsConstructor
21. @Builder
22. public class Book {
23.
24.     private String bookId;
25.
26.     @NotBlank(message = "ISBN is required")
27.     @Pattern(regexp = "^\d{13}$", message =
28.             "ISBN must be exactly 13 digits")
29.     private String isbn;
30.
31.     @NotBlank(message = "Title is required")
32.     @Size(min = 10, max = 100, message =
33.             "Title must be between 10 and 100 characters")
34.     private String title;
```

```

33.
34. @NotNull(message = "Author is required")
35. @Size(min = 10, max = 100, message =
   "Author name must be between 10 and 100 characters")
36. private String author;
37.
38. @NotNull(message = "Publication year is required")
39. @Min(value = 1000, message = "Publication year must be after 1000")
40. @Max(value = 9999, message = "Publication year must be before 9999")
41. private int publicationYear;
42.
43. private List<BookCopy> copies;
44.
45. }

```

We have used a set of validation annotations such as **@NotNull**, **@Size**, **@NotNull**, **@Min**, **@Max**, and **@Pattern** to declare our validation rules. After declaring rules through these annotations, we do not need to worry about performing the validation.

Next, we need to modify the **addBook** method in the **BookController** class to return the same form in case of validation errors. The following code segment provides the updated **addBook** method:

```

1. @PostMapping("/add")
2. public String addBook(@ModelAttribute("book") @Valid Book book, BindingResult binding
   Result,
3. RedirectAttributes redirectAttributes)
   throws BookServiceException {
4.
5. if (bindingResult.hasErrors()) {
6.   return "add";
7. }
8. bookService.createBook(book);
9. redirectAttributes.addFlashAttribute("message",
10.   "The book " + book.getTitle() + " by " + book.getAuthor() + " has been added");
11. return "redirect:/books";
12. }

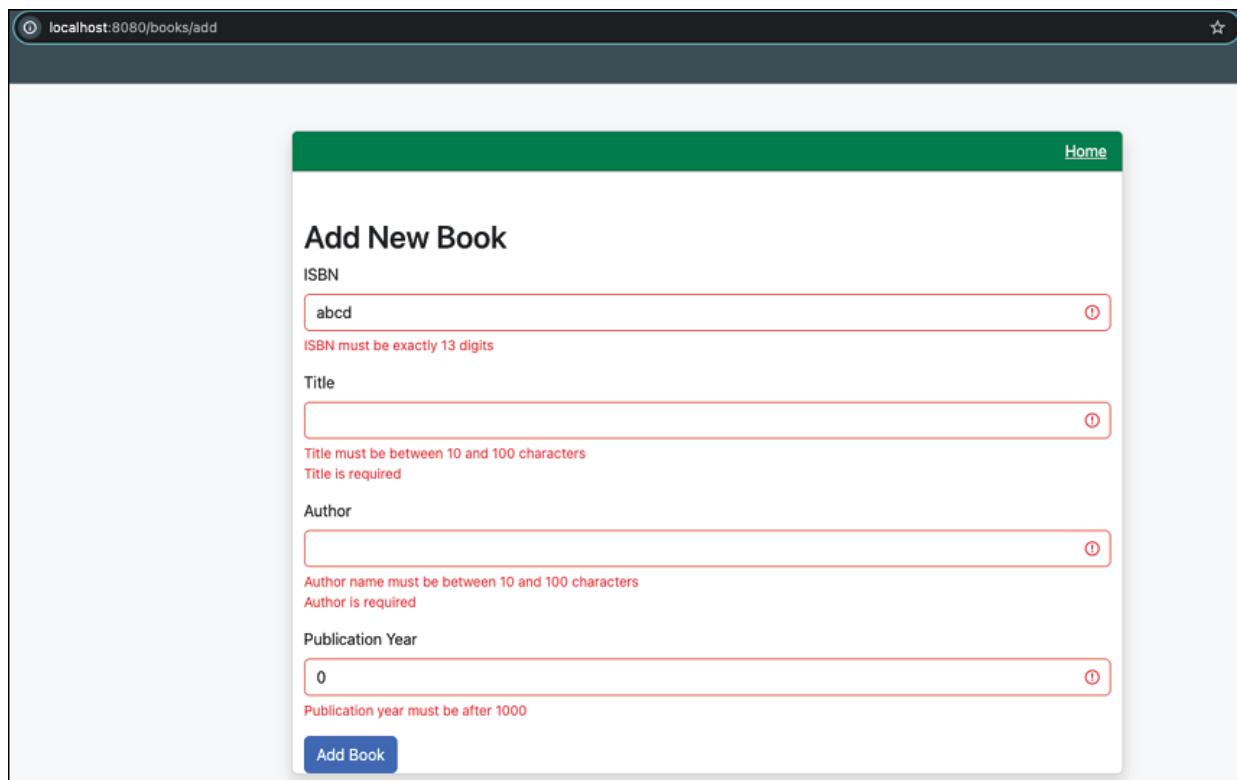
```

In the above code, we inject **BindingResult** as a parameter to the method so that we can evaluate whether there are any validation errors and handle them accordingly. In our application, we simply return the same form again, asking the user to correct the validation mistakes.

As the final step, we need to modify the add.html form to show the errors if there are any validation errors. The following code segment shows how you could add error messages to the form:

```
1. <form th:action="@{/books/add}" th:object="${book}" method="post">
2.   <div class="mb-3">
3.     <label for="isbn" class="form-label">ISBN</label>
4.     <input type="text" class="form-control" id="isbn" th:field="*{isbn}" th:errorclass="is-
   invalid">
5.     <div class="invalid-feedback" th:if="#{#fields.hasErrors('isbn')}" th:errors="*{isbn}">
   </div>
6.   </div>
7.
8.   <div class="mb-3">
9.     <label for="title" class="form-label">Title</label>
10.    <input type="text" class="form-control" id="title" th:field="*{title}" th:errorclass="is-
    invalid">
11.    <div class="invalid-feedback" th:if="#{#fields.hasErrors('title')}" th:errors="*{title}">
   </div>
12.  </div>
13.
14.  <div class="mb-3">
15.    <label for="author" class="form-label">Author</label>
16.    <input type="text" class="form-control" id="author" th:field="*{author}" th:errorclass="is-invalid">
17.    <div class="invalid-feedback" th:if="#{#fields.hasErrors('author')}" th:errors="*{author}"></div>
18.  </div>
19.
20.  <div class="mb-3">
21.    <label for="publicationYear"
   class="form-label">Publication Year</label>
22.    <input type="number" class="form-control" id="publicationYear" th:field="*{publicationYear}">
   <div class="invalid-feedback" th:if="#{#fields.hasErrors('publicationYear')}" th:errors="*{publicationYear}">
23.    </div>
24.    <div class="invalid-feedback" th:if="#{#fields.hasErrors('publicationYear')}" th:errors="*{publicationYear}">
25.    </div>
26.  </div>
27.
28.  <button type="submit" class="btn btn-primary">Add Book</button>
29. </form>
```

The following figure shows how validation errors are displayed in the add book form once you complete the above steps:



The screenshot shows a web browser window with the URL `localhost:8080/books/add` in the address bar. The page title is "Add New Book". The form has four fields: ISBN, Title, Author, and Publication Year. Each field has an error message displayed below it. The "ISBN" field contains "abcd" and the error message "ISBN must be exactly 13 digits". The "Title" field is empty and has two error messages: "Title must be between 10 and 100 characters" and "Title is required". The "Author" field is empty and has two error messages: "Author name must be between 10 and 100 characters" and "Author is required". The "Publication Year" field contains "0" and the error message "Publication year must be after 1000". A "Home" link is visible in the top right corner of the page.

**Figure 4.11:** Add new book form with validation errors

By following these steps, you have implemented validation for your form, ensuring that any input errors are handled gracefully and communicated back to the user.

## Error handling

Spring MVC provides a flexible and comprehensive set of error-handling features. These include the `@ExceptionHandler` annotation for simple exception handling, as well as `ExceptionResolver` implementations such as `HandlerExceptionResolver` and `SimpleMappingExceptionResolver` for more complex and custom error handling use cases. For our simple application, we will use the `@ExceptionHandler` annotation to handle exceptions.

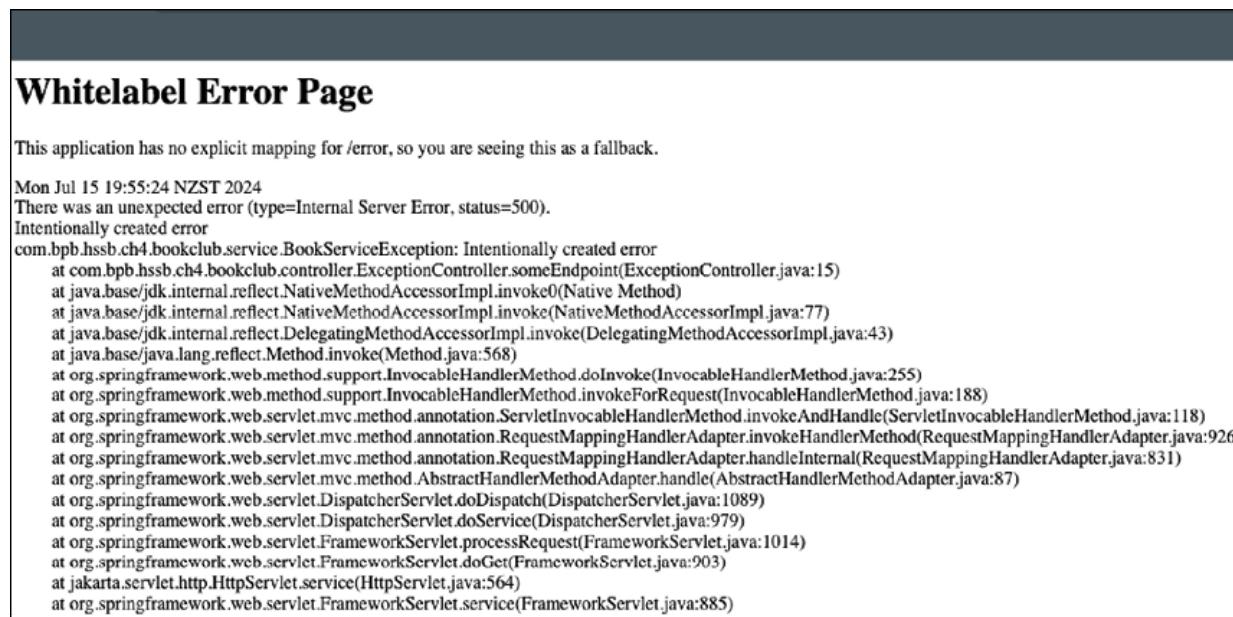
First, let us intentionally create a controller that throws an error. Create a class called `ExceptionController.java` with the following code:

```

1. package com.bpb.hssb.ch4.bookclub.controller;
2.
3. import org.springframework.stereotype.Controller;
4. import org.springframework.web.bind.annotation.RequestMapping;
5.
6. import com.bpb.hssb.ch4.bookclub.service.BookServiceException;
7.
8. @Controller
9. public class ExceptionController {
10.
11.     @RequestMapping("/geterror")
12.     public String someEndpoint() throws BookServiceException {
13.         throw new BookServiceException
14.             ("Intentionally created error");
15.     }

```

Now, if you call the **http://localhost:8080/geterror** URL, you will be able to see an error page similar to the following figure:



*Figure 4.12: Default error page*

Displaying raw error pages can make your application look broken and reveal critical internal details that should not be exposed to the outside world. As a programmer, you must handle exceptions carefully and render appropriate error pages to enhance the user experience and ensure sensitive

application details are not leaked.

Since we have intentionally produced an error, let us handle the exception and render a safe and meaningful error message. Add the following code to the **ExceptionController.java** class:

```
1. @ExceptionHandler(BookServiceException.class)
2. public String handleBookServiceException(BookServiceException ex, Model model) {
3.     model.addAttribute("message", ex.getMessage());
4.     return "error/error";
5. }
```

In the above code, we used **@ExceptionHandler** to indicate that the **handleBookServiceException** method is an error handling method capable of handling **BookServiceException** exceptions. Within the method **body**, we set the error message as a **model** attribute and return a view called **error**.

Next, create the error view. Create a directory called **error** under the **src/main/resources/templates** location and add **error.html** with the following code:

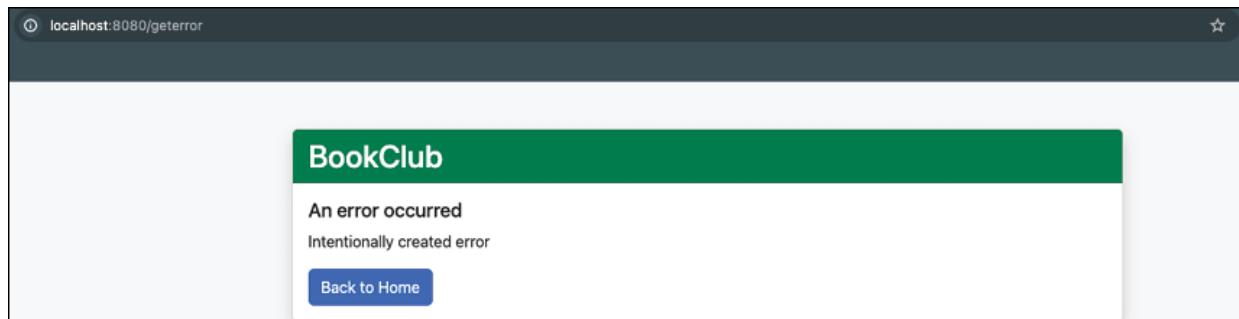
```
1. <!DOCTYPE HTML>
2. <html xmlns:th="http://www.thymeleaf.org">
3.
4. <head>
5.   <title>BookClub</title>
6.   <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
7.   <meta name="viewport" content="width=device-width, initial-scale=1">
8.   <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css" rel="stylesheet">
9. </head>
10.
11. <body class="bg-light">
12.   <div class="container mt-5">
13.     <div class="row justify-content-center">
14.       <div class="col-md-8">
15.         <div class="card shadow">
16.           <div class="card-header bg-success text-white">
17.             <h2 class="mb-0 bg-success">BookClub</h2>
18.           </div>
19.
20.         <div class="card-body">
21.           <h5 class="card-title">An error occurred</h5>
```

```

22.         <P th:text="${message}"></p>
23.         <a href="/" class="btn btn-primary">Back to Home</a>
24.     </div>
25.     </div>
26.     </div>
27.     </div>
28.     </div>
29. </body>
30.
31. </html>

```

This is a simple view designed to show the error message and provide a link for users to navigate back to the home page. At this point, if you revisit the **http://localhost:8080/geterror** URL, you should see the following screen:



*Figure 4.13: Custom error page*

Although we managed to handle the exception successfully, this method has a major limitation: it only handles exceptions thrown by request handler methods of the same controller. This can result in duplicating the same exception handling logic across multiple controllers, reducing the readability and maintainability of the code.

To address this, we can use the **@ExceptionHandler** annotation along with **@ControllerAdvice** to handle exceptions globally. Create a class called **GlobalExceptionHandler.java** with the following code:

```

1. package com.bpb.hssb.ch4.bookclub.controller;
2.
3. import org.springframework.ui.Model;
4. import org.springframework.web.bind.annotation.ControllerAdvice;
5. import org.springframework.web.bind.annotation.ExceptionHandler;
6. import org.springframework.web.servlet.resource.NoResourceNotFoundException;
7.

```

```

8. import com.bpb.hssb.ch4.bookclub.service.BookServiceException;
9.
10. @ControllerAdvice
11. public class GlobalExceptionHandler {
12.
13.     @ExceptionHandler(BookServiceException.class)
14.     public String handleBookServiceException(BookServiceException ex, Model model) {
15.         model.addAttribute("message", ex.getMessage());
16.         return "error/error";
17.     }
18.
19.     @ExceptionHandler(NoResourceFoundException.class)
20.     public String handleNoResourceFoundException(NoResourceFoundException ex, Model mo
del) {
21.         return "error/default";
22.     }
23.
24.     @ExceptionHandler(RuntimeException.class)
25.     public String handleRuntimeException(RuntimeException ex, Model model) {
26.         return "error/default";
27.     }
28. }
```

In the above code, we moved **handleBookServiceException**, which handles **BookServiceException**, to the **GlobalExceptionHandler** class. Additionally, we defined two other error-handling methods to handle **NoResourceFoundException** and **RuntimeException**.

We also defined the default error page with the following content:

```

1. <!DOCTYPE HTML>
2. <html xmlns:th="http://www.thymeleaf.org">
3.
4. <head>
5.     <title>BookClub</title>
6.     <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
7.     <meta name="viewport" content="width=device-width, initial-scale=1">
8.     <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css" rel="styl
esheet">
9. </head>
10.
11. <body class="bg-light">
12.     <div class="container mt-5">
```

```

13.   <div class="row justify-content-center">
14.     <div class="col-md-8">
15.       <div class="card shadow">
16.         <div class="card-header bg-success text-white">
17.           <h2 class="mb-0 bg-success">BookClub</h2>
18.         </div>
19.
20.       <div class="card-body">
21.         <h5 class="card-title">An error occurred</h5>
22.         <p class="card-text">Something went wrong!</p>
23.         <a href="/" class="btn btn-primary">Back to Home</a>
24.       </div>
25.     </div>
26.   </div>
27. </div>
28. </div>
29. </body>
30.
31. </html>

```

## Securing Spring MVC applications

If you have not tried the complete application available in the companion GitHub repository, now is a great time to play around with it. If you have, you might have noticed that we have not implemented any security yet. Specifically, anyone can add or remove books without needing to log in. In a real-world scenario, users must log in before making any changes. Based on what we learned in the previous chapter about Spring Security, let us add a login feature to our application.

As we have done several times, the first step is to add the Spring Security starter as a dependency to your project:

```

1. <dependency>
2.   <groupId>org.springframework.boot</groupId>
3.   <artifactId>spring-boot-starter-security</artifactId>
4. </dependency>

```

This dependency adds the necessary security components to your application. Next, we will create a simple HTML form for login:

```

1. <!DOCTYPE HTML>

```

```
2. <html xmlns:th="http://www.thymeleaf.org">
3.
4. <head>
5.   <title>Book Club</title>
6.   <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
7.   <meta name="viewport" content="width=device-width, initial-scale=1">
8.   <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css" rel="stylesheet">
9. </head>
10.
11. <body class="bg-light">
12.   <div class="container mt-5">
13.     <div class="row justify-content-center">
14.       <div class="col-md-8">
15.         <div class="card shadow">
16.
17.           <div class="card-header bg-success text-white">
18.             <h2 class="mb-0 bg-success">
    Book Club - Login</h2>
19.           </div>
20.           <div class="card-body">
21.             <form th:action="@{/login}" method="post">
22.               <div class="mb-3">
23.                 <label for="username"
    class="form-label">Username:</label>
24.                 <input type="text" id="username"
    name="username" class="form-control" required>
25.               </div>
26.               <div class="mb-3">
27.                 <label for="password"
    class="form-label">Password:</label>
28.                 <input type="password"
    id="password" name="password" class="form-control" required>
29.               </div>
30.               <div th:if="${param.error}" class="mb-3">
31.                 <p class="text-danger">
    Invalid username or password</p>
32.               </div>
33.               <div class="mb-3">
34.                 <input type="submit"
    value="Log in" class="btn btn-primary">
35.               </div>
```

```
36.          </form>
37.      </div>
38.
39.      </div>
40.      </div>
41.  </div>
42. </div>
43. </body>
44.
45. </html>
```

Given that we have designed the login page as a view, we need to register a controller to render the login view. Create a class called **LoginController.java** and add the following code:

```
1. package com.bpb.hssb.ch4.bookclub.controller;
2.
3. import org.springframework.stereotype.Controller;
4. import org.springframework.web.bind.annotation.GetMapping;
5.
6. @Controller
7. public class LoginController {
8.
9.     @GetMapping("/login")
10.    public String login() {
11.        return "login";
12.    }
13.
14. }
```

Now, we have a login page and a **LoginController**. Our next step is to configure Spring Security according to our requirements. Create a class called **SecurityConfiguration.java** with the following code:

```
1. package com.bpb.hssb.ch4.bookclub.config;
2.
3. import org.springframework.context.annotation.Bean;
4. import org.springframework.context.annotation.Configuration;
5. import org.springframework.security.
config.annotation.web.builders.HttpSecurity;
6. import org.springframework.security.web.SecurityFilterChain;
7. import org.springframework.security.web.authentication.logout.
SecurityContextLogoutHandler;
8.
```

```

9. @Configuration
10. public class SecurityConfiguration {
11.
12.     @Bean
13.     public SecurityFilterChain securityFilterChain
14.         (HttpSecurity http) throws Exception {
15.             http
16.                 .authorizeHttpRequests((requests) -> requests
17.                     .requestMatchers("/books/**").authenticated()
18.                     .anyRequest().permitAll())
19.                 .formLogin((form) -> form
20.                     .loginPage("/login")
21.                     .defaultSuccessUrl("/books", true)
22.                     .permitAll());
23.             http
24.                 .logout(logout -> logout
25.                     .logoutUrl("/logout")
26.                     .logoutSuccessUrl("/login")
27.                     .invalidateHttpSession(true)
28.                     .addLogoutHandler(new SecurityContextLogoutHandler()));
29.             http
30.                 .anonymous(anonymous -> anonymous
31.                     .disable());
32.
33.         return http.build();
34.     }
35. }

```

The explanation is as follows:

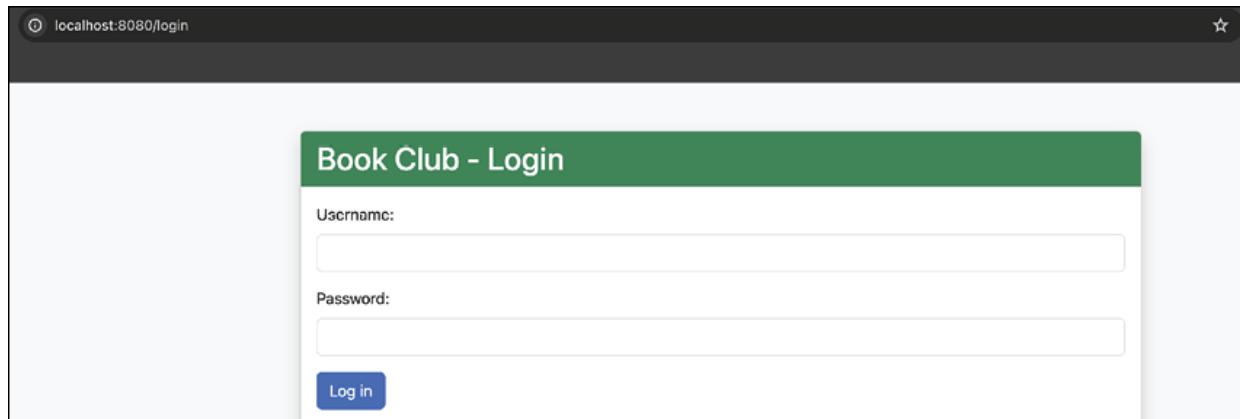
- *Line 16* specifies that any URL starting with `/books/` requires authentication.
- *Line 17* allows any other request (not matching `"/books/**"`) to be accessed without authentication.
- *Line 18* indicated the beginning of the form-based login configuration.
- *Line 19* specifies the custom login page URL as `/login`.
- *Line 20* sets the URL where users are redirected after successful login.
- *Line 20* allows anyone to access the login page without authentication.
- *Line 22* starts the configuration for the logout.

- *Line 24* Specifies the URL that triggers the logout.
- *Line 25* Sets the URL where users are redirected after successfully logging out.
- *Line 26* ensures that the user's HTTP session is invalidated when they log out.
- *Line 27* ensures that the user's authentication information is properly cleared.
- *Line 29* disables the anonymous authentication feature.

Before trying out our application, add the following test user credentials to the **application.properties** file:

1. `spring.security.user.name=sam`
2. `spring.security.user.password=abcd`

Now, if you visit **http://localhost:8080/books**, you should see the login screen as follows and be able to log in with the above credentials. Once logged in, you can see the book listing as before.



**Figure 4.14:** Custom login page

Once you logged in successfully, you would be able to see the book listing as you have seen earlier. The following figure shows what you would be able to see:

BookClub

Book ID	Title	Author	Publication	Copies
BBB1	Mastering Java Persistence API	Nisha Parameswaran	2022	3
BBB6	Selenium with Java	Pallavi Sharma	2022	0
BBB5	Fundamentals of Android App Developmen	Sujit Kumar	2021	0
BBB4	JavaScript for Gurus	Ockert Preez	2020	0
BBB3	Software Design Patterns for Java Developers	Lalit Mehra	2021	0
BBB2	JAVA Programming Simplified	Muneer Ahmad	2020	0

*Figure 4.15: Book listing page after user login*

Although you are logged in, your username is not displayed, and there are no visible logout links. Instead of setting the logged-in user in each controller method, let us write a **HandlerInterceptor** to set the logged-in username at a global level without cluttering the controllers. Create a class called **CommonAttributesInterceptor.java** and add the following code:

```

1. package com.bpb.hssb.ch4.bookclub.config;
2.
3. import org.springframework.security.core.Authentication;
4. import org.springframework.security.core.context.
   SecurityContextHolder;
5. import org.springframework.security.core.userdetails.UserDetails;
6. import org.springframework.web.servlet.HandlerInterceptor;
7. import org.springframework.web.servlet.ModelAndView;
8.
9. import jakarta.servlet.http.HttpServletRequest;
10. import jakarta.servlet.http.HttpServletResponse;
11.
12. public class CommonAttributesInterceptor
   implements HandlerInterceptor {
13.
14.     @Override
15.     public void postHandle(HttpServletRequest request,
   HttpServletResponse response, Object handler,
16.             ModelAndView modelAndView) throws Exception {
17.         if (modelAndView != null) {
18.             modelAndView.addObject("user", getCurrentUsername());
19.     }

```

```

20.  }
21.
22.  private Object getCurrentUsername() {
23.      Authentication authentication = SecurityContextHolder
24.          .getContext().getAuthentication();
25.      if (authentication != null &&
26.          authentication.isAuthenticated()) {
27.          UserDetails userDetails = (UserDetails)
28.              authentication.getPrincipal();
29.          return userDetails.getUsername();
30.      }
31.  }

```

The above **HandlerInterceptor** accesses logged-in user details from **SecurityContextHolder** and sets it as an attribute in the model.

Now that we have managed to set the logged-in username to the model, the next task is to modify the views to display the logged-in user. This can be done by adding the following code segment to each view template:

```

1. <div class="d-flex align-items-center">
2.   <span th:if="${user != null}" class="me-3">
3.     You logged in as : <span th:text="${user}"></span>
4.   </span>
5.   <a th:href="@{/logout}" class="text-white"
6.     onclick="event.preventDefault(); document.getElementById('logout-form').submit();">
7.     Logout
8.   </a>
9.   <form id="logout-form" th:action="@{/logout}" method="post" style="display: none;">
10.  </form>
11. </div>

```

We also added a hidden form to call the logout URL. Form submissions are made using HTTP POST, which is considered more secure than HTTP GET. If you log in to the application again, you should be able to see the following screen:

Book ID	Title	Author	Publication	Copies
BBB1	Mastering Java Persistence API	Nisha Parameswaran	2022	4
BBB6	Selenium with Java	Pallavi Sharma	2022	0
BBB5	Fundamentals of Android App Developmen	Sujit Kumar	2021	0
BBB4	JavaScript for Gurus	Ockert Preez	2020	0
BBB3	Software Design Patterns for Java Developers	Lalit Mehra	2021	0
BBB2	JAVA Programming Simplified	Muneer Ahmad	2020	0

**Figure 4.16:** Book listing page with the logged-in username and logout link

As the last exercise of this section, we will implement role-based access control for our application. Let us assume that users with the **ADMIN** role can perform the add and remove books to the system while users only with the **USER** role can list the book but they cannot add and remove books from the system.

As the first step, remove the following configuration from **application.properties** file:

1. `spring.security.user.name=sam`
2. `spring.security.user.password=abcd`

Instead of that, add the following two users with the roles by modifying the **SecurityConfiguration** class:

```

1.  @Bean
2.  public UserDetailsService userDetailsService() {
3.
4.      List<UserDetails> users = new ArrayList<>();
5.
6.      users.add(User.withDefaultPasswordEncoder()
7.                  .username("sam")
8.                  .password("abcd")
9.                  .roles("USER", "ADMIN")
10.                 .build());
11.
12.     users.add(User.withDefaultPasswordEncoder()
13.                  .username("alex")
14.                  .password("abcd"))

```

```

15.         .roles("USER")
16.         .build());
17.
18.     return new InMemoryUserDetailsManager(users);
19. }

```

The above bean definition adds two users called **sam** with both **ADMIN** and **USER** access and **alex** with **USER** level access.

Then, we need to modify the **CommonAttributesInterceptor** we developed previously to set whether the current user has admin-level access or not. Here is the updated code for **CommonAttributesInterceptor** class:

```

1. package com.bpb.hssb.ch4.bookclub.config;
2.
3. import org.springframework.security.core.Authentication;
4. import org.springframework.security.core.GrantedAuthority;
5. import org.springframework.security.core.context.SecurityContextHolder;
6. import org.springframework.security.core.userdetails.UserDetails;
7. import org.springframework.web.servlet.HandlerInterceptor;
8. import org.springframework.web.servlet.ModelAndView;
9.
10. import jakarta.servlet.http.HttpServletRequest;
11. import jakarta.servlet.http.HttpServletResponse;
12.
13. public class CommonAttributesInterceptor
14.     implements HandlerInterceptor {
15.
16.     @Override
17.     public void postHandle(HttpServletRequest request,
18.                           HttpServletResponse response, Object handler,
19.                           ModelAndView modelAndView) throws Exception {
20.
21.         if (modelAndView != null) {
22.             modelAndView.addObject("user", getCurrentUsername());
23.             modelAndView.addObject("admin", isAdmin());
24.         }
25.     }
26.
27.     private Object getCurrentUsername() {
28.         Authentication authentication =
29.             SecurityContextHolder.getContext().getAuthentication();
30.
31.         if (authentication != null &&
32.             authentication.isAuthenticated()) {

```

```

27.     UserDetails userDetails =
28.         (UserDetails) authentication.getPrincipal();
29.     }
30.     return null;
31. }
32.
33. private boolean isAdmin() {
34.     Authentication authentication =
35.         SecurityContextHolder.getContext().getAuthentication();
36.     if (authentication != null &&
37.         authentication.isAuthenticated()) {
38.         UserDetails userDetails =
39.             (UserDetails) authentication.getPrincipal();
40.         for(GrantedAuthority authority :
41.             userDetails.getAuthorities()){
42.             if (authority.getAuthority().equals("ROLE_ADMIN")) {
43.                 return true;
44.             }
45.         }
46.     }
47.     return false;
48. }
49.

```

In the above code, we retrieve roles associated with the current user using **SecurityContextHolder**.

In addition to that, we have to make a slight modification to views so that add book, remove book, add copy, and remove copy buttons are only visible if the current user has an **ADMIN** role. That can be done by adding **th:if="\${admin}"** attribute to the HTML element. Here is how you can add to a link:

```

1.  <a th:if="${admin}" th:href="@{/books/add}" class="btn bg-primary me-2">Add Book</a>

```

The following code segment shows how you can add the same condition to an HTML form:

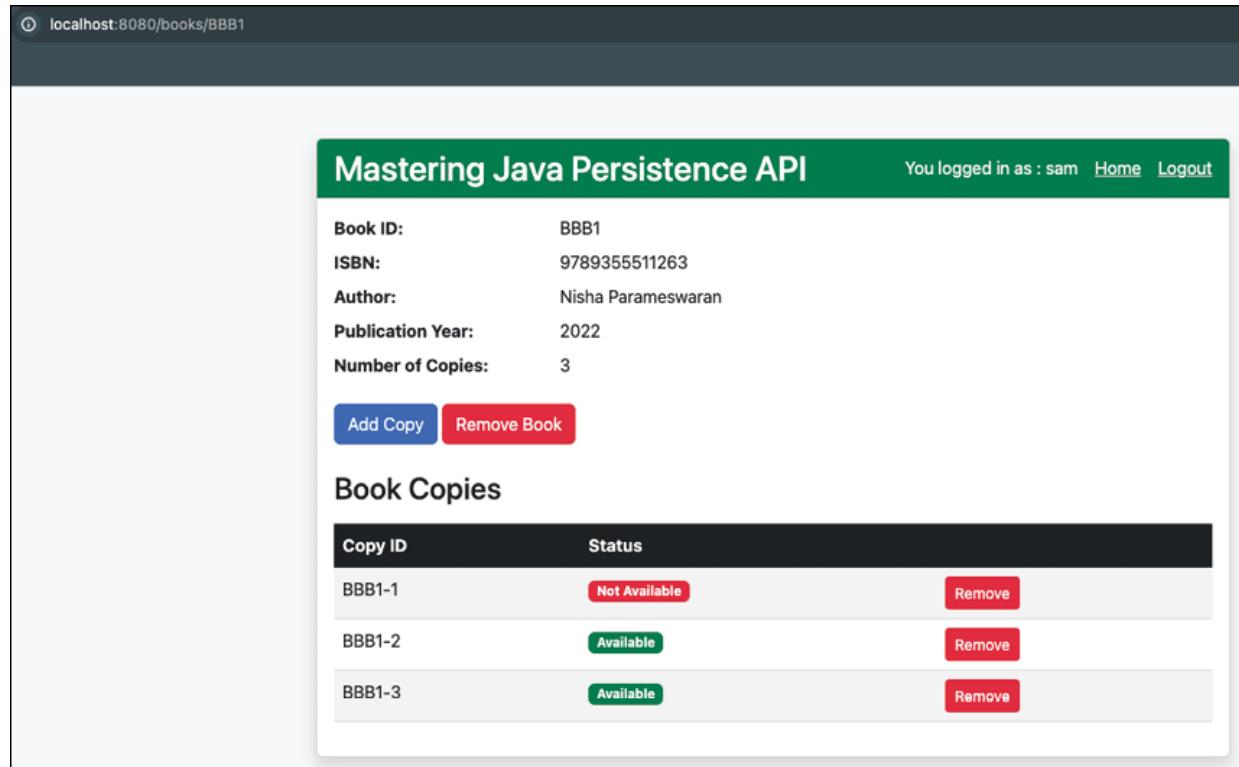
```

1.  <form th:if="${admin}" th:action="@{/books/remove/{id}}"
2.      (id=${book.bookId})" method="post" style="display: inline;">
3.      <button type="submit" class="btn btn-danger">Remove Book</button>
4.  </form>

```

Now, if you logged in as *Sam* you should be able to see the following book

detail page:



The screenshot shows a web application interface. At the top, a dark header bar displays the URL "localhost:8080/books/BBB1". Below this is a green navigation bar with the text "Mastering Java Persistence API" on the left, and "You logged in as : sam [Home](#) [Logout](#)" on the right. The main content area is white and contains the following information:

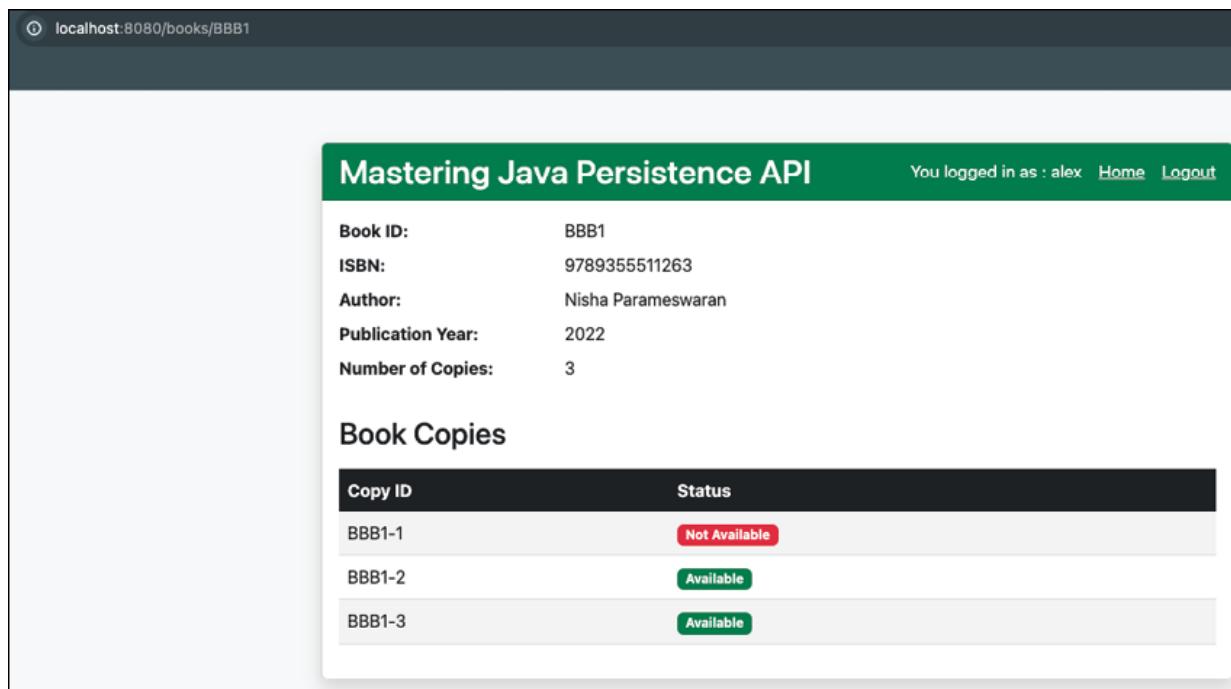
Book ID:	BBB1
ISBN:	9789355511263
Author:	Nisha Parameswaran
Publication Year:	2022
Number of Copies:	3

Below this, there are two buttons: "Add Copy" (blue) and "Remove Book" (red). The next section is titled "Book Copies" and contains a table:

Copy ID	Status	Action
BBB1-1	Not Available	Remove
BBB1-2	Available	Remove
BBB1-3	Available	Remove

*Figure 4.17: Book listing page rendered for the user Sam*

Now, if you logged in as *Alex*, you would be able to see the following screen:



*Figure 4.18: Book listing page rendered for the user Alex*

## Observability

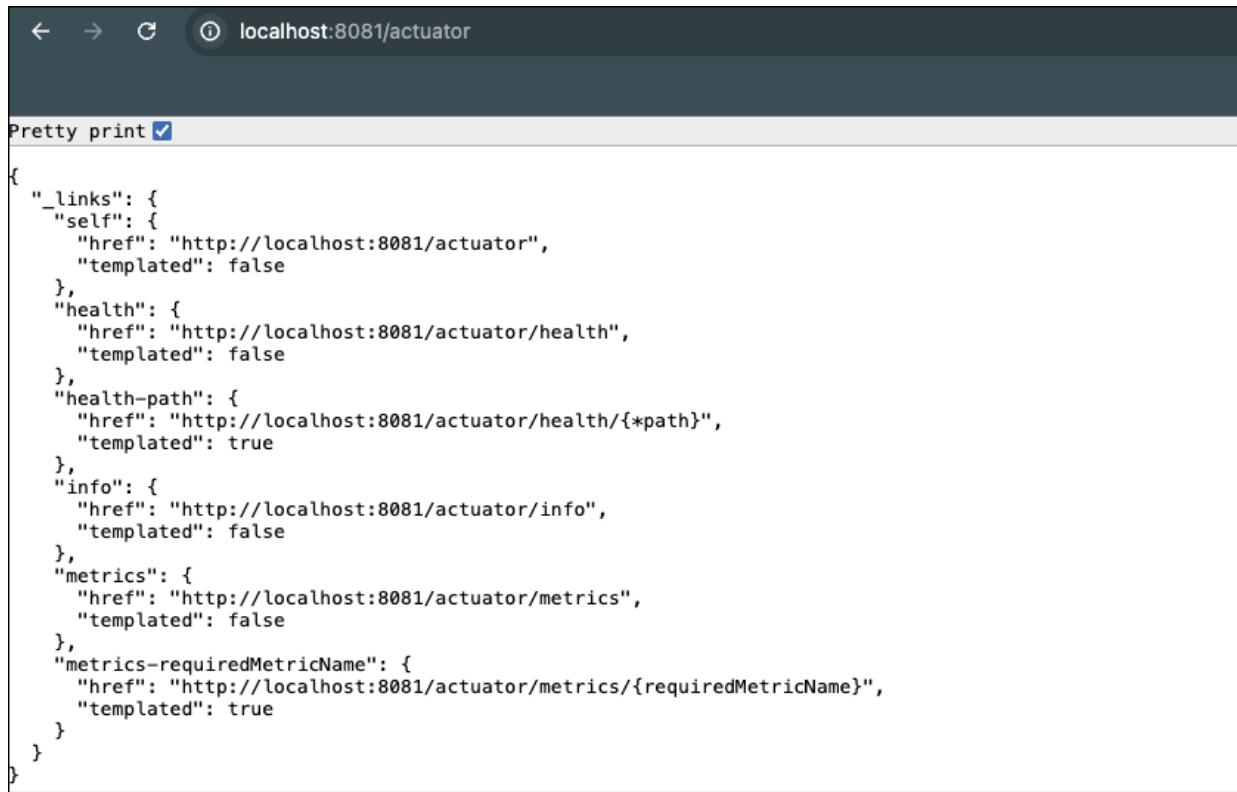
We have already discussed observability in a Spring MVC application in the previous chapter, so we will skip repeating those details here. You can easily enable observability for your application by adding the following Spring Boot Actuator starter dependency to your project:

1. <dependency>
2.    <groupId>org.springframework.boot</groupId>
3.    <artifactId>spring-boot-starter-actuator</artifactId>
4. </dependency>

Next, add the following configuration to your **application.properties** file to enable specific actuator endpoints on a different port than the one used to serve user requests:

1. management.server.port=8081
2. management.endpoints.web.exposure.include=metrics,info,health

Now, if you visit **http://localhost:8081/actuator**, you should see the following screen with the configured actuator endpoints:



A screenshot of a browser window showing the JSON response of the `/actuator` endpoint. The URL in the address bar is `localhost:8081/actuator`. The response is a well-structured JSON object with various links to other actuator endpoints. The JSON is displayed with "Pretty print" checked.

```
Pretty print   
{  
  "_links": {  
    "self": {  
      "href": "http://localhost:8081/actuator",  
      "templated": false  
    },  
    "health": {  
      "href": "http://localhost:8081/actuator/health",  
      "templated": false  
    },  
    "health-path": {  
      "href": "http://localhost:8081/actuator/health/{*path}",  
      "templated": true  
    },  
    "info": {  
      "href": "http://localhost:8081/actuator/info",  
      "templated": false  
    },  
    "metrics": {  
      "href": "http://localhost:8081/actuator/metrics",  
      "templated": false  
    },  
    "metrics-requiredMetricName": {  
      "href": "http://localhost:8081/actuator/metrics/{requiredMetricName}",  
      "templated": true  
    }  
  }  
}
```

**Figure 4.19:** The actuator endpoint

## Conclusion

In this chapter, we introduced the Spring MVC, and its most important features required to implement a web application. We also introduced our sample use case and gradually implemented it using Spring MVC and applied security and observability.

In the next chapter, we will learn the Spring Framework's data access capabilities and add data access to the same use case.

## Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



*OceanofPDF.com*

# CHAPTER 5

# Working with Spring Data Access

## Introduction

In this chapter, we will introduce you to the Spring Data project and explore various options available for implementing persistence, such as JDBC and JPA. We will begin by implementing the persistence layer of the sample application using the Spring JDBC template, followed by utilizing JPA and Hibernate as the secondary options. We will also discuss how to use MongoDB using the Spring Data capabilities. Additionally, we will cover techniques for testing applications both with and without databases.

## Structure

The chapter covers the following topics:

- Data access in Spring
- Spring JDBC Data Access
- Spring Data JPA Data Access
- Spring Data MongoDB Data Access

## Objectives

By the end of this chapter, you will understand the key data access features framework, Spring Boot, and Spring Data projects. You will also learn about how to use Spring JDBC and Spring JPA features practically. Furthermore,

we will also look at how to work with MongoDB and write test cases.

## Data access in Spring

The BookClub sample application we developed in the last chapter includes user interfaces, login capabilities, and essential security and observability features. However, it lacks persistence, as we relied on a simple Java map to store books, which is not a true persistence mechanism. In this chapter, we will explore several key data persistence mechanisms supported by Spring and apply them to our sample project. The Spring Framework provides the foundational, low-level data access capabilities, while the Spring Data project offers advanced features for integrating Spring applications with both relational and non-relational data sources. Before proceeding, let us understand what each of these projects offers.

Data Access features provided by the Spring core framework include the following:

- **Consistent programming model:** Spring offers a consistent programming model across various data access technologies, enabling seamless transitions between them or the ability to use multiple technologies within the same application
- **Exception handling:** Spring provides a consistent exception hierarchy for data access exceptions, simplifying error management across technologies like JDBC and JPA. The core of this hierarchy is the **DataAccessException** class, an extension of Java's **RuntimeException**. This design reduces boilerplate error-handling code while allowing targeted exception handling where needed
- **Transaction management:** Spring supports both programmatic and declarative transaction management, designed to overcome the limitations of traditional global and local transaction models in JavaEE. It is not tied to any specific execution environment, such as an application server, or to a specific programming model like EJB
- **JDBC support:** Spring provides an abstraction layer over JDBC, including features like **JdbcTemplate** and **NamedParameterJdbcTemplate**, which minimize the need for extensive JDBC coding and automatically handle database connections.

- **ORM integration:** Spring has built-in support for integrating with popular **Object-Relational Mapping (ORM)** frameworks such as JPA, Hibernate, and JDO.

The Spring Data project builds on the foundational data access features of the Spring Core Framework and enhances them with a robust set of tools to improve productivity and maintain consistency. Key features include the following:

- **Repository abstraction:** Spring Data offers a repository abstraction that spans various data access technologies like JDBC and JPA. By extending predefined interfaces, developers can quickly create data access layers with built-in support for powerful CRUD operations, pagination, sorting, and custom query methods.
- **Query methods:** Spring Data simplifies query creation by allowing developers to define queries directly through method names in repository interfaces. These method names adhere to a specific convention that Spring Data interprets to automatically generate the necessary queries.
- **Various data source integration:** Beyond JDBC and JPA, Spring Data integrates seamlessly with a wide range of popular data sources, including MongoDB, Redis, LDAP, Elasticsearch, Cassandra, Geode, Couchbase, and Neo4j.
- **Auditing:** Spring Data includes built-in support for auditing changes to entities. It can automatically populate fields such as **createdBy**, **createdDate**, **lastModifiedBy**, and **lastModifiedDate**, which is invaluable for maintaining accurate records for audit purposes.

Finally, the Spring Boot project offers a suite of capabilities that streamline configuration and enable production-ready features with minimal effort. The key features include the following:

- **Starter dependencies:** Spring Boot simplifies dependency management by providing starter dependencies such as **spring-boot-starter-data-jdbc** and **spring-boot-starter-data-jpa**, bundling all the necessary components for data access.
- **Auto-configuration:** Spring Boot automatically configures data access components based on the detected dependencies in the classpath, significantly reducing the need for manual setup compared to the Spring

Framework core.

- **Embedded database support:** Spring Boot can automatically configure and initiate an embedded database like H2, HSQLDB, or Derby, making it convenient for testing and development environments.
- **Property-based configuration:** Spring Boot facilitates easy configuration of data sources and other data access components through **application.properties** or **application.yml** files, simplifying the externalization of database-related configuration parameters.

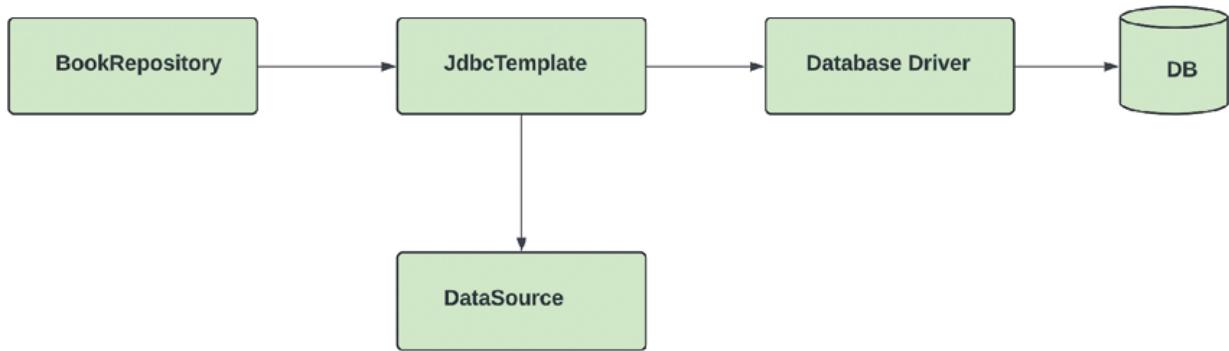
## Spring JDBC Data Access

**Java Database Connectivity (JDBC)** is the standard API in Java for connecting to and interacting with databases through queries and updates. Database vendors provide JDBC drivers that serve as intermediaries between the database and the JDBC API. Writing JDBC code requires developers to handle several low-level tasks, such as managing connections, creating Statements or PreparedStatements, executing queries, processing ResultSets, mapping results to Java objects, and managing exceptions based on the underlying database driver. These tasks often result in lengthy, boilerplate code, and developers must carefully manage resources using try-catch-finally blocks to avoid resource leaks.

As discussed, Spring's data access features are designed to reduce this boilerplate code and simplify resource management and complex exception handling. When using Spring JDBC capabilities, the framework takes over the responsibilities of opening, closing, and managing connection pools, leaving developers to simply provide connection parameters and configuration. After providing an SQL statement and its values, Spring handles the preparation and execution of the statement. During response processing, Spring iterates through the result set, allowing developers to focus only on mapping the result to a Java bean or any other action.

Furthermore, Spring ensures that all resources are properly released, translates technology-specific exceptions into its own data access exception hierarchy, and manages transactions. The `JdbcTemplate` class is central to Spring's JDBC core package and is the most common way to work with JDBC in Spring. Other JDBC tools, like `NamedParameterJdbcTemplate`,

also use `JdbcTemplate` under the hood. Spring relies on a special bean called `DataSource`, which serves as a factory for database connections, abstracting the complexities of connection creation and management. The following figure shows how a database is connected to an application:



*Figure 5.1: Data access runtime view*

We will explore these Spring JDBC features using the `BookClub` sample application from the previous chapter. As before, the first step is to add the necessary dependencies to the project, specifically `spring-boot-starter-jdbc` and the H2 database, as shown in the following code snippet. In this case, H2 will serve as the embedded database for our sample application.

```
1. <dependency>
2. <groupId>org.springframework.boot</groupId>
3. <artifactId>spring-boot-starter-jdbc</artifactId>
4. </dependency>
5.
6. <dependency>
7. <groupId>com.h2database</groupId>
8. <artifactId>h2</artifactId>
9. <scope>runtime</scope>
10. </dependency>
```

When you add the above dependencies, Spring Boot's auto-configuration mechanism detects the presence of `spring-boot-starter-jdbc` and H2 in the classpath and automatically configures an in-memory H2 database for your application. It also creates a `DataSource` bean with the connection parameters for the embedded H2 database, along with a `JdbcTemplate` configured to interact with it. Additionally, Spring Boot configures HikariCP as the default connection pool and sets up a transaction manager.

The next step involves creating the database schema and populating it with

initial data. While there are several approaches to achieve this, we will use Flyway due to its production readiness and rich feature set, including database schema migration capabilities.

**Tip:** Flyway is an open-source database migration tool that helps manage and version database schema changes. It allows developers to evolve database schemas reliably across different environments by using versioned SQL migration scripts or Java-based migrations.

To configure Flyway in our application, we simply need to add the following dependency to the project's POM file, and Spring Boot's auto-configuration will handle the rest:

1. <dependency>
2. <groupId>org.flywaydb</groupId>
3. <artifactId>flyway-core</artifactId>
4. </dependency>

Once the dependency is added, proceed by creating the directory structure **src/main/resources/db/migration** as specified by Flyway's configuration, and add a file named **V1\_\_Init.sql** with the following database schema:

1. `DROP TABLE book IF EXISTS;`
2. `DROP TABLE book_copy IF EXISTS;`
3.
4. `CREATE TABLE book (`
5.  `book_id INT NOT NULL PRIMARY KEY,`
6.  `isbn VARCHAR(255) NOT NULL,`
7.  `title VARCHAR(100) NOT NULL,`
8.  `author VARCHAR(100) NOT NULL,`
9.  `publication_year INT NOT NULL`
10. `);`
11.
12. `CREATE TABLE book_copy (`
13.  `copy_id INT NOT NULL PRIMARY KEY,`
14.  `book_id INT NOT NULL,`
15.  `is_available BOOLEAN NOT NULL,`
16.  `CONSTRAINT fk_book FOREIGN KEY (book_id) REFERENCES book(book_id)`
17. `);`

Additionally, create another file, **V2\_\_Add\_initial\_book\_data**, with the following entries to insert the sample data:

1. `INSERT INTO`
2.  `book (book_id, isbn, title, author, publication_year)`
3. `VALUES`

4. (1001,'9789355511263','Mastering Java Persistence API', 'Nisha Parameswaran',2022),
5. (1002,'9789389845143','JAVA Programming Simplified', 'Muneer Ahmad',2020 ),
6. (1003,'9789391392475','Software Design Patterns for Java Developers', 'Lalit Mehra', 2022),
7. (1004,'9789389423655','JavaScript for Gurus', 'Ockert Preez',2022),
8. (1005,'9789389845204','Fundamentals of Android App Development', 'Sujit Kumar',2022);
- 9.

Next, define the **BookRepository** interface as follows:

```

1. package com.bpb.hssb.ch5.bookclub.repository;
2.
3. import com.bpb.hssb.ch5.bookclub.domain.Book;
4. import com.bpb.hssb.ch5.bookclub.domain.BookCopy;
5.
6. public interface BookRepository {
7.
8.     public Book saveBook(Book book);
9.
10.    public Book findBookById(int bookId);
11.
12.    public Iterable<Book> findAllBooks();
13.
14.    public Book deleteBook(int bookId);
15.
16.    public BookCopy saveBookCopy(Book book);
17.
18.    public BookCopy findBookCopyById(int bookCopyId);
19.
20.    public BookCopy deleteBookCopy(int bookId, int bookCopyId);
21. }
```

The next step is to create the implementation class called **JDBCBookRepository** and use **JdbcTemplate** to write our data access logic. We can start by adding **JDBCBookRepository.java** with the following code:

```

1. package com.bpb.hssb.ch5.bookclub.repository;
2.
3. import org.springframework.jdbc.core.JdbcTemplate;
4.
5. public class JDBCBookRepository implements BookRepository {
```

```
6.  
7.     private JdbcTemplate jdbcTemplate;  
8.  
9.     public JDBCBookRepository(JdbcTemplate jdbcTemplate) {  
10.         this.jdbcTemplate = jdbcTemplate;  
11.     }  
12.  
13. }
```

In the code, we have injected **JdbcTemplate** into the **JDBCBookRepository** using the constructor injection. Next, to add a new book to the database, implement the **saveBook** method:

```
1.     @Override  
2.     public Book saveBook(Book book) {  
3.  
4.         String sql = "INSERT INTO book (book_id, isbn,  
5.             title, author, publication_year) VALUES (?, ?, ?, ?, ?)";  
6.         int bookId = getBookId();  
7.         book.setBookId(bookId);  
8.         jdbcTemplate.update(sql, book.getBookId(),  
9.             book.getIsbn(), book.getTitle(), book.getAuthor(),  
10.            book.getPublicationYear());  
11.         return book;  
12.     }
```

In the code, we provide the SQL statement and corresponding values to the update method of **JdbcTemplate**. We use the ? placeholders in the SQL statement and pass the actual values separately to the update method, ensuring that any potential SQL injection attacks are avoided

Let us also look at the **findAllBooks** method to see how we can query data using **jdbcTemplate**. In the following code, we have provided the SQL statement to the query method of **jdbcTemplate** and used a lambda expression as the **RowMapper** to map each row of the result set into a Book object:

```
1.     @Override  
2.     public Iterable<Book> findAllBooks() {  
3.  
4.         String sql = "SELECT book_id, isbn, title, author, publication_year FROM book";  
5.         List<Book> books = jdbcTemplate.query(  
6.             sql,  
7.             (rs, rowNum) -> new Book(
```

```

8.         rs.getInt("book_id"),
9.         rs.getString("isbn"),
10.        rs.getString("title"),
11.        rs.getString("author"),
12.        rs.getInt("publication_year"),
13.        new ArrayList<>());
14.    return books;
15. }

```

The method for retrieving a copy of a book is provided as follows:

```

1.  @Override
2.  public BookCopy findBookCopyById(int copyId) {
3.
4.    String sql = "SELECT copy_id, book_id, is_available FROM book_copy WHERE copy_id
   = ?";
5.    return jdbcTemplate.queryForObject(
6.      sql,
7.      (rs, rowNum) -> new BookCopy(
8.        rs.getInt("copy_id"),
9.        findBookById(rs.getInt("book_id")),
10.       rs.getBoolean("is_available")),
11.       copyId);
12. }

```

The remaining methods, **findBookById**, **deleteBook**, **saveBookCopy**, and **deleteBookCopy**, also follow the same style, and you can refer to them in the companion GitHub repository for this book. Finally, we can define the Spring bean configuration by updating the **BookclubConfiguration** as shown in the following code:

```

1. package com.bpb.hssb.ch5.bookclub.config;
2.
3. import org.springframework.context.annotation.Bean;
4. import org.springframework.context.annotation.Configuration;
5. import org.springframework.jdbc.core.JdbcTemplate;
6.
7. import com.bpb.hssb.ch5.bookclub.repository.BookRepository;
8. import com.bpb.hssb.ch5.bookclub.repository.JDBCBookRepository;
9. import com.bpb.hssb.ch5.bookclub.service.BookService;
10. import com.bpb.hssb.ch5.bookclub.service.BookServiceImpl;
11.
12. @Configuration

```

```
13. public class BookclubConfiguration {  
14.  
15.     @Bean  
16.     public BookRepository bookRepository(JdbcTemplate jdbcTemplate) {  
17.         return new JDBCBookRepository(jdbcTemplate);  
18.     }  
19.  
20.     @Bean  
21.     public BookService bookService(BookRepository bookRepository) {  
22.         return new BookServiceImpl(bookRepository);  
23.     }  
24. }
```

As we emphasize **test-driven development (TDD)** throughout this book, it is important to discuss how to write effective unit tests for our JDBC repository implementation. Among the various options for writing JDBC tests, we will use the **@DataJdbcTest** annotation to create our test cases. This annotation represents a test slice that configures an in-memory embedded database, **JdbcTemplate**, and provides transactional support by rolling back the changes after each test.

You can begin writing the test class with the following minimal code:

```
1. package com.bpb.hssb.ch5.bookclub.repository;  
2.  
3. import org.springframework.beans.factory.annotation.Autowired;  
4. import org.springframework.boot.test.autoconfigure.jdbc.JdbcTest;  
5. import org.springframework.context.annotation.Import;  
6. import org.springframework.jdbc.core.JdbcTemplate;  
7.  
8. import com.bpb.hssb.ch5.bookclub.config.BookclubConfiguration;  
9.  
10. import static org.assertj.core.api.Assertions.assertThat;  
11.  
12. @JdbcTest  
13. @Import({ BookclubConfiguration.class })  
14. class BookRepositoryTest {  
15.  
16.     @Autowired  
17.     private BookRepository bookRepository;  
18.  
19.     @Autowired
```

```
20. private JdbcTemplate jdbcTemplate;
21.
22. }
```

In the code, the **@JdbcTest** annotation configures an in-memory embedded database and sets up a **JdbcTemplate** bean. However, it does not load other components into the **ApplicationContext**, which is why we needed to explicitly use the **@Import** annotation to load the bean definitions from **BookclubConfiguration**. Additionally, we have injected **BookRepository** and **JdbcTemplate** to be used in our test cases.

Now, you can define your first test case using the following code:

```
1. @Test
2. void testCreateBook() {
3.
4.     Book book = Book.builder().isbn("9789355511260").
    title("Spring Testing")
5.     .author("SpringTeam").publicationYear(2022).build();
6.     bookRepository.saveBook(book);
7.
8.     String sql = "SELECT COUNT(*)
    FROM book WHERE isbn = ? AND title = ?";
9.     Integer count = jdbcTemplate.queryForObject(sql,
    Integer.class, book.getIsbn(), book.getTitle());
10.    assertThat(count).isEqualTo(1);
11. }
```

In the code, we first created an instance of **Book** and saved it using the **BookRepository**. Later in the method, we used **JdbcTemplate** to verify that the book has persisted in the database properly. Let us also look at one more test method:

```
1. @Test
2. void testGetBook() {
3.
4.     jdbcTemplate.execute(
5.         "INSERT INTO book (book_id, isbn, title, author, publication_year) VALUES (2001,'978935
5511273',
6.         'Java Testing','SpringTeam',2022)");
7.     Book book = bookRepository.findBookById(2001);
8.     assertThat(book).isNotNull();
9.     assertThat(book.getBookId()).isEqualTo(2001);
```

```
10. assertThat(book.getIsbn()).isEqualTo("9789355511273");
11. assertThat(book.getTitle()).isEqualTo("Java Testing");
12. assertThat(book.getAuthor()).isEqualTo("SpringTeam");
13. assertThat(book.getPublicationYear()).isEqualTo(2022);
14. }
```

In contrast to the previous test case, the aforementioned code first inserts a specific book instance into the database using the **JdbcTemplate** and then uses the **BookRepository** to retrieve it. You can refer to the complete test class with other test cases in the companion GitHub repository for this book. So far, we have focused on **JdbcTemplate** because it is the most popular and core JDBC abstraction provided by the Spring Framework. However, it is worth discussing **NamedParameterJdbcTemplate** briefly. **NamedParameterJdbcTemplate** offers more readable and maintainable code, especially for complex queries, by replacing '?' placeholders with named parameters like '**:paramName**'. The following code segment shows the **saveBook** method using **NamedParameterJdbcTemplate**. As an exercise, you can convert the other methods in our repository class to use **NamedParameterJdbcTemplate**.

```
1. @Override
2. public Book saveBook(Book book) {
3.
4.     String sql = "INSERT INTO book (book_id, isbn, title, author, publication_year) VALUES
5.                 (:bookId, :isbn, :title, :author, :publicationYear)";
6.
7.     int bookId = getBookId();
8.     book.setBookId(bookId);
9.     Map<String, Object> parameters = new HashMap<>();
10.    parameters.put("bookId", book.getBookId());
11.    parameters.put("isbn", book.getIsbn());
12.    parameters.put("title", book.getTitle());
13.    parameters.put("author", book.getAuthor());
14.    parameters.put("publicationYear", book.getPublicationYear());
15.    namedParameterJdbcTemplate.update(sql, parameters);
16.
17.    return book;
18. }
```

Before we wrap up this section, we need to discuss an important topic:

connecting to an actual database. So far, we have been using the embedded H2 database, which is not suitable for production. Spring Boot significantly reduces the effort required to connect to a production database server. All you need to do is update the **application.properties** file with the production database connection parameters and include the correct database driver as a dependency in your project. The following example shows sample values for connecting to a MySQL server. Once you provide these values, Spring Boot will stop creating the embedded database and instead configure the DataSource bean and connection pool to connect to the MySQL server.

1. `spring.datasource.url=jdbc:mysql://localhost/test`
2. `spring.datasource.username=dbuser`
3. `spring.datasource.password=dbpass`

In this section, we have only utilized the JDBC capabilities of the Spring Framework along with Spring Boot's features, without tapping into any Spring Data functionalities. If we had used Spring Data JDBC features, we could have avoided writing any SQL code or repository methods entirely. In the next section, we will explore Spring Data features with JPA.

## Spring Data JPA Data Access

**Java Persistence API (JPA)** is a Java specification for accessing, persisting, and managing data between Java objects and a relational database, providing a standardized approach to handle relational data in Java applications. The process of mapping Java objects to relational database tables is known as **Object-Relational Mapping (ORM)**. JPA is a specification, not a complete implementation, and requires an implementation to function. Popular JPA implementations include Hibernate, EclipseLink, and TopLink.

In this section, we will again use the BookClub sample from the previous chapter to explore JPA and Spring Data capabilities. First, you need to add the following dependencies to the project's POM file:

1. `</dependency>`
2. `<dependency>`
3. `<groupId>org.springframework.boot</groupId>`
4. `<artifactId>spring-boot-starter-data-jpa</artifactId>`
5. `</dependency>`
6. `<dependency>`

```
7. <groupId>com.h2database</groupId>
8. <artifactId>h2</artifactId>
9. <scope>runtime</scope>
10. </dependency>
```

The dependencies above enable Spring Boot to auto-configure everything needed to use Spring Data JPA with Hibernate as the default JPA implementation. They also set up H2 as the embedded database and configured the necessary connectivity beans.

To use JPA in our application, we need to annotate our domain classes with the annotations provided by JPA. Let us start with the **Book** class:

```
1. package com.bpb.hssb.ch5.bookclub.domain;
2.
3. import java.util.List;
4.
5. import jakarta.persistence.CascadeType;
6. import jakarta.persistence.Entity;
7. import jakarta.persistence.GeneratedValue;
8. import jakarta.persistence.GenerationType;
9. import jakarta.persistence.Id;
10. import jakarta.persistence.OneToMany;
11. import jakarta.validation.constraints.Max;
12. import jakarta.validation.constraints.Min;
13. import jakarta.validation.constraints.NotBlank;
14. import jakarta.validation.constraints.NotNull;
15. import jakarta.validation.constraints.Pattern;
16. import jakarta.validation.constraints.Size;
17. import lombok.AllArgsConstructor;
18. import lombok.Builder;
19. import lombok.Getter;
20. import lombok.NoArgsConstructor;
21. import lombok.Setter;
22. import lombok.ToString;
23.
24. @Getter
25. @Setter
26. @AllArgsConstructor
27. @NoArgsConstructor
28. @Builder
29. @ToString
```

```

30. @Entity
31. public class Book {
32.
33.     @Id
34.     @GeneratedValue(strategy = GenerationType.AUTO)
35.     private int bookId;
36.
37.     @NotBlank(message = "ISBN is required")
38.     @Pattern(regexp = "^\d{13}$", message =
39.             "ISBN must be exactly 13 digits")
40.     private String isbn;
41.
42.     @NotBlank(message = "Title is required")
43.     @Size(min = 10, max = 100, message =
44.             "Title must be between 10 and 100 characters")
45.     private String title;
46.
47.     @NotBlank(message = "Author is required")
48.     @Size(min = 10, max = 100, message =
49.             "Author name must be between 10 and 100 characters")
50.     private String author;
51.
52.     @NotNull(message = "Publication year is required")
53.     @Min(value = 1000, message =
54.             "Publication year must be after 1000")
55.     @Max(value = 9999, message =
56.             "Publication year must be before 9999")
57.     private int publicationYear;
58.
59.     @OneToMany(mappedBy = "book", cascade =
60.             CascadeType.ALL, orphanRemoval = true)
61.     private List<BookCopy> copies;
62.
63. }

```

In the above code, we have annotated the Book class with the **@Entity** annotation. This JPA annotation marks the class as an entity with metadata for persistence in a relational database, including primary keys, relationships, and other JPA-specific configurations, and it is managed by **EntityManager**. Additionally, it maps the class to a database table with the same name. If desired, you can override this default table name mapping using the **@Table** annotation. Similar to the **@Table** annotation, you can use

**@Column** annotation to change the default mapping between the field names and the columns of the database table.

The **@Id** annotation, used with **bookId**, designates the field as the primary key of the entity, uniquely identifying each row in the database table. The **@GeneratedValue** annotation specifies that the primary key value should be automatically generated and defines the strategy for this generation. In this case, **strategy = GenerationType.AUTO** allows the persistence provider to choose the most appropriate strategy for the database. We have also used the **@OneToMany** annotation to indicate that one **Book** can have many **BookCopy** entities, specifying a one-to-many relationship between the entities. Additionally, we have specified that all cascade operations should be applied to the related **BookCopy** entities. For example, if you delete a Book instance, all its associated **BookCopy** instances will also be deleted, ensuring that no orphan copies are left behind.

Here is the **BookCopy** class with JPA annotations:

```
1. package com.bpb.hssb.ch5.bookclub.domain;
2.
3. import jakarta.persistence.Entity;
4. import jakarta.persistence.GeneratedValue;
5. import jakarta.persistence.GenerationType;
6. import jakarta.persistence.Id;
7. import jakarta.persistence.JoinColumn;
8. import jakarta.persistence.ManyToOne;
9. import lombok.AllArgsConstructor;
10. import lombok.Builder;
11. import lombok.Getter;
12. import lombok.NoArgsConstructor;
13. import lombok.Setter;
14. import lombok.ToString;
15.
16. @Getter
17. @Setter
18. @AllArgsConstructor
19. @NoArgsConstructor
20. @Builder
21. @ToString
22. @Entity
23. public class BookCopy {
```

```
24.  
25. @Id  
26. @GeneratedValue(strategy = GenerationType.AUTO)  
27. private int copyId;  
28.  
29. @ManyToOne  
30. @JoinColumn(name = "book_id", nullable = false)  
31. private Book book;  
32.  
33. private boolean isAvailable;  
34.  
35. }
```

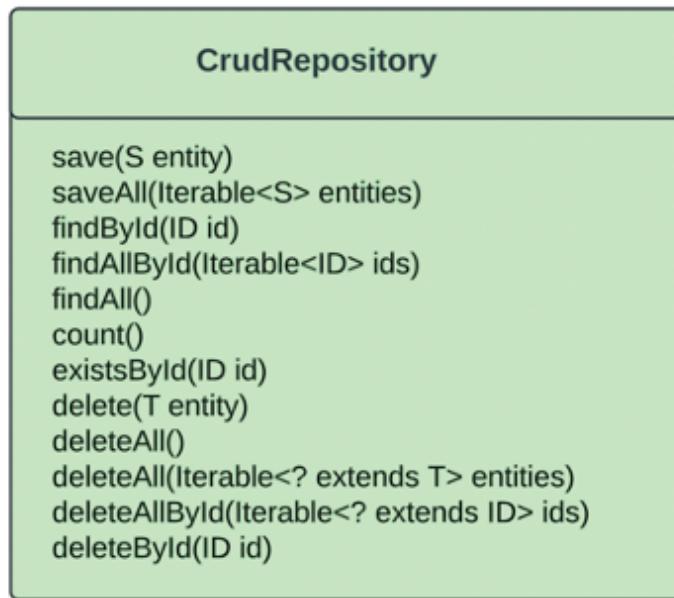
In the above code, in addition to the JPA annotations we have already discussed, you can see the use of the **@ManyToOne** annotation to indicate that many **BookCopy** entities can be associated with one **Book** entity. The **@ManyToOne** annotation represents a many-to-one relationship between two entities. We also used the **@JoinColumn** annotation to specify the foreign key column in the database table.

As the next step, you can define the repository interface for the **Book** class:

```
1. package com.bpb.hssb.ch5.bookclub.repository;  
2.  
3. import org.springframework.data.repository.CrudRepository;  
4. import java.util.List;  
5.  
6. import com.bpb.hssb.ch5.bookclub.domain.Book;  
7.  
8. public interface BookRepository extends  
    CrudRepository<Book, Integer> {  
9.  
10. List<Book> findByAuthor(String author);  
11.  
12. List<Book> findByTitleOrAuthor(String title, String author);  
13.  
14. }
```

Unlike the previous repository interface we defined in the JDBC section, we have not specified any CRUD (create, update, read, delete) methods in the code above. The CRUD methods are inherited from the **CrudRepository** interface; we just need to specify the domain object type that the repository should manage and the type of the domain object's ID. The **CrudRepository**

interface is illustrated in the following figure:



*Figure 5.2: CURDRepository interface*

When you create a repository interface that extends **JpaRepository** or **CrudRepository**, Spring Data JPA dynamically generates an implementation of that interface at runtime, so you only need to define the repository interface. Spring Data JPA uses proxy objects to create implementations of the repository interfaces. These proxies intercept method calls and translate them into the appropriate database queries.

An important point regarding the **CrudRepository** interface is that the `save` method is used for both inserting new records and updating existing ones. The determination of whether to insert or update is made primarily based on the presence and value of the entity's primary key, typically annotated with the `@Id` annotation. If the field annotated with `@Id` is null or 0 (for primitive numeric types like `long` and `int`), it is treated as a new record and will be inserted. Otherwise, it is considered an existing entry, and the corresponding record will be updated.

Additionally, you might wonder about the `findByAuthor` and `findByTitleOrAuthor` methods and how they work. Spring Data JPA parses these method names to understand the query intent and automatically generates the appropriate **Java Persistence Query Language (JPQL)**. This feature is very useful for defining query methods based on our requirements.

Here is the repository interface for **BookCopy**:

```
1. package com.bpb.hssb.ch5.bookclub.repository;
2.
3. import org.springframework.data.repository.CrudRepository;
4.
5. import com.bpb.hssb.ch5.bookclub.domain.BookCopy;
6.
7. public interface BookCopyRepository extends
   CrudRepository<BookCopy, Integer> {
8.
9. }
10.
```

In addition to the **CrudRepository**, there are a couple of more important repositories available for different purposes, which include:

- **PagingAndSortingRepository**: This repository extends from the **CrudRepository** and provides additional methods for pagination and sorting of data.
- **JpaRepository**: This repository extends from the **PagingAndSortingRepository** and provides additional JPA-specific methods like **flush()**, **saveAndFlush()**, **deleteInBatch()**.
- **ListCrudRepository**: This repository extends from the **CrudRepository** and returns **List** instead of **Iterable** for methods returning multiple entities.
- **ListPagingAndSortingRepository**: This repository extends from the **PagingAndSortingRepository** and returns **List** instead of **Iterable** for methods returning multiple entities.

At this point, we have completed all the main tasks related to using JPA in our sample application. However, you will need to make a slight adjustment to the **BookServiceImpl** class to reflect the method name changes in the repository interfaces. The following listing provides a sample code segment:

```
1. package com.bpb.hssb.ch5.bookclub.service;
2.
3. import com.bpb.hssb.ch5.bookclub.domain.Book;
4. import com.bpb.hssb.ch5.bookclub.repository.BookCopyRepository;
5. import com.bpb.hssb.ch5.bookclub.repository.BookRepository;
6.
```

```

7. public class BookServiceImpl implements BookService {
8.
9.     private BookRepository bookRepository;
10.    private BookCopyRepository bookCopyRepository;
11.
12.    public BookServiceImpl(BookRepository bookRepository,
13.                           BookCopyRepository bookCopyRepository) {
13.        this.bookRepository = bookRepository;
14.        this.bookCopyRepository = bookCopyRepository;
15.    }
16.
17.    @Override
18.    public Book createBook(Book book) throws BookServiceException {
19.        return bookRepository.save(book);
20.    }
21.
22.    @Override
23.    public Book getBook(int bookId) throws BookServiceException {
24.        return bookRepository.findById(bookId).get();
25.    }
26.
27. }

```

Finally, here is our updated **BookclubConfiguration**:

```

1. package com.bpb.hssb.ch5.bookclub.config;
2.
3. import org.springframework.context.annotation.Bean;
4. import org.springframework.context.annotation.Configuration;
5.
6. import com.bpb.hssb.ch5.bookclub.repository.BookCopyRepository;
7. import com.bpb.hssb.ch5.bookclub.repository.BookRepository;
8. import com.bpb.hssb.ch5.bookclub.service.BookService;
9. import com.bpb.hssb.ch5.bookclub.service.BookServiceImpl;
10.
11. @Configuration
12. public class BookclubConfiguration {
13.
14.     @Bean
15.     public BookService bookService(BookRepository bookRepository,
16.                                   BookCopyRepository bookCopyRepository) {
16.         return new BookServiceImpl(bookRepository, bookCopyRepository);
17.     }

```

```
18. }
```

Now, if you run the application, it will work without any issues. Spring Boot not only automatically configures an embedded database but also generates the correct database schema. You can view the automatically generated schema and queries by adding the following properties to the **application.properties** file. This is also useful for debugging purposes.

1. `spring.jpa.show-sql=true`
2. `spring.jpa.properties.hibernate.format_sql=true`

However, in a production deployment, we need control over manually creating the database and populating it with initial data. The automatic schema generation can be disabled by adding the following property to the **application.properties** file:

1. `spring.jpa.hibernate.ddl-auto=none`

Additionally, you can disable the embedded database and connect to the production database using the following configuration. This will set up the connection pool to connect with the MySQL server:

1. `spring.datasource.url=jdbc:mysql://localhost/test`
2. `spring.datasource.username=dbuser`
3. `spring.datasource.password=dbpass`

As the final topic of this section, let us look at how to write test cases for JPA. We have a few options, but we will use the **@DataJpaTest** annotation to write our JPA test cases. You can start by adding the following code to create your test cases:

```
1. package com.bpb.hssb.ch5.bookclub.repository;
2.
3. import org.springframework.beans.factory.annotation.Autowired;
4. import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;
5.
6.
7. import jakarta.persistence.EntityManager;
8.
9. @DataJpaTest
10. public class BookRepositoryTest {
11.
12.     @Autowired
13.     private BookRepository bookRepository;
14.
15.     @Autowired
```

```
16. private EntityManager entityManager;  
17.  
18. }
```

In the above code, **@DataJpaTest** is used for testing JPA components by scanning for classes annotated with **@Entity** annotation and configuring Spring Data JPA repositories accordingly. It also provides **TestEntityManager** and disables full auto-configuration to make test execution more efficient. By default, it configures an in-memory embedded database for testing, replacing any explicit or auto-configured **DataSource**. If required, this behavior can be adjusted using the **@AutoConfigureTestDatabase** annotation. **@DataJpaTest** annotation also enables transaction support and ensures that changes are rolled back at the end of each test. Additionally, we have injected **BookRepository** and **EntityManager** to be used with our test methods.

Now, you can add test cases. Here is the **test** method for the **createBook** method:

```
1. @Test  
2. void testCreateBook() {  
3.  
4.     Book book = Book.builder().isbn("9789355511260").title("Spring Testing")  
5.         .author("SpringTeam").publicationYear(2022).build();  
6.     bookRepository.save(book);  
7.  
8.     entityManager.flush();  
9.     entityManager.clear();  
10.    Book foundBook = entityManager.find(Book.class, book.getBookId());  
11.    assertThat(foundBook).isNotNull();  
12.    assertThat(foundBook.getTitle()).isEqualTo(book.getTitle());  
13.    assertThat(foundBook.getIsbn()).isEqualTo(book.getIsbn());  
14.    assertThat(foundBook.getAuthor()).isEqualTo(book.getAuthor());  
15.    assertThat(foundBook.getPublicationYear()).isEqualTo(book.getPublicationYear());  
16. }
```

In the above **test** method, we saved an instance of **Book** using **BookRepository** and then verified it using the JPA **EntityManager**.

Next, here is the test case for the **getBook** method:

```
1. @Test  
2. public void testGetBook() {  
3.
```

```

4. entityManager.flush();
5. entityManager.clear();
6. Book book = Book.builder().isbn("9789355511260").title("Spring Testing")
7.         .author("SpringTeam").publicationYear(2022).build();
8. entityManager.persist(book);
9.
10. Book foundBook = bookRepository.findById(book.getBookId()).get();
11. assertThat(foundBook).isNotNull();
12. assertThat(book.getTitle()).isEqualTo(foundBook.getTitle());
13. assertThat(book.getIsbn()).isEqualTo(foundBook.getIsbn());
14. assertThat(book.getAuthor()).isEqualTo(foundBook.getAuthor());
15. assertThat(book.getPublicationYear()).isEqualTo(foundBook.getPublicationYear());
16. }

```

In the above test method, we saved an instance of **Book** using **EntityManager** and then retrieved it using the JPA **EntityManager**.

In this section, we learned how to enable Spring JPA for our project, annotate domain classes, and auto-generate repository implementations. In the next section, we will explore how to use Spring Data capabilities with MongoDB.

## Spring Data MongoDB Data Access

MongoDB is an open-source, document-oriented NoSQL database designed to handle large volumes of data with flexibility and scalability. To follow the sample in this section, you need to install MongoDB in your local development environment. Refer to the MongoDB official installation guide for software installation.

Once you have installed and started MongoDB locally, you can begin our sample project by adding the following dependencies to your project's POM file:

```

1. <dependency>
2. <groupId>org.springframework.boot</groupId>
3. <artifactId>spring-boot-starter-data-mongodb</artifactId>
4. </dependency>

```

The above dependency includes everything needed to interact with MongoDB using Spring Data and provides auto-configuration for connecting to a MongoDB database. It also automatically configures a **MongoTemplate**

bean, which is used for MongoDB operations such as querying, saving, and updating documents in MongoDB collections. Additionally, it supports the creation of MongoDB repositories that follow Spring Data's repository pattern, enabling simple CRUD operations and custom queries using just interfaces, as we have seen in the JPA section.

The BookClub has a requirement to allow users to add comments about books, which should also be displayed on the book's details page. We can use MongoDB to persist these comments. First, let us introduce a new domain class called **Comment**, as follows:

```
1. package com.bpb.hssb.ch5.bookclub.domain;
2.
3. import org.springframework.data.mongodb.core.mapping.Document;
4.
5. import org.springframework.data.annotation.Id;
6. import lombok.AllArgsConstructor;
7. import lombok.Builder;
8. import lombok.Getter;
9. import lombok.NoArgsConstructor;
10. import lombok.Setter;
11. import lombok.ToString;
12.
13. @Getter
14. @Setter
15. @AllArgsConstructor
16. @NoArgsConstructor
17. @Builder
18. @ToString
19. @Document(collection = "comments")
20. public class Comment {
21.
22.     @Id
23.     private String id;
24.     private int bookId;
25.     private String user;
26.     private String description;
27.
28. }
```

The **@Id (org.springframework.data.annotation.Id)** annotation used in the above code is from Spring Data Commons and is different from JPA's

**@Id (jakarta.persistence.Id)** annotation. It marks the field as the identifier field of the MongoDB document and is automatically mapped to the `_id` field in the MongoDB document. We also used the **@Document** annotation to mark the class as a domain object that should be persisted to MongoDB and to specify the collection name. Additionally, the **@Field** annotation can be used to modify the default mapping between object fields and document properties, while **@Transient** instructs that a property should not be persisted. Among these annotations, only **@Id** and **@Document** are mandatory.

Now, you can define the repository interface just as we did in the JPA section. The complete code for the repository interface is provided as follows:

```
1. package com.bpb.hssb.ch5.bookclub.repository;
2.
3. import java.util.List;
4.
5. import org.springframework.data.mongodb.repository.MongoRepository;
6.
7. import com.bpb.hssb.ch5.bookclub.domain.Comment;
8.
9. public interface CommentRepository extends
   MongoRepository<Comment, String> {
10.
11.   List<Comment> findByBookId(int bookId);
12. }
```

The above interface extends the **MongoRepository** interface, which provides CRUD methods for the repository. Similar to the JPA repository, once we define the repository interface, Spring Data automatically generates the repository implementation at runtime, so we do not need to worry about it.

Next, we can add the following two additional methods to the **BookService** interface:

```
1.   public void createComment(Comment comment);
2.
3.   public Iterable<Comment> findCommentsByBookId(int bookId);
```

Here is how you can use **commentRepository** within the **BookServiceImpl**

class:

```
1. @Override
2. public void createComment(Comment comment) {
3.     commentRepository.save(comment);
4. }
5.
6. @Override
7. public Iterable<Comment> findCommentsByBookId(int bookId) {
8.     return commentRepository.findByBookId(bookId);
9. }
```

You can refer to the complete code for this sample in the companion GitHub repository of the book. It includes the user interface for comment submission and viewing based on the Spring MVC features we discussed in the previous chapter.

Before we conclude this section, let us also look at how to write test cases for a MongoDB repository. We can use the **@DataMongoTest** annotation for writing these test cases. Here is the minimal code to get started with the test class:

```
1. package com.bpb.hssb.ch5.bookclub.repository;
2.
3. import org.junit.jupiter.api.BeforeEach;
4. import org.springframework.beans.factory.annotation.Autowired;
5. import org.springframework.boot.test.autoconfigure.data.mongo.DataMongoTest;
6. import org.springframework.data.mongodb.core.MongoTemplate;
7. import org.springframework.data.mongodb.core.query.Criteria;
8. import org.springframework.data.mongodb.core.query.Query;
9. import com.bpb.hssb.ch5.bookclub.domain.Comment;
10.
11. @DataMongoTest
12. public class CommentRepositoryTest {
13.
14.     @Autowired
15.     private CommentRepository commentRepository;
16.
17.     @Autowired
18.     private MongoTemplate mongoTemplate;
19.
20.     @BeforeEach
```

```
21. public void init(){  
22.     mongoTemplate.remove(new Query(), Comment.class);  
23. }  
24.  
25. }  
26.
```

In the above code, we used the **@DataMongoTest** annotation, which configures only MongoDB data access components like **MongoTemplate** and **MongoRepository**, without loading other Spring components. We then injected **MongoTemplate**, a low-level utility for interacting with MongoDB, along with the **CommentRepository**.

Now, we can write our first test case for the **createComment** method as follows:

```
1. @Test  
2. public void testCreateComment() {  
3.  
4.     Comment comment = Comment.builder().bookId(5001).description("Good book").user("Sa  
m").build();  
5.     commentRepository.save(comment);  
6.  
7.     Query query = new Query();  
8.     query.addCriteria(Criteria.where("bookId").is(5001));  
9.     List<Comment> foundComment = mongoTemplate.find(query, Comment.class);  
10.    assertThat(foundComment.size()).isEqualTo(1);  
11. }
```

In the above code, we saved an instance of **Comment** class using the **CommentRepository** and then verified it in MongoDB using the **MongoTemplate**.

Here is the code for the **findById** method:

```
1.  
2. @Test  
3. public void testFindByBookId() {  
4.  
5.     Comment comment = Comment.builder().bookId(5001).description("Good book").user("Sa  
m").build();  
6.  
7.     mongoTemplate.save(comment);  
8.  
9.     List<Comment> foundComments = commentRepository.findById(comment.getBookI
```

```
    d());
8.     assertThat(foundComments.size()).isEqualTo(1);
9.     assertThat(foundComments.get(0).getDescription()).isEqualTo(comment.getDescription());
10.    assertThat(foundComments.get(0).getUser()).isEqualTo(comment.getUser());
11.
12. }
```

In the above code, we first used **MongoTemplate** to save a comment in MongoDB and then used our **CommentRepository** to retrieve it.

In this section, we discussed how to write data access logic and test cases for MongoDB using the capabilities offered by Spring Data and Spring Boot projects.

Regarding the observability aspects of the data access layer, the health endpoints we covered in *Chapter 3, Spring Essentials for Enterprise Applications* provide health information about the databases, while the metrics endpoint offers various metrics related to database connections and connection pools. We are not trying to repeat the observability aspect here.

## Conclusion

This chapter provided a comprehensive overview of key data access features in the Spring Framework, Spring Boot, and Spring Data projects. You now have practical knowledge of using Spring JDBC and Spring JPA for effective data management. Additionally, you explored how to integrate MongoDB into your applications and learned how to write test cases to ensure the reliability of your data access layer.

In the next chapter, we will learn how to properly write RESTful services and secure them.

## Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



*OceanofPDF.com*

# CHAPTER 6

# Building RESTful Spring Services

## Introduction

This chapter begins by introducing RESTful design concepts and then moves on to implement a RESTful service for the sample use case. Following that, we will explore how to use the OpenAPI specification to describe RESTful services. We will then explore the use of REST Template and RestClient for consuming a RESTful service. Finally, we will discuss how to secure RESTful services.

## Structure

The chapter covers the following topics:

- Introduction to REST
- Spring RESTful services
- Error handling
- REST service testing
- REST service documentation
- RESTful Clients
- Securing RESTful services

## Objectives

By the end of this chapter, you will understand the basics of the REST architecture style, and you should be able to develop a RESTful service using Spring Boot. You will also learn about documenting RESTful services and testing them using tools like Postman. Additionally, we will cover how to consume Restful services and basic security aspects.

## Introduction to REST

More than three decades ago, **Hypertext Transfer Protocol (HTTP)** was originally designed to facilitate the transfer of hypertext documents, typically written in HTML, across the **World Wide Web (WWW)**. HTTP defines a set of building blocks to facilitate hypertext exchange, usually between a web server and a web browser functioning as client software. The web browser sends a request to a web server's network location, known as an HTTP endpoint, using one of the delivery semantics called HTTP methods, such as GET or POST. The HTTP endpoint is formatted as a **Uniform Resource Locator (URL)**. When the web server responds to the web browser, it uses HTTP status codes to indicate the type of response, whether the request was successful, or if the document is unavailable. The actual hypertext document is then placed inside the HTTP message body. Additionally, HTTP headers are used to communicate metadata and **quality-of-service (QoS)** information between the server and the client, such as indicating whether authentication information is required with the request and, if so, facilitating the client in providing that authentication.

It is important to note that these HTTP building blocks are not limited to the exchange of hypertext documents; they can also be used for data exchange beyond hypertext. **Representational State Transfer (REST)** is an architectural style that utilizes the same HTTP building blocks to transfer business data across systems in a manner similar to how hypertext is transferred between web servers and web browsers. In REST, you have the flexibility to use various data formats, such as JSON, XML, or even plain text.

The following section will discuss how these HTTP building blocks are used in REST and provide examples using our BookClub use case:

- **Resource identification:** The hierarchical structure of HTTP URLs is used to uniquely identify each resource, as well as a collection of

resources of the same type. Parts of this hierarchical structure can be mapped as path parameters on the server to facilitate the passing of dynamic data. Additionally, query parameters can be used with HTTP URLs to provide optional or control data, such as for pagination purposes. The following table provides a few examples of how URL mapping can be implemented:

URL format	Description
/books	Identify a collection of books
/books/231	Identify a particular book with id = 231
/books/231/copies	Identify a collection of copied of book - 231
books/231/copies/3	Identify a copy with id =3 of book - 231
/books?sortby=year	Identify a collection of books sorted by publication year

**Table 6.1 : URL mapping with the resources of the sample use case**

- **HTTP methods:** HTTP methods can be used to alter the state or make modifications to resources identified by the URL. The following example illustrates how CRUD operations can be mapped to a book resource using HTTP methods:

HTTP method	Description
GET /books	List all the books
GET /books/231	View the details of the particular book with id = 231
POST /books	Create a new book
PUT /books/231	Modify the book- 231
Delete /books/231	Remove the book- 231

**Table 6.2 : HTTP methods with the resources of the sample use case**

- **HTTP status code:** HTTP status codes can be used to indicate the outcome of a requested operation or state change. For instance, making a GET request to /books/231 could result in one of several status codes as shown in the following table.

Status code	Description

<b>200 – OK</b>	The particular book found and included in the body
<b>200 – Created</b>	A new book is created
<b>404 – Not Found</b>	The particular book is not found
<b>401 - Unauthorized</b>	Authentication is required to access the book

**Table 6.3 : HTTP status codes mapping with the responses**

- **HTTP headers:** HTTP headers can be used to convey additional metadata about the business data or provide processing instructions. For example, the Content-Type header can specify the type of content being returned, and the Authorization header can be used to send authentication details.

## Spring RESTful services

The Spring MVC framework, which we previously used to develop web applications with user interfaces, can also be utilized to develop RESTful services. In this section, we will create a RESTful service for the BookClub project that we have been developing throughout this book. To make it closer to real-world services, we will use the JPA repository implementation that we developed in the last chapter. You can download the initial project, which includes a working JPA repository and service implementation, from the companion GitHub repository of this book.

As you might have guessed, our first step is to add `spring-boot-starter-web` as a dependency to your project. This Spring Boot starter ensures that all necessary dependencies, such as Spring MVC and Jackson, which is used for JSON serialization and deserialization, are included, and it configures an embedded Tomcat server to run the RESTful service as a standalone application.

1. `<dependency>`
2. `<groupId>org.springframework.boot</groupId>`
3. `<artifactId>spring-boot-starter-web</artifactId>`
4. `</dependency>`

Next, create a new controller class called **BookController** with the following code:

1. `package com.bpb.hssb.ch6.bookclub.controller;`
- 2.

```
3. import org.springframework.web.bind.annotation.RequestMapping;
4. import org.springframework.web.bind.annotation.RestController;
5.
6. @RestController
7. @RequestMapping("/api/books")
8. public class BookController {
9.
10.    private BookService bookService;
11.
12.    public BookController(BookService bookService) {
13.
14.        this.bookService = bookService;
15.    }
16.
17. }
```

In this code, the **@RestController** annotation, a specialized version of the **@Controller** annotation, simplifies the creation of RESTful web services and indicates that this controller is designed to handle RESTful service requests. The **@RestController** annotation combines **@Controller** and **@ResponseBody**, eliminating the need to define **@ResponseBody** on each method. Additionally, this annotation ensures automatic data binding based on content negotiation. The **@RequestMapping** annotation is used to map web requests to a specific class or method. In this case, we use it to map any request URLs that match the **/api/books** base path to the **BookController**, regardless of the HTTP method and the remaining parts of the request URL. This helps keep the controller class simple by eliminating duplicate code and reducing the chance of errors. Additionally, we have used constructor injection to inject an instance of **BookService** into the **BookController**.

Now, we can introduce a couple of methods to the above class to retrieve a specific book and a collection of books.

```
1.  @GetMapping("/{bookId}")
2.  public Book getBook(@PathVariable int bookId)
   throws BookServiceException {
3.      return bookService.getBook(bookId);
4.  }
5.
6.  @GetMapping
7.  public Iterable<Book> getAllBooks() throws BookServiceException {
```

```
8.     return bookService.getAllBooks();
9. }
```

In the code above, we used the **@GetMapping** annotation to map HTTP GET requests with specific URL values or patterns to the **getBook** and **getAllBooks** methods. The **{bookId}** notation represents a placeholder for the actual value, which is passed to the method using the **@PathVariable** annotation. The **@PathVariable** annotation is used to extract values from the URI path and bind them to method parameters. Similarly, the **@RequestParam** annotation is used to bind query parameters to method parameters in controller methods. For return types, we simply return Java objects, allowing Spring to handle JSON serialization automatically. If needed, you can modify the default serialization logic, such as changing automatically generated field names, by annotating the domain class. For example, as shown in the following code, we used the **@JsonIgnore** annotation in our **Book** class to exclude the **copies** field from serialization:

```
1. @JsonIgnore
2. @OneToMany(mappedBy = "book", cascade = CascadeType.ALL, orphanRemoval = true)
3. private List<BookCopy> copies;
```

Similarly, you can also use the HTTP POST method to create new books, as shown in the following code:

```
1. @PostMapping
2. @ResponseStatus(HttpStatus.CREATED)
3. public Book createBook(@RequestBody Book book)
   throws BookServiceException {
4.     return bookService.createBook(book);
5. }
```

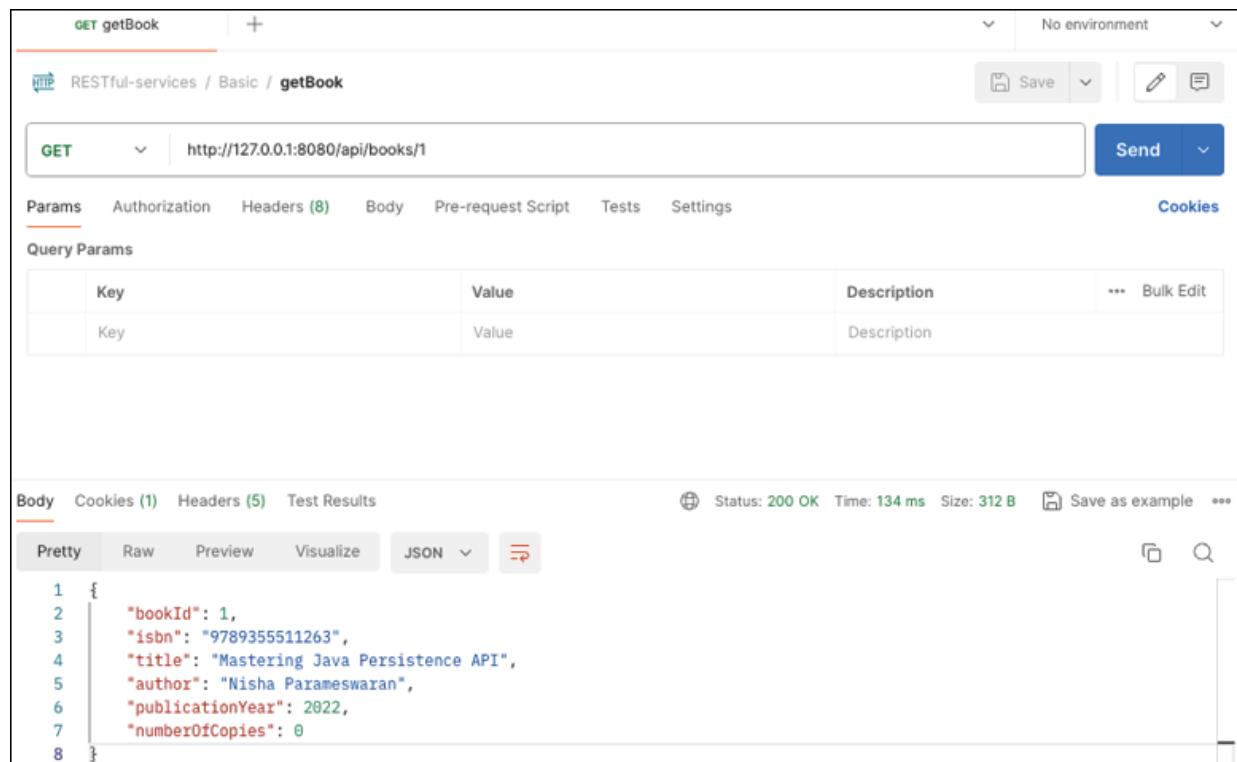
In the code above, we used the **@PostMapping** annotation to map HTTP POST requests with specific URL values or patterns to the **createBook** method. The **@RequestBody** annotation is used to pass the body of the request as a Java object to the method. In this case, Spring automatically handles the deserialization of JSON request data into a Java object using the information provided in the domain classes. Additionally, we used the **@ResponseStatus** annotation to instruct Spring to return an **HTTP 201 Created** status code instead of the default **HTTP 200 OK** status code.

We can also implement the capability to remove a book using the HTTP DELETE method, as shown in the following code. In this code, we used the **@DeleteMapping** annotation to map HTTP DELETE requests with specific

URL values or patterns to the **removeBook** method, and we used the **@RequestBody** annotation to pass the **bookId** to the method:

```
1. @DeleteMapping("/{bookId}")
2. public void removeBook(@PathVariable int bookId) throws BookServiceException {
3.     bookService.removeBook(bookId);
4. }
```

Once you have completed the above steps, you are ready to run the RESTful service by executing the **BookclubApplication** class. After running the application, you can use any HTTP client, such as Postman or cURL, to test the service. The following figure shows how you can access the details of a book using Postman:



The screenshot shows a Postman collection named 'getBook' with a single request. The request is a GET to `http://127.0.0.1:8080/api/books/1`. The 'Params' tab is selected, showing a single query parameter 'Key' with a value 'Value'. The 'Body' tab shows the JSON response:

```
1 {
2     "bookId": 1,
3     "isbn": "9789355511263",
4     "title": "Mastering Java Persistence API",
5     "author": "Nisha Parameswaran",
6     "publicationYear": 2022,
7     "numberOfCopies": 0
8 }
```

*Figure 6.1: Get details of a book*

The following figure demonstrates how you can create a new book:

POST createBook

HTTP RESTful-services / Basic / createBook

POST http://127.0.0.1:8080/api/books

Params Authorization Headers (10) Body **raw** Pre-request Script Tests Settings Cookies

Body (1) Headers (5) Test Results

Status: 201 Created Time: 117 ms Size: 315 B Save as example

```

1 {
2   "isbn": "9789355511254",
3   "title": "Hands on Spring Spring Boot 3",
4   "author": "Sagara Gunathunga",
5   "publicationYear": 2024
6 }

```

```

1 {
2   "bookId": 6,
3   "isbn": "9789355511254",
4   "title": "Hands on Spring Spring Boot 3",
5   "author": "Sagara Gunathunga",
6   "publicationYear": 2024,
7   "numberOfCopies": 0
8 }

```

Figure 6.2: Creating a new book

## Error handling

Before we discuss error handling, let us try querying a non-existing book by passing a non-existing book ID through the GET request. You will get a result similar to the one shown in [Figure 6.3](#). This is not ideal, as it complicates processing on the client side and, more importantly, exposes internal details of your service, which can have security implications. Ideally, we should provide a well-crafted error message instead of revealing internal error details. In the following section, we will cover how to handle these errors properly.

The screenshot shows the Postman application interface. At the top, a header bar indicates a GET request to 'getBook-Error-1'. Below this, the URL 'http://127.0.0.1:8080/api/books/100' is entered. The 'Params' tab is selected, showing a single 'Key' entry. The 'Body' tab is selected, displaying a JSON response with the following content:

```

1  {
2      "timestamp": "2024-08-26T04:17:03.956+00:00",
3      "status": 500,
4      "error": "Internal Server Error",
5      "trace": "java.util.NoSuchElementException: No value present\n\tat java.base/java.util.Optional.get(Optional.java:143)\n\tat com.bpb.hssb.ch6.bookclub.service.BookServiceImpl.getBook(BookServiceImpl.java:25)\n\tat com.bpb.hssb.ch6.bookclub.controller.BookController.getBook(BookController.java:31)\n\tat java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)\n\tat java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:77)\n\tat java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)\n\tat java.base/java.lang.reflect.Method.invoke(Method.java:568)\n\tat org.springframework.web.method.support.InvocableHandlerMethod.doInvoke(InvocableHandlerMethod.java:255)\n\tat org.springframework.web.method.support.InvocableHandlerMethod.invokeForRequest(InvocableHandlerMethod.java:188)\n\tat org.springframework.web.servlet.mvc.method.annotation.ServletInvocableHandlerMethod.invokeAndHandle(ServletInvocableHandlerMethod.java:118)\n\tat org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter.invokeHandlerMethod(RequestMappingHandlerAdapter.java:877)\n\tat org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter.handleInternal(RequestMappingHandlerAdapter.java:800)\n\tat org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter.handleInternal(RequestMappingHandlerAdapter.java:760)\n\tat org.springframework.web.servlet.DispatcherServlet.doDispatch(DispatcherServlet.java:1072)\n\tat org.springframework.web.servlet.DispatcherServlet.doService(DispatcherServlet.java:962)\n\tat org.springframework.web.servlet.FrameworkServlet.processEvent(FrameworkServlet.java:1010)\n\tat org.springframework.web.servlet.FrameworkServlet.doGet(FrameworkServlet.java:902)\n\tat javax.servlet.http.HttpServlet.service(HttpServlet.java:682)\n\tat org.springframework.web.servlet.FrameworkServlet.service(FrameworkServlet.java:884)\n\tat javax.servlet.http.HttpServlet.service(HttpServlet.java:761)\n\tat org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:227)\n\tat org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:162)\n\tat org.apache.tomcat.websocket.server.WsFilter.doFilter(WsFilter.java:52)\n\tat org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:187)\n\tat org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:162)\n\tat org.springframework.web.filter.CharacterEncodingFilter.doFilterInternal(CharacterEncodingFilter.java:126)\n\tat org.springframework.web.filter.OncePerRequestFilter.doFilter(OncePerRequestFilter.java:110)\n\tat org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:187)\n\tat org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:162)\n\tat org.apache.catalina.core.StandardWrapperValve.invoke(StandardWrapperValve.java:197)\n\tat org.apache.catalina.core.StandardContextValve.invoke(StandardContextValve.java:97)\n\tat org.apache.catalina.authenticator.AuthenticatorBase.invoke(AuthenticatorBase.java:540)\n\tat org.apache.catalina.core.StandardHostValve.invoke(StandardHostValve.java:135)\n\tat org.apache.catalina.valves.ErrorReportValve.invoke(ErrorReportValve.java:92)\n\tat org.apache.catalina.core.StandardEngineValve.invoke(StandardEngineValve.java:78)\n\tat org.apache.catalina.connector.CoyoteAdapter.service(CoyoteAdapter.java:357)\n\tat org.apache.coyote.http11.Http11Processor.service(Http11Processor.java:375)\n\tat org.apache.coyote.AbstractProcessorLight.process(AbstractProcessorLight.java:65)\n\tat org.apache.coyote.AbstractProtocol$ConnectionHandler.process(AbstractProtocol.java:814)\n\tat org.apache.tomcat.util.net.NioEndpoint$SocketProcessor.doRun(NioEndpoint.java:1690)\n\tat org.apache.tomcat.util.net.SocketProcessorBase.run(SocketProcessorBase.java:49)\n\tat java.base/java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1144)\n\tat java.base/java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:628)\n\tat org.apache.tomcat.util.threads.TaskThread$WrappingRunnable.run(TaskThread.java:61)\n\tat java.base/java.lang.Thread.run(Thread.java:834)"

```

*Figure 6.3: Unmanaged errors*

There are several approaches to handling errors in REST services, including handling errors locally within each method or globally across the application. As we did with web applications in *Chapter 4, Building Spring MVC Web Applications*, we will use global error handling because it offers several advantages, such as cleaner controller methods and greater reuse of error-handling code. First, let us create a **GlobalExceptionHandler** with the following minimal code:

```

1. package com.bpb.hssb.ch6.bookclub.config;
2.
3. import org.springframework.web.bind.annotation.ControllerAdvice;
4.
5. @ControllerAdvice
6. public class GlobalExceptionHandler {
7.
8.
9. }

```

10.

In this code, we use **@ControllerAdvice** to indicate that this class handles exceptions globally across the entire application. Now, we can introduce handling methods for each exception. For example, the following code handles **NoSuchElementException**:

```
1. @ExceptionHandler(NoSuchElementException.class)
2. public ResponseEntity<Object>
  handleBookNotFoundException(NoSuchElementException ex) {
3.
4.   Map<String, Object> body = new HashMap<>();
5.   body.put("message", "Book not found");
6.   body.put("timestamp", LocalDateTime.now());
7.   return new ResponseEntity<>(body, HttpStatus.NOT_FOUND);
8. }
```

Within the method `body`, we create an HTTP response object with the necessary error data, such as an error description and timestamp, and return it with an **HTTP 404** status code.

Here is a generic method to handle errors that we have not specifically addressed, using the **HTTP 500** status code:

```
1. @ExceptionHandler(Exception.class)
2. protected ResponseEntity<Void>
  handleException(Exception ex, WebRequest request) {
3.   return ResponseEntity.internalServerError().build();
4. }
```

If you rerun the test after adding all the necessary error handling methods, you should see the following error message in Postman:

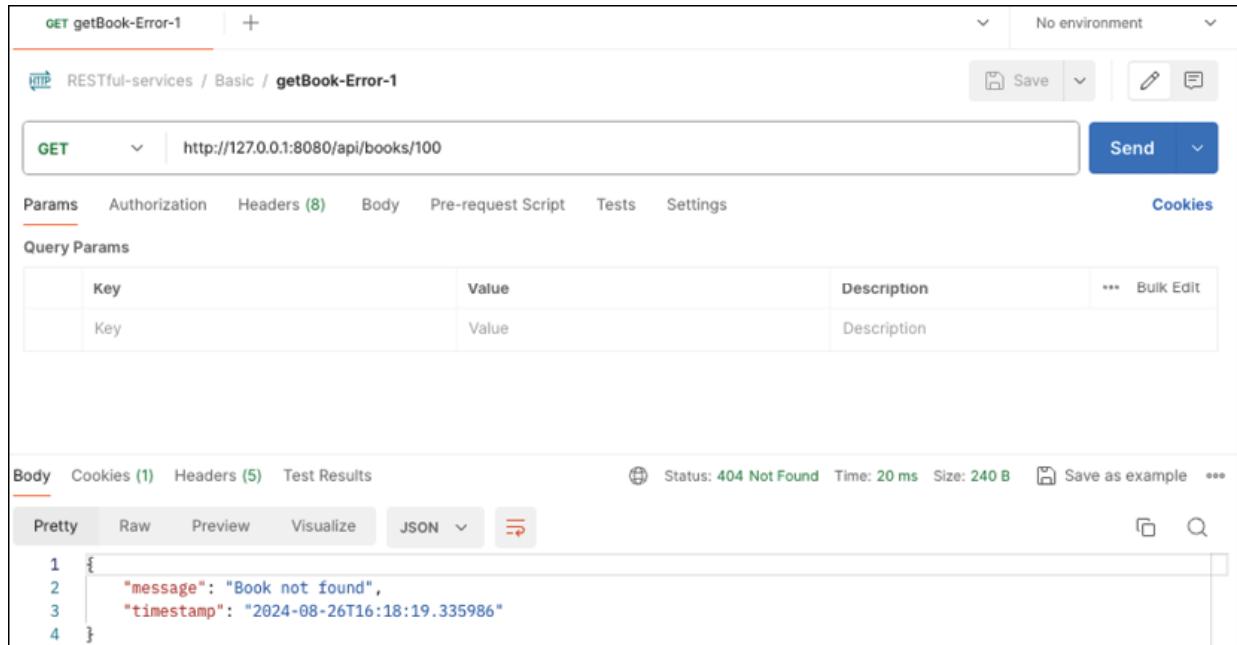


Figure 6.4: A managed error

## REST service testing

When it comes to writing test cases for Spring RESTful services, there are a few options available. You can write tests using an embedded web server, connect to a separate web server like **TestContainers**, or create unit tests that isolate the REST controller using **MockMVC**. We have already discussed some of these options in *Chapter 3, Spring Essentials for Enterprise Applications*. In this section, let us focus on writing a unit test for **BookController**. First, create a **BookControllerTest** class using the following code:

```

1. package com.bpb.hssb.ch6.bookclub.controller;
2.
3. import org.springframework.beans.factory.annotation.Autowired;
4. import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
5. import org.springframework.boot.test.mock.mockito.MockBean;
6. import org.springframework.test.web.servlet.MockMvc;
7.
8. import com.bpb.hssb.ch6.bookclub.service.BookService;
9.
10. @WebMvcTest(BookController.class)
11. public class BookControllerTest {
12.

```

```
13. @Autowired
14. private MockMvc mockMvc;
15.
16. @MockBean
17. private BookService bookService;
18.
19. }
```

In this code, adding the **@WebMvcTest** annotation ensures that only the web layer of the application is loaded rather than the full application context. Additionally, we specify that we intend to test **BookController**. The **@MockBean** annotation creates a Mockito mock for the **BookService** interface to be used in the test cases. Finally, we inject a **MockMvc** instance, which will be used to simulate HTTP requests to the controller.

Now, we can add our test cases to the class. Let us start by adding the **testGetBook** test case, as follows:

```
1. @Test
2. void testGetBook() throws Exception {
3.
4.     Book book = Book.builder().bookId(1).author("Nisha").title("Java 17")
5.         .isbn("9789355511263").publicationYear(2022).build();
6.     when(bookService.getBook(1)).thenReturn(book);
7.
8.     this.mockMvc.perform(get("/api/books/1"))
9.         .andDo(print())
10.        .andExpect(status().isOk())
11.        .andExpect(content().contentType(MediaType.APPLICATION_JSON))
12.        .andExpect(jsonPath("$.bookId").value(book.getBookId()))
13.        .andExpect(jsonPath("$.author").value(book.getAuthor()))
14.        .andExpect(jsonPath("$.title").value(book.getTitle()))
15.        .andExpect(jsonPath("$.isbn").value(book.getIsbn()))
16.        .andExpect(jsonPath("$.publicationYear").value(book.getPublicationYear()));
17. }
```

In the code above, we first configure the mock behavior for the **bookService** interface and then use **MockMVC** to make an HTTP GET request. After that, we verify the results using **JsonPath**.

Here is the code for the **testCreateBook** case. In this code, we post a JSON request to create a book and verify the result again using **JsonPath**:

```
1. @Test
```

```

2. void testCreateBook() throws Exception {
3.
4.     Book book = Book.builder().bookId(1).author("Nisha").title("Java 17")
5.         .isbn("9789355511263").publicationYear(2022).build();
6.     when(bookService.createBook(any(Book.class))).thenReturn(book);
7.
8.     String newBookJson = "{"
9.         + "\"author\": \"Nisha\","
10.        + "\"title\": \"Java 17\","
11.        + "\"isbn\": \"9789355511263\","
12.        + "\"publicationYear\": 2022"
13.        + "}";
14.
15.     this.mockMvc.perform(post("/api/books")
16.         .contentType(MediaType.APPLICATION_JSON)
17.         .content(newBookJson))
18.         .andDo(print())
19.         .andExpect(status().isCreated())
20.         .andExpect(content().contentType(MediaType.APPLICATION_JSON))
21.         .andExpect(jsonPath("$.author").value("Nisha"))
22.         .andExpect(jsonPath("$.title").value("Java 17"))
23.         .andExpect(jsonPath("$.isbn").value("9789355511263"))
24.         .andExpect(jsonPath("$.publicationYear").value(2022));
25. }

```

We can write test cases not only for successful scenarios but also for error messages. For instance, the following code verifies an HTTP GET request with a non-existing **bookId**:

```

1. @Test
2. void testGetBookNotFound() throws Exception {
3.
4.     when(bookService.getBook(1000)).thenThrow(new NoSuchElementException());
5.
6.     this.mockMvc.perform(get("/api/books/1000"))
7.         .andDo(print())
8.         .andExpect(status().isNotFound())
9.         .andExpect(content().contentType(MediaType.APPLICATION_JSON))
10.        .andExpect(jsonPath("$.message").value("Book not found"))
11.        .andExpect(jsonPath("$.timestamp").exists());

```

```
12. }
```

## REST service documentation

In this section, we will explore how to add documentation to the REST service we developed using the OpenAPI specification, originally known as **Swagger**. OpenAPI has become the de-facto standard for documenting REST services. We will use a Java library called **springdoc-openapi** to generate the documentation for our REST service.

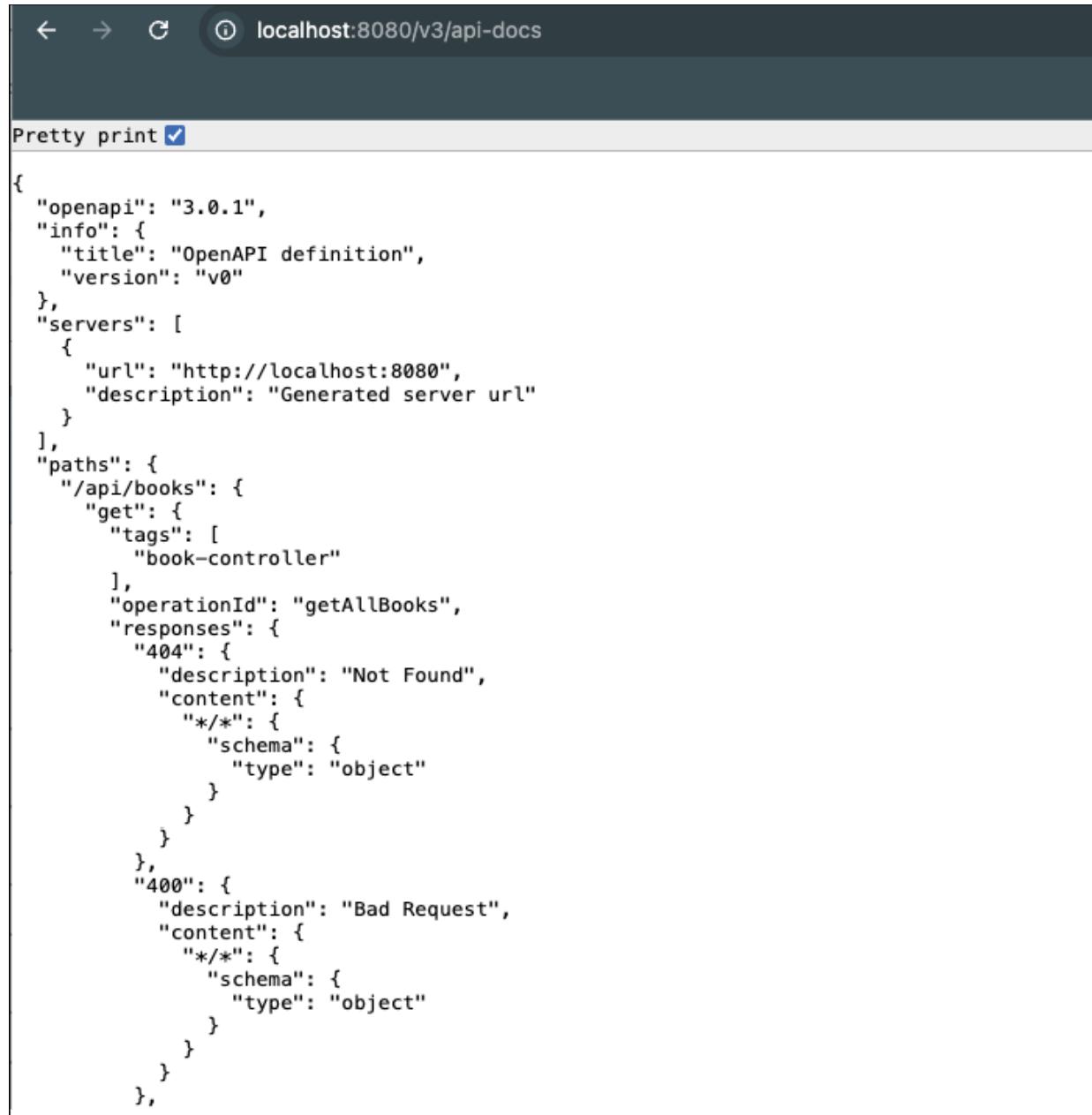
First, add the following Maven dependency to include **springdoc-openapi** in your project:

1. <dependency>
2.    <groupId>org.springdoc</groupId>
3.    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
4.    <version>2.6.0</version>
5. </dependency>

To generate basic documentation, you do not need to do anything extra, **springdoc-openapi** handles everything. Once the application is running, you can visit the following URL in your web browser to access the autogenerated JSON-formatted OpenAPI specification:

<http://localhost:8080/v3/api-docs>

You should see something similar to the following figure:



```
Pretty print   
{  
  "openapi": "3.0.1",  
  "info": {  
    "title": "OpenAPI definition",  
    "version": "v0"  
  },  
  "servers": [  
    {  
      "url": "http://localhost:8080",  
      "description": "Generated server url"  
    }  
  ],  
  "paths": {  
    "/api/books": {  
      "get": {  
        "tags": [  
          "book-controller"  
        ],  
        "operationId": "getAllBooks",  
        "responses": {  
          "404": {  
            "description": "Not Found",  
            "content": {  
              "/*/*": {  
                "schema": {  
                  "type": "object"  
                }  
              }  
            }  
          },  
          "400": {  
            "description": "Bad Request",  
            "content": {  
              "/*/*": {  
                "schema": {  
                  "type": "object"  
                }  
              }  
            }  
          },  
        }  
      }  
    }  
  }  
},
```

*Figure 6.5: JSON-formatted OpenAPI specification*

You can also access the YAML-formatted OpenAPI specification at the following URL:

<http://localhost:8080/v3/api-docs.yaml>

Finally, visit the interactive Swagger UI interface at:

<http://localhost:8080/swagger-ui/index.html>

In addition to being a human-readable documentation, Swagger UI serves as a testing tool where you can provide necessary inputs and invoke resources.

The following figure shows how BookClub API is rendered in Swagger UI.

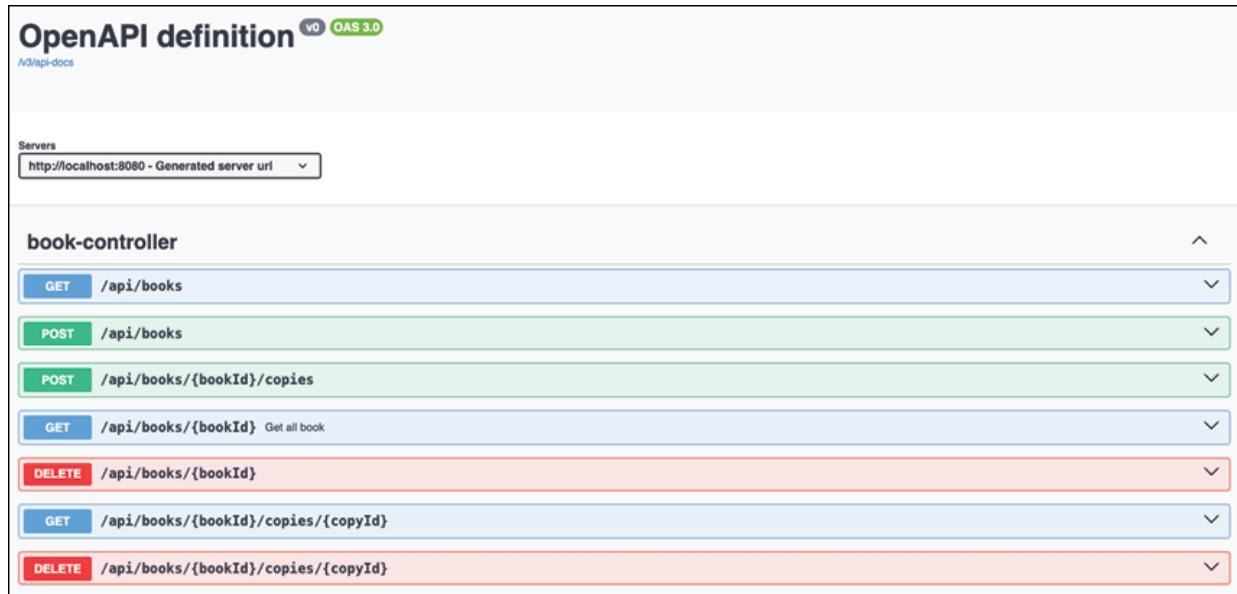


Figure 6.6: Swagger UI interface

You can enhance the automatic document generation by annotating the controller methods using the annotations provided by **springdoc-openapi**. For example, the annotated **getBook** method is shown as follows:

1. `@Operation(summary = "Get all book", description = "List all the available books with some basic details")`
2. `@ApiResponse(responseCode = "200", description = "Successful", content = @Content(mediaType = "application/json", schema = @Schema(implementation = Book.class)))`
3. `@GetMapping("/{bookId}")`
4. `public Book getBook(@PathVariable int bookId)`
5. `throws BookServiceException {`
6. `return bookService.getBook(bookId);`
7.

## RESTful Clients

So far, we have explored how to develop RESTful services, which are invaluable for exposing your business capabilities as APIs and simplifying integration with both your own and third-party applications. However, sometimes, you need to integrate your applications with REST APIs exposed by third parties. Spring offers a set of capabilities to assist with such

scenarios as well. In this section, we will look at a few options that can help in developing REST Clients.

First, visit [Spring Initializr](#) and create a project with the following details:

**Project : Maven**      **Language : Java**

## Spring Boot : Latest stable version ( e.g. – 3.3.0)

## Project Metadata :

## Group : com.bpb.hssb.ch6

## Artifact : rest-clients

## Name : rest-clients

## Packaging : Jar (Default)

## Java : 17 (Default)

## Dependencies: `spring-boot-starter-web`, `Lombok`

Download and open this project using VS Code IDE. Since we do not need a web server to run the client code, you can disable the automatic startup of the web server by adding the following property to the **application.properties** file. Otherwise, **spring-boot-starter-web** will automatically configure an embedded web server for us.

## 1. `spring.main.web-application-type=none`

Next, let us create a client-side domain class named **Book** using the following code:

```
1. package com.bpb.hssb.ch6.bookclub.client;  
2.  
3. import com.fasterxml.jackson.annotation.JsonIgnoreProperties;  
4.  
5. import lombok.AllArgsConstructor;  
6. import lombok.Builder;  
7. import lombok.Getter;  
8. import lombok.NoArgsConstructor;  
9. import lombok.Setter;  
10. import lombok.ToString;  
11.  
12. @Getter  
13. @Setter  
14. @AllArgsConstructor  
15. @NoArgsConstructor  
16. @Builder
```

```
17. @ToString
18. @JsonIgnoreProperties(ignoreUnknown = true)
19. public class Book {
20.
21.     private int bookId;
22.     private String isbn;
23.     private String title;
24.     private String author;
25.     private int publicationYear;
26. }
```

In the code, we have used a set of Lombok annotations that are already familiar to you by now. The only addition here is the **@JsonIgnoreProperties** annotation, which prevents exceptions from being thrown when the program encounters unknown properties.

Next, create a Java class called **BookClubRestTemplateClient** and add the following code:

```
1. package com.bpb.hssb.ch6.bookclub.client;
2.
3. import org.springframework.beans.factory.annotation.Autowired;
4. import org.springframework.beans.factory.annotation.Value;
5. import org.springframework.boot.CommandLineRunner;
6. import org.springframework.boot.SpringApplication;
7. import org.springframework.boot.autoconfigure.SpringBootApplication;
8. import org.slf4j.Logger;
9. import org.slf4j.LoggerFactory;
10. import org.springframework.web.client.RestTemplate;
11.
12.
13. @SpringBootApplication
14. public class BookClubRestTemplateClient
15.     implements CommandLineRunner {
16.
17.     private static final Logger logger =
18.         LoggerFactory.getLogger(BookClubRestClient.class);
19.
20.     @Autowired
21.     private RestTemplate restTemplate;
22.
23.     @Value("${bookclub.api.baseurl}")
24.     private String baseurl;
```

```
23.
24. public static void main(String[] args) {
25.     SpringApplication.run(BookClubRestClient.class, args);
26. }
27.
28. @Override
29. public void run(String... args) throws Exception {
30.
31. }
32.
33. }
```

In the above code, we have injected **RestTemplate** into our application. Just like any other template available in Spring, **RestTemplate** simplifies the process of making HTTP requests and handling responses. It reduces the boilerplate code typically associated with making HTTP calls, making it easier to interact with RESTful web services without dealing with low-level HTTP details. **RestTemplate** defines a set of methods that reflect HTTP methods such as **GET**, **POST**, **PUT**, and **DELETE**, along with an exchange method offering more flexible features. Additionally, we used the **@Value** annotation to read the **baseurl** property from the **application.properties** file.

Next, we can define a method to retrieve the details of a book using **RestTemplate**, as follows:

```
1. private void printBookInfo(int bookId) {
2.     Book book = restTemplate.getForObject(baseurl + "/books/{bookId}", Book.class, bookId);
3.     logger.info("Book details : " + book);
4. }
```

In the code provided above, we used **getForObject** by specifying the HTTP endpoint URL, the response type, and **bookId** as a parameter. Once the results are available, **RestTemplate** ensures that the JSON response is deserialized into a Java bean.

Next, let us add the **createBook** method:

```
1. public Book createBook() {
2.
3.     String bookStoreUrl = baseurl + "/books";
4.     Book book = Book.builder()
5.         .title("Hands on Spring and Spring Boot 3")
```

```

6. .author("Sagara Gunathunga")
7. .isbn("9789355511232")
8. .publicationYear(2024)
9. .build();
10.
11. HttpHeaders headers = new HttpHeaders();
12. headers.setContentType(MediaType.APPLICATION_JSON);
13. HttpEntity<Book> request = new HttpEntity<>(book, headers);
14. ResponseEntity<Book> response = restTemplate.postForEntity(bookStoreUrl, request, Book.class);
15.
16. if (response.getStatusCode() == HttpStatus.CREATED) {
17.     return response.getBody();
18. } else {
19.     throw new RuntimeException("Failed to create book.
        Status code: " + response.getStatusCode());
20. }
21. }

```

In this code, we used the **postForEntity** method of **RestTemplate**, along with request data in the form of a Java bean and a **ContentType** header to indicate the type of the content.

After adding these two methods, you can call them within the run method of our client program, as follows:

```

1. @Override
2. public void run(String... args) throws Exception {
3.     Book book = createBook();
4.     printBookInfo(book.getBookId());
5. }

```

The final step is to define the Spring configuration by setting up **RestTemplate**, as follows:

```

1. package com.bpb.hssb.ch6.bookclub.client;
2.
3. import org.springframework.boot.web.client.RestTemplateBuilder;
4. import org.springframework.context.annotation.Bean;
5. import org.springframework.context.annotation.Configuration;
6. import org.springframework.web.client.RestTemplate;
7.
8. @Configuration
9. public class BookclubConfiguration {

```

```
10.
11. @Bean
12. public RestTemplate restTemplate(RestTemplateBuilder builder) {
13.     return builder.build();
14. }
15.
16. }
```

Also, make sure to update the **application.properties** file with the **baseurl** property:

```
1. bookclub.api.baseurl=http://127.0.0.1:8080/api
```

Now, you can run our client-side application, and if everything is correct, you should be able to see the details of the newly added book in the console. Since this is a client-side application, you will need a service to call. You can run the RESTful service that we developed in the previous section.

**RestTemplate** is quite popular among developers; however, it is a synchronous client for performing HTTP requests, which can be seen as a drawback by modern standards. In fact, the Spring team themselves recommend using **RestClient**, introduced with the Spring 6 release, as an alternative. **RestClient** is simpler and supports both synchronous and asynchronous communications. Before moving on to the next section, let us rewrite the above client application using **RestClient**.

First, we can update the Spring configuration by introducing a **RestClient** bean, as shown in the following code segment:

```
1. @Value("${bookclub.api.baseurl}")
2. private String baseurl;
3.
4. @Bean
5. public RestClient restClient() {
6.     return RestClient.builder()
7.         .baseUrl(baseurl)
8.         .build();
9. }
```

Next, we can inject the **RestClient** into our using main program the **@Autowired** annotation as follows:

```
1. @Autowired
2. private RestClient restClient;
```

After that, update the **printBookInfo** method according to the following

code:

```
1. private void printBookInfo(int bookId) {  
2.  
3.     Book book = restClient.get()  
4.     .uri("/books/{bookId}", bookId)  
5.     .retrieve()  
6.     .body(Book.class);  
7.  
8.     logger.info("Book details : {}", book);  
9. }
```

We can also update the **createBook** method using the following code segment:

```
1. public Book createBook() {  
2.  
3.     Book book = Book.builder()  
4.     .title("Hands on Spring and Spring Boot 3")  
5.     .author("Sagara Gunathunga")  
6.     .isbn("9789355511232")  
7.     .publicationYear(2024)  
8.     .build();  
9.  
10.    return restClient.post()  
11.        .uri("/books")  
12.        .contentType(MediaType.APPLICATION_JSON)  
13.        .body(book)  
14.        .retrieve()  
15.        .toEntity(Book.class)  
16.        .getBody();  
17. }
```

As you can see, **RestClient** offers cleaner and more readable code compared to **RestTemplate**. Additionally, Spring provides **WebClient**, which can be used with reactive services as well. To summarize, if you are transitioning from **RestTemplate** and need a simple HTTP client for traditional Spring MVC applications, RestClient is an excellent choice. However, if you have high-concurrency and non-blocking requirements that demand advanced features like streaming and backpressure, **WebClient** is the better option.

## Securing RESTful services

The Spring Security project, which we covered in previous chapters, can also be used to secure RESTful services. You can easily implement **BasicAuth** for simple use cases or opt for OAuth2-based JWT token security to align with real-world security requirements. In this section, we will implement **BasicAuth** for our RESTful service.

As we have done several times already, as the first step, add the necessary Spring Security dependencies to your project:

```
1. <dependency>
2.   <groupId>org.springframework.boot</groupId>
3.   <artifactId>spring-boot-starter-security</artifactId>
4. </dependency>
5. <dependency>
6.   <groupId>org.springframework.security</groupId>
7.   <artifactId>spring-security-test</artifactId>
8.   <scope>test</scope>
9. </dependency>
```

Next, add the **SecurityConfiguration** class with the following code to enable **BasicAuth** for our service:

```
1. package com.bpb.hssb.ch6.bookclub.config;
2.
3. import org.springframework.context.annotation.Bean;
4. import org.springframework.context.annotation.Configuration;
5. import org.springframework.security.config.Customizer;
6. import org.springframework.security.config.annotation.
   web.builders.HttpSecurity;
7. import org.springframework.security.web.SecurityFilterChain;
8.
9. @Configuration
10. public class SecurityConfiguration {
11.
12.   @Bean
13.   public SecurityFilterChain filterChain(HttpSecurity http)
      throws Exception {
14.     http
15.       .authorizeHttpRequests((authz) -> authz
16.         .anyRequest().authenticated())
17.       .httpBasic(Customizer.withDefaults());
18.   return http.build();
```

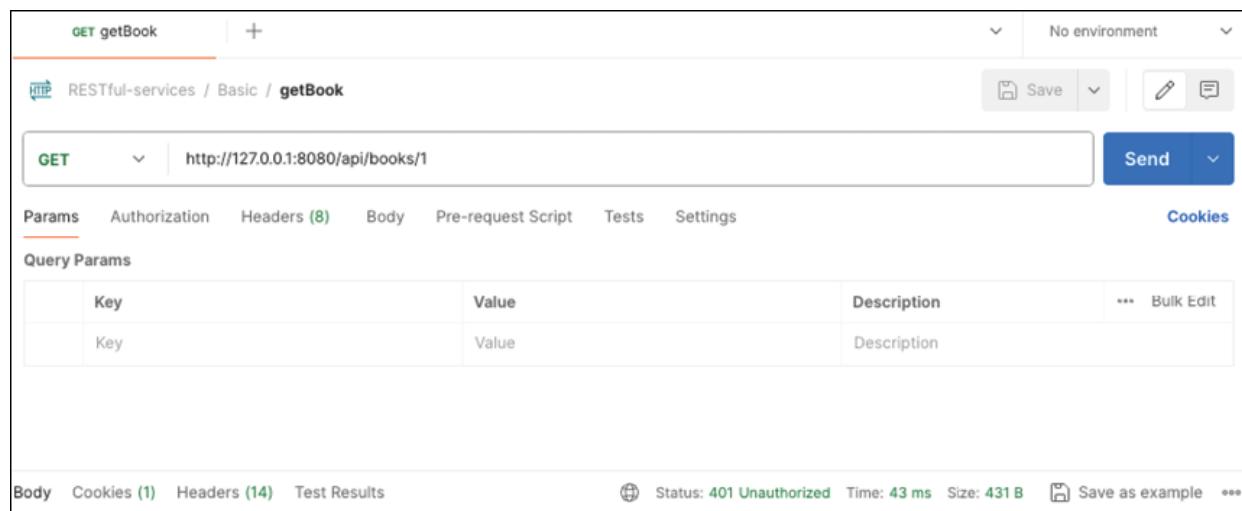
```
19.  }
20.
21. }
```

As seen in previous chapters, we define the **SecurityFilterChain** bean, which configures security settings for HTTP requests. We have instructed it to authenticate any requests using the HTTP **BasicAuth** schema with default settings.

Finally, add a test user by including the following lines in the **application.properties** file:

1. `spring.security.user.name=sam`
2. `spring.security.user.password=abcd`
3. `spring.security.user.roles=ADMIN`

With **BasicAuth** configured for the service, we can proceed to test it using Postman. If you run the application again and send an HTTP GET request to retrieve details of book ID 1, as shown in the following figure, you will now receive an **HTTP 401 Unauthorized** status code:



The screenshot shows the Postman interface with a successful API call. The request URL is `http://127.0.0.1:8080/api/books/1`. The response status is `200 OK`, and the response body is a JSON object:

```
{  "id": 1,  "title": "The Great Gatsby",  "author": "F. Scott Fitzgerald",  "published": "1925-04-10",  "genre": "Fiction",  "rating": 4.5,  "price": 12.99}
```

*Figure 6.7: A request without security*

Additionally, you should see the following HTTP headers in the Postman console, indicating the authentication schema supported by the server, in this case, HTTP BasicAuth:



The screenshot shows the Postman interface with the 'Console' tab selected. The response headers are listed as follows:

```
Vary: "Origin"
Vary: "Access-Control-Request-Method"
Vary: "Access-Control-Request-Headers"
WWW-Authenticate: "Basic realm="Realm"""
X-Content-Type-Options: "nosniff"
X-XSS-Protection: "0"
Cache-Control: "no-cache, no-store, max-age=0, must-revalidate"
Pragma: "no-cache"
Expires: "0"
X-Frame-Options: "DENY"
Content-Length: "0"
Date: "Mon, 26 Aug 2024 10:48:03 GMT"
Keep-Alive: "timeout=60"
Connection: "keep-alive"
```

*Figure 6.8: Postman console with headers*

To pass security credentials, you can use Postman's **Authorization** tab to provide the username and password for the test user we added earlier. After that, you should be able to receive successful results again, as shown in the following figure:

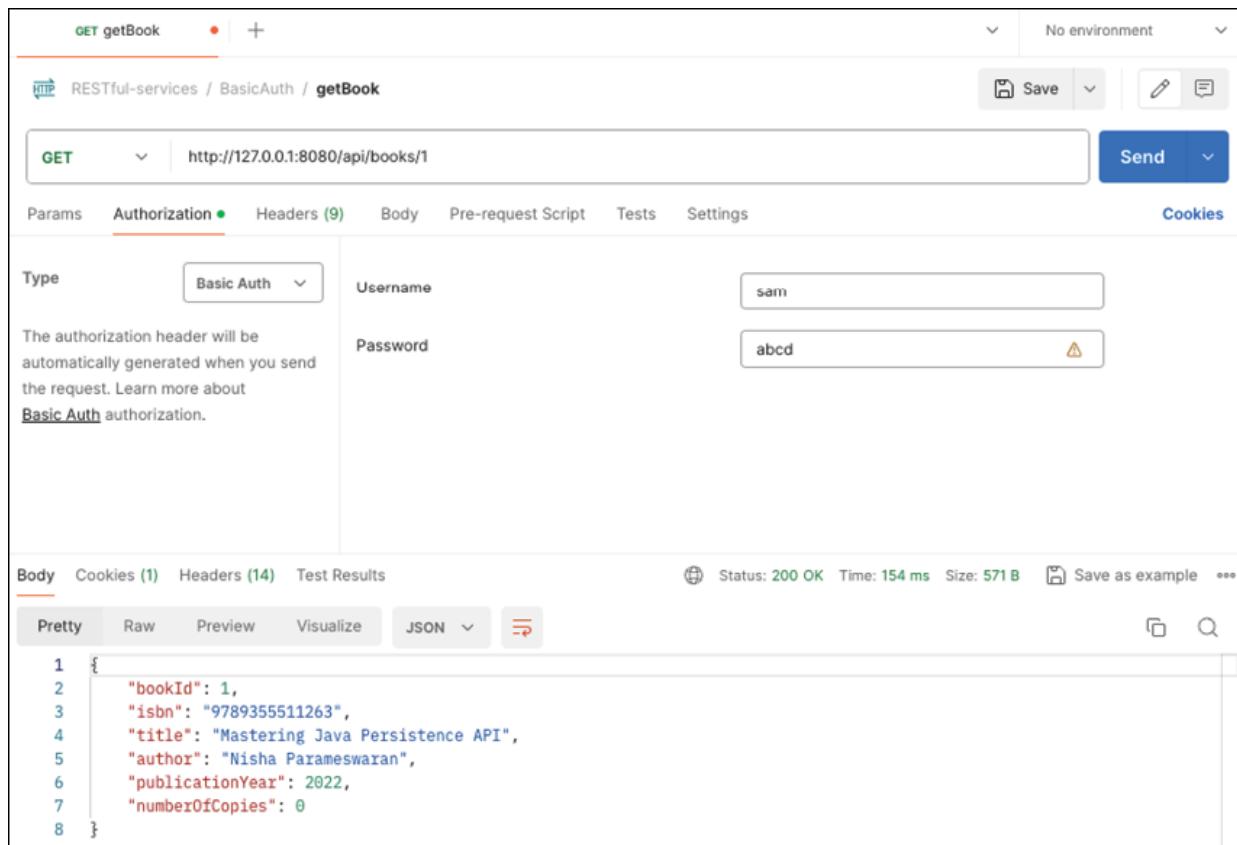


Figure 6.9: A request with security credentials

## Conclusion

This chapter provided a comprehensive introduction to the REST architecture style and its practical implementation using Spring Boot. By now, you should be able to develop RESTful services, document them effectively, and test them using tools like Postman. Additionally, the chapter explored how to consume RESTful services and introduced essential security concepts to safeguard your APIs.

In the next chapter, we will learn the basics of GraphQL and develop a GraphQL service using Spring Boot.

## Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



*OceanofPDF.com*

# CHAPTER 7

# Building GraphQL Spring Services

## Introduction

This chapter begins by exploring the fundamentals of GraphQL and comparing it to REST services. We will then develop our first GraphQL service for the use case, implementing both data read and modification operations. Additionally, we will cover how to write test cases for GraphQL services and utilize testing tools. Finally, we will discuss securing a GraphQL service.

## Structure

The chapter covers the following topics:

- Introduction to GraphQL and GraphQL schema
- Spring GraphQL services
- GraphQL service testing
- GraphQL clients
- Securing GraphQL services

## Objectives

By the end of this chapter, you will understand the fundamentals of GraphQL and how it differs from other technologies. You will also learn how

to develop a GraphQL service and a client using Spring Boot, complete with proper test cases and security measures.

## Introduction to GraphQL

The RESTful services or REST APIs we explored in the previous chapter have gained significant popularity, especially over the last decade, and are now considered mainstream technology. They are widely used across various industry verticals to the point of giving rise to concepts like the API-led economy. However, REST API services do come with certain drawbacks.

When designing a RESTful service, the developer needs to establish a fixed API, which includes a specification of supported operations and a set of defined data elements for inputs, outputs, and errors. This is typically described using the OpenAPI specification. This approach simplifies the job of the developer who consumes the service because REST APIs behave in a predictable manner, and the responses are consistent, so clients do not have to worry about unexpected results.

However, in the real world, as REST APIs evolve to meet changing business requirements and consumer expectations, having a fixed API specification can present challenges and reduce agility. For instance, as more applications begin to consume the REST API, each application might require a different set of data elements. This could lead to the creation of new REST APIs to accommodate varying expectations. Additionally, when business requirements change, it may be necessary to add or remove data elements from the REST API, potentially resulting in the creation of new API versions. While this discussion focuses on REST APIs, since they are the most common type we develop today, these challenges are also applicable to other API types, including SOAP and gRPC. The following points summarize these challenges.

- **Overfetching:** APIs often return more data than the client needs, wasting bandwidth and processing power.
- **Underfetching:** API consumers sometimes need to make multiple API calls to different endpoints to gather all the required data, leading to increased network requests and delays.
- **Inflexibility:** API consumers are constrained by the predefined data

structure of an API, making it difficult to request only the specific data they need.

GraphQL addresses these limitations by providing efficient and flexible query language for APIs, which allows API consumers to request exactly the data they need in a single query. GraphQL services are based on a schema, which requires the definition of all the types and data fields supported by the service, then the queries, and then the operations. GraphQL supports two types of operations called queries and mutations:

- **Queries:** Queries are read-only operations used to fetch data from the service according to the client's expectations. A query can take a hierarchical structure and support nested queries.
- **Mutations:** Mutations are write operations that can create, update, or delete data in the service.

The following points summarize some of the important features of GraphQL:

- **Flexibility and efficiency:** GraphQL allows clients to request exactly the data they need, no more and no less. This solves the over fetching and under fetching issues that we discussed.
- **Data aggregation:** GraphQL simplifies the process of aggregating data from multiple sources or APIs, letting clients retrieve all required data in a single API call.
- **Strongly typed schema:** GraphQL provides a strongly typed schema with a clear contract and data validation.
- **Self-documenting:** The schema serves as documentation for the API.
- **Versioning:** GraphQL makes it easier to evolve APIs over time without breaking existing clients. New fields can be added to the schema without affecting existing queries.

It is also important to remember that while GraphQL offers many great features, they come at a cost. To choose the right API technology, you need to have a clear understanding of the drawbacks associated with GraphQL. Some of the main drawbacks are:

- **Complexity:** GraphQL can introduce additional complexity, especially for simple APIs. The client development is also more complex than the REST clients.

- **Performance concerns:** Without proper optimization, complex queries can lead to performance issues, and in some cases, this may lead to security implications as well. You need to think thoroughly about the query complexity limits and query depth limits in terms of performance and security.
- **Learning curve:** Developers need to adapt to the GraphQL paradigm and learn the query language. Specially this become a requirement for API consumers.

## Spring GraphQL services

Spring provides a comprehensive set of features to support the development of GraphQL services and clients by integrating the GraphQL Java library. In addition to providing standard Spring features such as test support, security, and observability, it also provides integration with GraphiQL—a browser-based tool to test the GraphQL services. In this section, we will try to develop our first GraphQL service for the BookClub use case.

Spring integrates with the GraphQL Java library to provide a comprehensive set of features to support the development of GraphQL services and clients. In addition to standard Spring features such as test support, security, and observability, it also integrates with GraphiQL, a browser-based tool for testing GraphQL services. In this section, we will develop our first GraphQL service for the BookClub use case.

Unlike RESTful services, which are inherently tied to HTTP, GraphQL is not bound to any specific transport protocol. This flexibility allows us to use various transport protocols with GraphQL. Spring GraphQL supports the following transport options:

- HTTP
- WebSocket
- **Server-Sent Events (SSE)**
- RSocket

To get started with our GraphQL service, download the initial project, which includes a working JPA repository and service implementation, from the companion GitHub repository of this book. Then, add the following starter dependencies to the POM of the file of the project:

```
1. <dependency>
2. <groupId>org.springframework.boot</groupId>
3. <artifactId>spring-boot-starter-graphql</artifactId>
4. </dependency>
5. <dependency>
6. <groupId>org.springframework.boot</groupId>
7. <artifactId>spring-boot-starter-web</artifactId>
8. </dependency>
```

The **spring-boot-starter-graphql** dependency offers key components for building GraphQL services, including the automatic configuration of essential beans and components to set up a GraphQL server. It also adds necessary dependencies like GraphQL Java, automatically loads schemas, exposes a GraphQL endpoint, handles exceptions, and enables development tools like GraphiQL. Essentially, this starter makes sure you have a pre-configured, production-ready setup for developing GraphQL services.

As the next step, let us develop the schema for our service. Create a file named **schema.graphqls** in the **src/main/resources/graphql** directory and add the following code:

```
1. type Book {
2.   bookId: ID
3.   isbn: String
4.   title: String
5.   author: String
6.   publicationYear: Int
7.   numberOfCopies: Int
8.   copies: [BookCopy]
9. }
10.
11. type BookCopy {
12.   copyId: ID
13.   book: Book
14.   isAvailable: Boolean
15. }
```

In the schema above, we have defined the **Book** type with several fields using GraphQL data types. In our case, this mirrors the **Book** domain class, but it does not have to be bound to a specific domain class. You have the flexibility to define schema types based on your business requirements. The **bookId** and **copyId** fields are defined using the **ID** type, representing unique

identifiers.

Next, as shown in the following code, we add a couple of queries using these types:

```
1. type Query {  
2.   bookById(bookId: ID): Book  
3.   booksByTitle(title: String): [Book]  
4.   booksByAuthor(author: String): [Book]  
5. }
```

The first query, **bookById**, fetches a single book by its **bookId**. The other two queries allow searching for books by title and author. Before diving deeper, let us create a controller class and try out these queries.

The complete schema developed so far is provided as follows:

```
1. type Book {  
2.   bookId: ID  
3.   isbn: String  
4.   title: String  
5.   author: String  
6.   publicationYear: Int  
7.   numberOfCopies: Int  
8.   copies: [BookCopy]  
9. }  
10.  
11. type BookCopy {  
12.   copyId: ID  
13.   book: Book  
14.   isAvailable: Boolean  
15. }  
16.  
17.  
18. type Query {  
19.   bookById(bookId: ID): Book  
20.   booksByTitle(title: String): [Book]  
21.   booksByAuthor(author: String): [Book]  
22. }
```

Create a class named **BookController** with the following code:

```
1. package com.bpb.hssb.ch7.bookclub.controller;  
2.  
3. import org.springframework.stereotype.Controller;
```

```
4.
5. @Controller
6. public class BookController {
7.
8.     private BookService bookService;
9.
10.    public BookController(BookService bookService) {
11.        this.bookService = bookService;
12.    }
13.
14. }
```

As you may have noticed, this code is not much different from the initial **BookController** we created for Spring MVC applications in [Chapter 4, Building Spring MVC Web Applications](#). Here, the **@Controller** annotation marks the class as a Spring MVC controller used to handle HTTP requests related to the GraphQL endpoint.

Next, we introduce the following method to process the queries:

```
1. @QueryMapping
2. public Book bookById(@Argument int bookId) throws BookServiceException {
3.     return bookService.getBook(bookId);
4. }
5.
6. @QueryMapping
7. public Iterable<Book> booksByTitle(@Argument String title) throws BookServiceException {
8.     return bookService.findByTitleContaining(title);
9. }
10.
11. @QueryMapping
12. public Iterable<Book> booksByAuthor(@Argument String author) throws BookServiceException {
13.     return bookService.findByAuthorContaining(author);
14. }
```

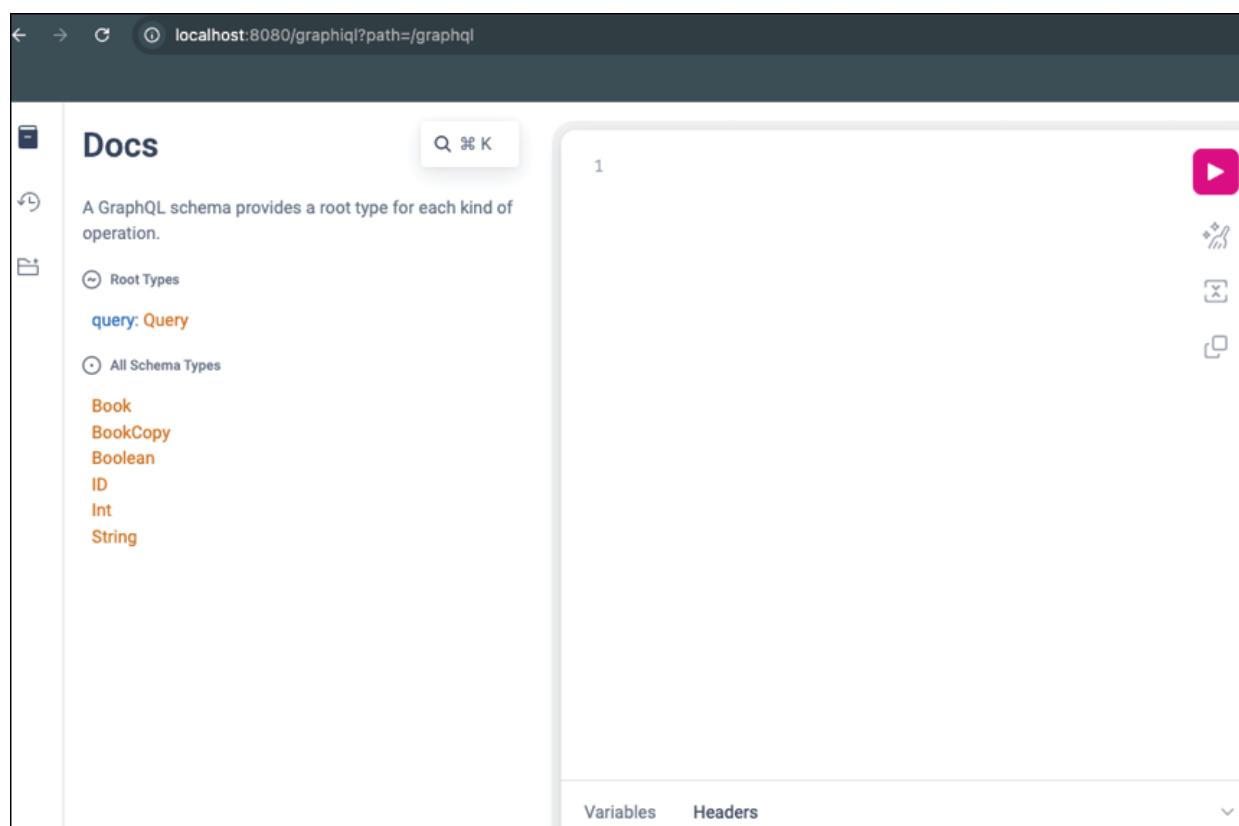
In the code, the **@QueryMapping** annotation binds the method to a query in the GraphQL schema and registers a DataFetcher for the corresponding query field. By default, the query field name is derived from the method name. The **@Argument** annotation is then used to bind GraphQL field arguments to method parameters. Note that two JPA repository methods have been introduced to handle the **booksByAuthor** and **booksByTitle** queries in

the sample project.

Now, we are ready to run and test our first query. Start the **BookclubApplication** using VS Code or Maven, and open the following URL in a web browser:

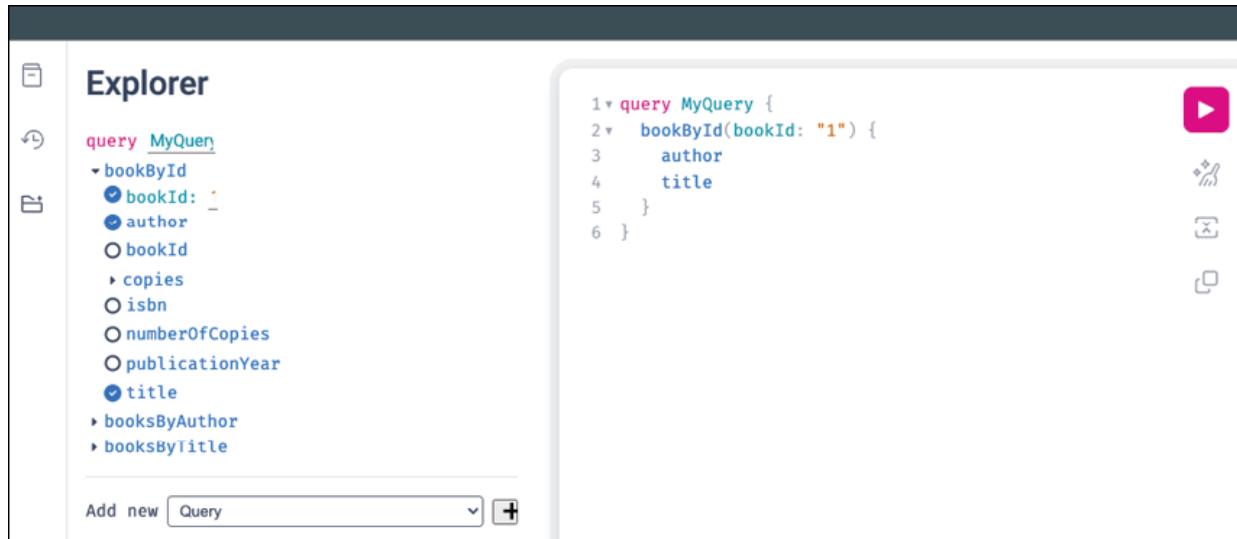
1. <http://localhost:8080/graphiql>

You should see the following GraphQL tool, which we will use to test the service:



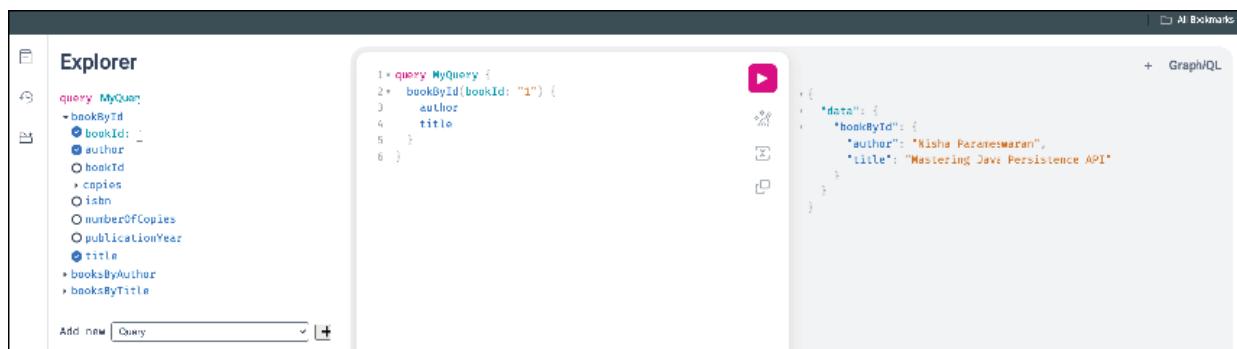
*Figure 7.1: GraphiQL UI*

The GraphiQL interface allows you to type in your queries and execute them against the service we have just started. It also provides easy-to-read documentation for the schema. Additionally, it offers a GraphQL Explorer feature, which we will use next. Click on **GraphQL Explorer** to move to the GraphQL Explorer interface. Here, as shown in *Figure 7.2*, you can visually design your query and add argument values:



**Figure 7.2:** GraphQL Explore with a query

All you need to do now is click the execute button, and you should see the following results:



**Figure 7.3:** GraphQL Explore with a query results

Here is the query we designed using the GraphQL IDE:

```

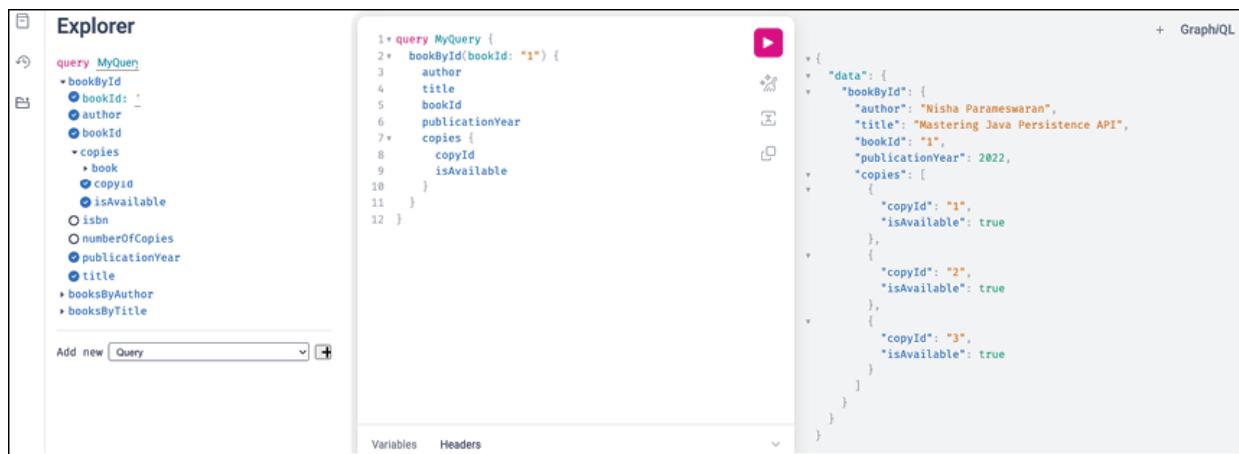
1. query MyQuery {
2.   bookById(bookId: "1") {
3.     author
4.     title
5.   }
6. }

```

In this query, we have set 1 as the value for the **bookId** argument and specified that we want to receive only the author and title of the matching book. Unlike in the RESTful service we developed in the previous chapter, the client here has the flexibility to define and request only the needed data elements. If you look at the **bookById** method in the **BookController** class,

it returns the entire matching **Book** object, but we only received a subset of fields as the response. What happened here is that Spring, by examining the query and the schema, filtered and provided only the required subset of fields.

To explore the query concept further, let us modify our query in the GraphiQL IDE by selecting additional fields. When you rerun the query, you should see the following output, which includes more fields than before:



```

query MyQuery {
  bookById(bookId: "1") {
    author
    title
    bookId
    publicationYear
    copies {
      copyId
      isAvailable
    }
  }
}

```

```

{
  "data": {
    "bookById": {
      "author": "Nisha Parameswaran",
      "title": "Mastering Java Persistence API",
      "bookId": "1",
      "publicationYear": 2022,
      "copies": [
        {
          "copyId": "1",
          "isAvailable": true
        },
        {
          "copyId": "2",
          "isAvailable": true
        },
        {
          "copyId": "3",
          "isAvailable": true
        }
      ]
    }
  }
}

```

Figure 7.4: GraphiQL Explore with a query results

Here is the complete query we used in the latter case:

```

1. query MyQuery {
2.   bookById(bookId: "1") {
3.     author
4.     title
5.     bookId
6.     publicationYear
7.     copies {
8.       copyId
9.       isAvailable
10.    }
11.  }
12. }

```

At this stage, you should have a better understanding of how you can control the service response by specifying a query. Next, let us explore the concept of GraphQL mutations. Start by adding the following code to the schema to introduce an input type that can be used with mutations:

```

1. input BookInput {

```

```
2. isbn: String!
3. title: String!
4. author: String!
5. publicationYear: Int!
6. }
```

In the code, `input` is used to define a special object type that can serve as an argument for queries or mutations. Input types are specifically designed for passing complex objects as arguments. Additionally, the exclamation mark (!) after each field type indicates that these fields are non-nullable.

Next, add the following mutation to the schema:

```
1. type Mutation {
2.   createBook(book: BookInput!): Book!
3.   updateBook(book: BookInput!): Book!
4. }
```

Both mutations take **BookInput** as a non-nullable input and return a newly created or updated Book type as the response. The complete schema is as follows:

```
1. type Book {
2.   bookId: ID
3.   isbn: String
4.   title: String
5.   author: String
6.   publicationYear: Int
7.   numberOfCopies: Int
8.   copies: [BookCopy]
9. }
10.
11. type BookCopy {
12.   copyId: ID
13.   book: Book
14.   isAvailable: Boolean
15. }
16.
17. input BookInput {
18.   isbn: String!
19.   title: String!
20.   author: String!
21.   publicationYear: Int!
22. }
```

```

23.
24. type Query {
25.   bookById(bookId: ID): Book
26.   booksByTitle(title: String): [Book]
27.   booksByAuthor(author: String): [Book]
28. }
29.
30. type Mutation {
31.   createBook(book: BookInput!): Book!
32.   updateBook(book: BookInput!): Book!
33. }
34.

```

As the final step, we need to add methods to handle the defined mutation in our controller. Add the following method to the **BookController** class:

```

1.  @MutationMapping
2.  public Book createBook(@Argument BookInput book) throws BookServiceException {
3.    Book bookToAdd = Book.builder().isbn(book.getIsbn()).title(book.getTitle())
4.      .author(book.getAuthor()).publicationYear(book.getPublicationYear()).build();
5.    return bookService.createBook(bookToAdd);
6.  }
7.
8.  @MutationMapping
9.  public Book updateBook(@Argument BookInput book) throws BookServiceException {
10.   Book bookToUpdate = Book.builder().isbn(book.getIsbn()).title(book.getTitle())
11.     .author(book.getAuthor()).publicationYear(book.getPublicationYear()).build();
12.   return bookService.updateBook(bookToUpdate);
13. }

```

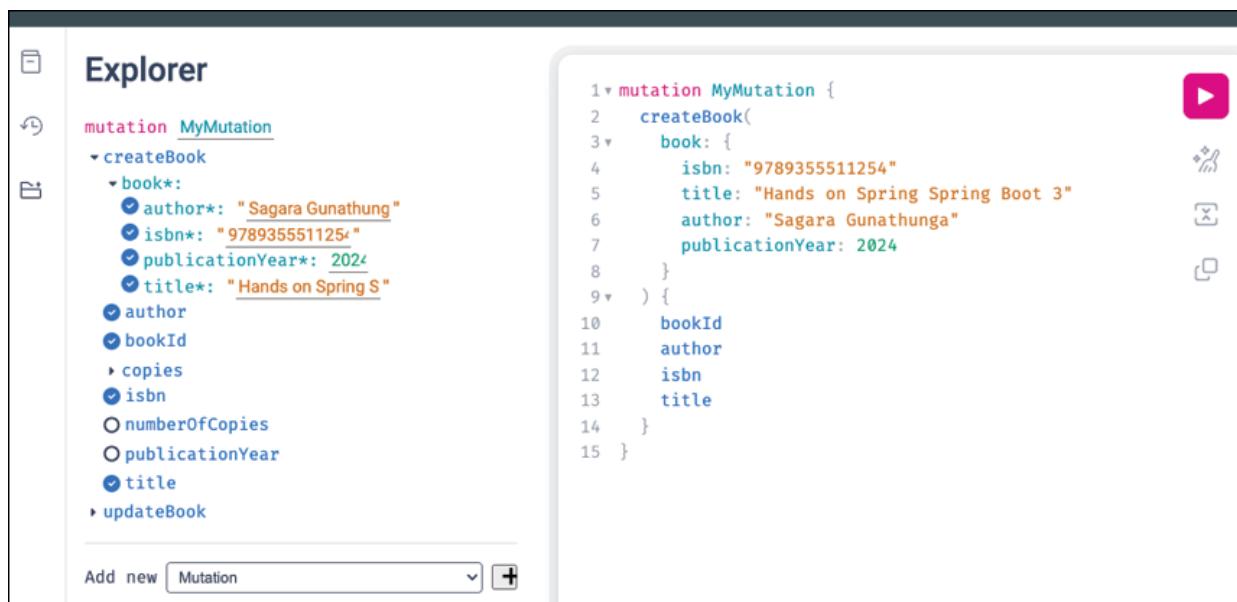
In this code, the **@MutationMapping** annotation is used to bind the **updateBook** method to a mutation field in the GraphQL schema and register the method as a **DataFetcher** for the corresponding mutation. By default, the mutation field name is derived from the method name unless explicitly specified.

Now, if you rerun the application and refresh the GraphiQL interface, you should see that the schema documentation has been updated with the newly added mutations, as shown in the following figure:



*Figure 7.5: GraphiQL with a mutation*

As before, we can use the GraphQL Explorer feature in the GraphiQL IDE to design the mutation, as follows:



```
1 mutation MyMutation {
2   createBook(
3     book: {
4       isbn: "9789355511254"
5       title: "Hands on Spring Spring Boot 3"
6       author: "Sagara Gunathunga"
7       publicationYear: 2024
8     }
9   ) {
10     bookId
11     author
12     isbn
13     title
14   }
15 }
```

*Figure 7.6: GraphiQL Explore with a mutation*

When you execute the mutation, you should see the results displayed as follows:



Figure 7.7: GraphQL Explore with mutation results

Here is the full mutation we used earlier:

```

1. mutation MyMutation {
2.   createBook(
3.     book: {
4.       isbn: "9789355511254"
5.       title: "Hands on Spring Spring Boot 3"
6.       author: "Sagara Gunathunga"
7.       publicationYear: 2024
8.     }
9.   ) {
10.   bookId
11.   author
12.   isbn
13.   title
14. }
15. }

```

In the mutation code, we first defined the input values for the new book and passed them as an argument. Additionally, we specified which fields we want to receive in the response after the book has been added to the system.

Before we conclude this section, let us explore how Spring GraphQL handles errors. Add the following query to the GraphQL Explorer in the GraphQL IDE:

```

1. query MyQuery {
2.   bookById(bookId: "5000") {
3.     author
4.     bookId
5.   }
6. }

```

In the query, we are passing a **bookId** for a non-existing book to the service. When you execute this query, you will see the following error message:

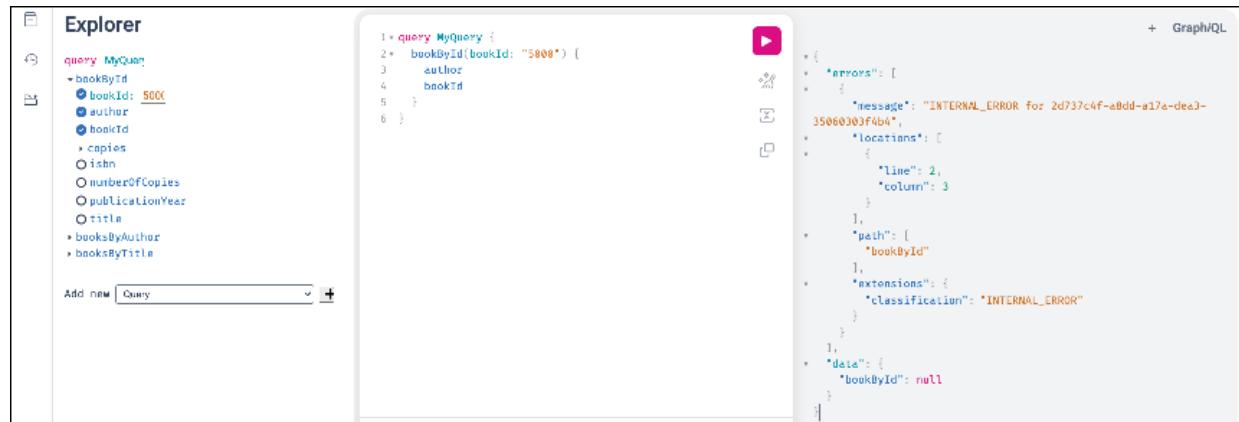


Figure 7.8: GraphQL Explorer with an error

As you can see, Spring GraphQL provides a meaningful error message without exposing any internal stack trace. If needed, you can customize the automatic error handling mechanism.

In this section, we have successfully developed our first GraphQL service with both queries and mutations.

## GraphQL service testing

In this section, let us write a test case for the GraphQL controller we developed earlier. First, add the following test dependency to the project's POM file:

1. <dependency>
2. <groupId>org.springframework.graphql</groupId>
3. <artifactId>spring-graphql-test</artifactId>
4. <scope>test</scope>
5. </dependency>

The **spring-graphql-test** dependency provides dedicated support for testing GraphQL requests and offers tools for testing GraphQL requests over various transports, including HTTP, WebSocket, and RSocket, as well as testing directly against a server. Additionally, it enables auto-configuration for tests and allows developers to test GraphQL controllers in isolation without loading the entire application context.

Next, create the **BookControllerTests** class with the following code:

```

1. package com.bpb.hssb.ch7.bookclub.controller;
2.
3. import org.springframework.beans.factory.annotation.Autowired;
4. import org.springframework.boot.test.autoconfigure.graphql.GraphQlTest;
5. import org.springframework.boot.test.mock.mockito.MockBean;
6. import org.springframework.graphql.test.tester.GraphQlTester;
7.
8. import com.bpb.hssb.ch7.bookclub.service.BookService;
9.
10. @GraphQlTest(BookController.class)
11. public class BookControllerTests {
12.
13.     @Autowired
14.     private GraphQlTester graphQlTester;
15.
16.     @MockBean
17.     private BookService bookService;
18.
19. }
20.

```

In the code, the **@GraphQlTest** annotation is configured to load only a subset of the application configuration needed to test **BookController** and sets up everything required for the test cases. We have also injected **GraphQlTester**, which is used to execute GraphQL queries, mutations, and subscriptions in a test environment, providing methods for asserting the response, including JSON path matching. We have injected **BookService** as a mock for use in testing.

Now, we are ready to add our first test case. Create a file named **bookDetails.graphql** in the **src/test/resources/graphql-test** directory with the following query that we will use in our test case:

```

1. query bookDetails($bookId: ID) {
2.     bookById(bookId: $bookId) {
3.         bookId
4.         isbn
5.         title
6.     }
7. }

```

Next, add the following test method to the **BookControllerTests** class:

```

1.  @Test
2.  void testBookDetails() throws Exception {
3.
4.      Book book = Book.builder().bookId(1).author("Nisha").title("Java 17")
5.          .isbn("9789355511263").publicationYear(2022).build();
6.      when(bookService.getBook(1)).thenReturn(book);
7.
8.      graphQlTester.documentName("bookDetails")
9.          .variable("bookId", 1)
10.         .execute()
11.         .path("bookById")
12.             .path("bookById.bookId").entity(Integer.class).isEqualTo(1)
13.             .path("bookById.isbn").entity(String.class).isEqualTo("9789355511263")
14.             .path("bookById.title").entity(String.class).isEqualTo("Java 17");
15.
16.  }

```

In the code above, we used **GraphQlTester** to execute a query and verify the results.

We can also add a test case for the **createBook** mutation. Create a file named **createBook.graphql** in the **src/test/resources/graphql-test** directory with the following mutation code:

```

1. mutation createBook($book: BookInput!) {
2.   createBook(book: $book) {
3.     bookId
4.     isbn
5.     title
6.     author
7.     publicationYear
8.   }
9. }

```

Next, add the following test case to the **BookControllerTests** class:

```

1.  @Test
2.  void testCreateBook() throws BookServiceException {
3.
4.      Map<String, Object> bookInput = Map.of(
5.          "isbn", "1234567890",
6.          "title", "Test Book",
7.          "author", "Test Author",
8.          "publicationYear", 2023);

```

```

9.
10.    Book createdBook = Book.builder()
11.        .bookId(1)
12.        .isbn("1234567890")
13.        .title("Test Book")
14.        .author("Test Author")
15.        .publicationYear(2023)
16.        .build();
17.
18.    when(bookService.createBook(any(Book.class))).thenReturn(createdBook);
19.
20.    graphQlTester.documentName("createBook")
21.        .variable("book", bookInput)
22.        .execute()
23.        .path("createBook")
24.        .path("createBook.bookId").entity(Integer.class).isEqualTo(1)
25.        .path("createBook.isbn").entity(String.class).isEqualTo("1234567890")
26.        .path("createBook.title").entity(String.class).isEqualTo("Test Book")
27.        .path("createBook.author").entity(String.class).isEqualTo("Test Author")
28.        .path("createBook.publicationYear").entity(Integer.class).isEqualTo(2023);
29.    }

```

In the code, we again used **GraphQLTester** to execute a mutation and verify the results.

At this point, you should have a good grasp of how to write test cases for GraphQL services, and you have noticed it is not fundamentally different from the other test cases we have written so far.

## GraphQL clients

In this section, we will write a GraphQL client for the service developed in the previous sections. Start by visiting Spring Initializr and creating a project with the following configuration:

**Project : Maven**

**Language : Java**

**Spring Boot : Latest stable version ( e.g. – 3.3.0)**

**Project Metadata :**

**Group : com.bpb.hssb.ch7**

**Artifact : graphql-clients**

**Name : graphql -clients**

**Packaging : Jar (Default)**

**Java : 17 (Default)**

**Dependencies: Spring web, Spring GraphQL, Lombok**

Download and open this project using the VS Code IDE. Since we do not need a web server to run the client code, you can disable the automatic startup of the web server by adding the following property to the **application.properties** file. Otherwise, **spring-boot-starter-web** will automatically configure an embedded web server for us.

1. `spring.main.web-application-type=none`

Next, create a client-side domain class named **Book** using the following code:

```
1. package com.bpb.hssb.ch6.bookclub.client;
2.
3. import lombok.AllArgsConstructor;
4. import lombok.Builder;
5. import lombok.Getter;
6. import lombok.NoArgsConstructor;
7. import lombok.Setter;
8. import lombok.ToString;
9.
10. @Getter
11. @Setter
12. @NoArgsConstructor
13. @AllArgsConstructor
14. @Builder
15. @ToString
16. public class Book {
17.
18.     private int bookId;
19.     private String isbn;
20.     private String title;
21.     private String author;
22.     private int publicationYear;
23. }
```

In the code, we have simply used a set of Lombok annotations that you are already familiar with.

Spring GraphQL provides an interface called **GraphQlClient**, which defines a common workflow for GraphQL requests across all supported transports. This allows you to switch the underlying transport without modifying your client code. The following are transport-specific implementations of the **GraphQlClient** interface:

- **HttpSyncGraphQlClient**: This implementation uses **RestClient** to execute GraphQL requests over HTTP through a blocking transport.
- **HttpGraphQlClient**: This implementation uses **WebClient** to execute GraphQL requests over HTTP through a non-blocking transport.
- **WebSocketGraphQlClient**: This implementation uses a WebSocket connection to execute GraphQL requests.
- **RSocketGraphQlClient**: This implementation uses **RSocketRequester** to execute GraphQL requests over **RSocket** requests.

Next, create a class named **BookGraphQlClient** with the following code:

```
1. package com.bpb.hssb.ch6.bookclub.client;
2.
3. import org.springframework.beans.factory.annotation.Autowired;
4. import org.springframework.boot.CommandLineRunner;
5. import org.springframework.boot.SpringApplication;
6. import org.springframework.boot.autoconfigure.SpringBootApplication;
7. import org.springframework.graphql.client.GraphQlClient;
8.
9. import org.slf4j.Logger;
10. import org.slf4j.LoggerFactory;
11.
12. @SpringBootApplication
13. public class BookGraphQlClient implements CommandLineRunner {
14.
15.     private static final Logger logger =
16.         LoggerFactory.getLogger(BookGraphQlClient.class);
17.
18.     private GraphQlClient graphQlClient;
19.
20.     public static void main(String[] args) {
21.         SpringApplication.run(BookGraphQlClient.class, args);
22.     }
23. }
```

```
24. @Override
25. public void run(String... args) throws Exception {
26.
27.
28. }
29.
30. }
```

In the code, we have injected **HttpSyncGraphQIClient**, which we will use to call the GraphQL service.

Next, create a file named **createBook.graphql** in the **src/main/resources/graphql-documents** directory. When we specify the filename of this mutation in our client code, Spring ensures the correct mutation is loaded from this location.

```
1. mutation createBook($book: BookInput!) {
2.   createBook(book: $book) {
3.     bookId
4.     isbn
5.     title
6.     author
7.     publicationYear
8.   }
9. }
```

Now, add the following method to the **BookGraphQIClient** class to create a book:

```
1.
2.   public Book createBook() {
3.
4.     Map<String, Object> bookInput = Map.of(
5.       "isbn", "9789355511254",
6.       "title", "Hands on Spring Spring Boot 3",
7.       "author", "Sagara Gunathunga",
8.       "publicationYear", 2024);
9.
10.    Mono<ClientGraphQIResponse> responseMono = graphQIClient.documentName("createBook
11.      ")
12.      .variable("book", bookInput)
13.      .execute();
14.
```

```
13. ClientGraphQlResponse response = responseMono.block();
14.
15. return response.field("createBook").toEntity(Book.class);
16.
17. }
```

In the code, we instructed Spring to load the **createBook** mutation from the file system and then constructed the input for the mutation as a Java Map. After executing the mutation, we can access the result using **Mono<ClientGraphQlResponse>**. We are not discussing Mono here and will leave it for the next chapter.

We also call a query in the GraphQL service. To do that, first, create a file named **bookDetails.graphql** in the **src/main/resources/graphql-documents** directory. When we specify the filename of this query in our client code, Spring ensures the correct query is loaded from this location.

```
1. query bookDetails($bookId: ID) {
2.   bookById(bookId: $bookId) {
3.     bookId
4.     isbn
5.     title
6.   }
7. }
```

8.

9.

Next, add the following method to the **BookGraphQlClient** class to query a book by providing its **bookId**:

```
1. public void printBookInfo(int bookId) {
2.
3.   Mono<ClientGraphQlResponse> responseMono = graphQlClient.documentName("bookDetail
s")
4.     .variable("bookId",bookId)
5.     .execute();
6.
7.   ClientGraphQlResponse response = responseMono.block();
8.   Book book = response.field("bookById").toEntity(Book.class);
9.   logger.info(" Book : " + book);
10. }
```

The code for the method above is similar to the previous one, except that we used a mutation earlier and are now using a query instead.

We can also update the run method to call the two methods we just created:

```
1. public void run(String... args) throws Exception {  
2.  
3.     Book book = createBook();  
4.     printBookInfo(book.getBookId());  
5. }
```

Finally, before running the client code, we need to specify the Spring bean configuration. This can be done by adding the **BookclubConfiguration** class with the following code:

```
1. package com.bpb.hssb.ch6.bookclub.client;  
2.  
3. import org.springframework.beans.factory.annotation.Value;  
4. import org.springframework.context.annotation.Bean;  
5. import org.springframework.context.annotation.Configuration;  
6. import org.springframework.graphql.client.HttpSyncGraphQlClient;  
7. import org.springframework.web.client.RestClient;  
8.  
9. @Configuration  
10. public class BookclubConfiguration {  
11.  
12.     @Value("${bookclub.api.baseurl}")  
13.     private String baseurl;  
14.  
15.     @Bean  
16.     public RestClient restClient() {  
17.  
18.         return RestClient.builder()  
19.             .baseUrl(baseurl)  
20.             .build();  
21.     }  
22.  
23.     @Bean  
24.     public HttpSyncGraphQlClient  
25.         httpSyncGraphQlClient(RestClient restClient) {  
26.             return HttpSyncGraphQlClient.create(restClient);  
27.         }  
28. }
```

As you can easily see from the code above, **HttpSyncGraphQlClient** uses

**RestClient** under the hood. Now, you can run your client code successfully. Make sure the GraphQL service is running before you execute the client code.

## Securing GraphQL services

As we have already used a couple of times, the Spring Security project can also be used to secure GraphQL services. You can easily enable **Basic Authentication** for your service, or if you need a more sophisticated security scheme, you can use OAuth2-based JWT token security. In this section, we will implement Basic Authentication for our GraphQL service.

First, add the necessary Spring Security dependencies to your project, as follows:

```
1. <dependency>
2.   <groupId>org.springframework.boot</groupId>
3.   <artifactId>spring-boot-starter-security</artifactId>
4. </dependency>
5. <dependency>
6.   <groupId>org.springframework.security</groupId>
7.   <artifactId>spring-security-test</artifactId>
8.   <scope>test</scope>
9. </dependency>
```

Next, add the **SecurityConfiguration** class with the following code to enable Basic Authentication for our service:

```
1. package com.bpb.hssb.ch7.bookclub.config;
2.
3. import org.springframework.context.annotation.Bean;
4. import org.springframework.context.annotation.Configuration;
5. import org.springframework.security.config.Customizer;
6. import org.springframework.security.config.annotation.
   web.builders.HttpSecurity;
7. import org.springframework.security.config.http.SessionCreationPolicy;
8. import org.springframework.security.web.SecurityFilterChain;
9.
10. @Configuration
11. public class SecurityConfiguration {
12.   @Bean
13.   public SecurityFilterChain
```

```
configure(HttpSecurity http) throws Exception {  
14.    return http  
15.        .csrf(csrf -> csrf.disable())  
16.        .authorizeRequests(auth -> {  
17.            auth.anyRequest().authenticated();  
18.        })  
19.        .sessionManagement(  
20.            session -> session.sessionCreationPolicy  
            (SessionCreationPolicy.STATELESS))  
21.        .httpBasic(Customizer.withDefaults())  
22.        .build();  
23.    }  
24.  
25. }
```

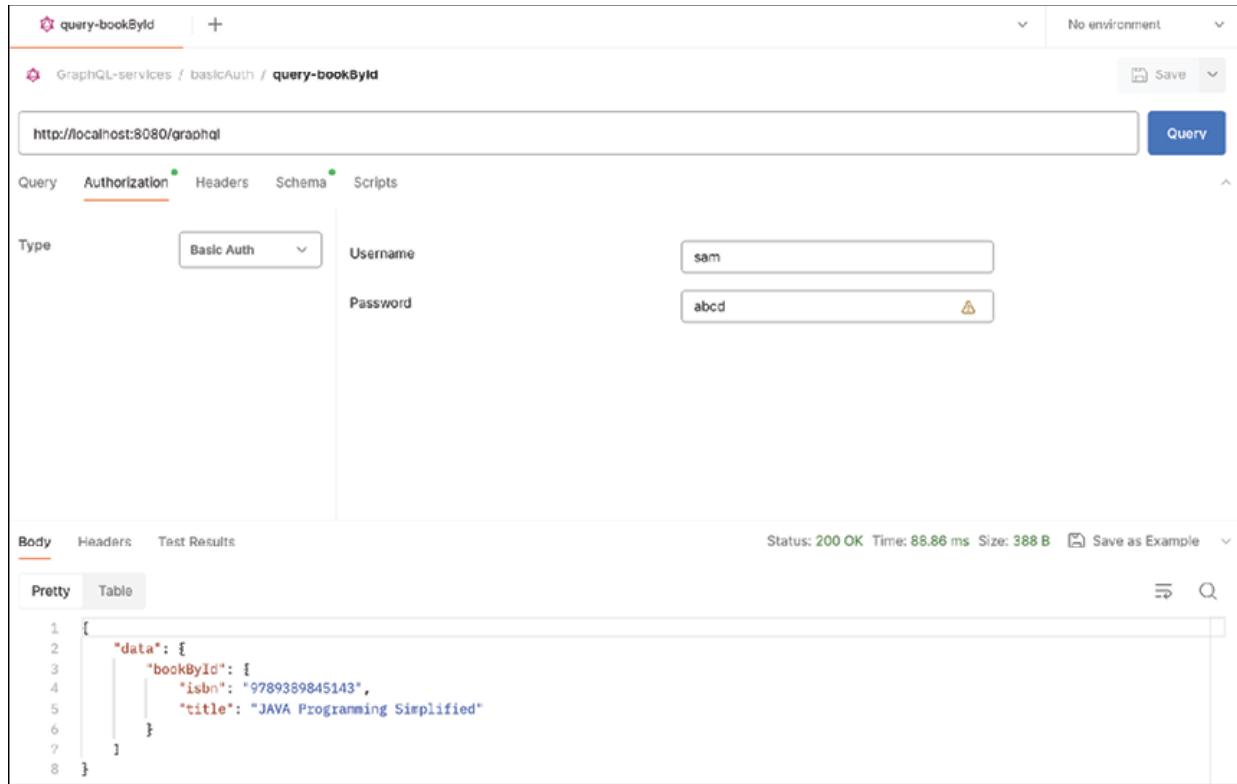
In the code above, we defined the **SecurityFilterChain** bean, which configures security settings for HTTP requests. We have also set it up to authenticate all requests using the HTTP Basic Authentication schema with default settings.

Next, add a test user by including the following lines in the **application.properties** file:

1. `spring.security.user.name=sam`
2. `spring.security.user.password=abcd`
3. `spring.security.user.roles=ADMIN`

Now, if you run the service and try to access the GraphiQL IDE, it will prompt you to enter a username and password. Once you provide the credentials of the test user, you should be able to use the GraphiQL IDE just as you did before.

The following figure shows how to provide user credentials if you are using Postman as your GraphQL client:



*Figure 7.9: Calling GraphQL service using Postman*

## Conclusion

In this chapter, we introduced GraphQL and developed a GraphQL service and client using Spring Boot. We also covered how to write test cases for GraphQL and how to secure GraphQL services.

In the next chapter, we will learn the basics of reactive programming and implement your first reactive application using Spring Boot.

## Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



*OceanofPDF.com*

# CHAPTER 8

# Building Reactive Spring Applications

## Introduction

This chapter introduces the reactive programming model and discusses its benefits. Then, we will look into implementing a reactive service using Spring WebFlux and explore supporting persistence with Spring Data R2DBC. Finally, we will discuss testing and implementing reactive clients.

## Structure

The chapter covers the following topics:

- Introduction to reactive programming
- Spring reactive services
- Writing test for reactive services
- Functional endpoint model
- Building reactive clients

## Objectives

By the end of this chapter, you will understand reactive programming and how it is fundamentally different from imperative programming. You will also learn how to build reactive services using Spring WebFlux, write test

cases, and build reactive client applications.

## Introduction to reactive programming

Imagine a small corner café on a quiet street, where one or two people work most of the time. When you walk in, a staff member greets you and takes your order. This same staff member then prepares your meal and drink, handling one task at a time while you relax, perhaps reading a newspaper or watching the news. Once your order is ready and served, the staff member moves on to the next customer. In this scenario, the staff attends to customers sequentially, serving one before moving on to the next. While some tasks can happen in parallel, the focus remains on serving a single customer at a time. This approach might mean other customers have to wait, and equipment may not be fully utilized, but for a small café on a quiet street, it works perfectly well.

The web applications, RESTful services, and GraphQL services we have developed in this book so far behave much like this café. Each waits for a request, and once it arrives, they execute a series of tasks in a specific sequence one at a time; often, the next task cannot be executed until the previous task has been completed. For example, first, a thread validates the request, then checks the inventory database, processes payment through a payment gateway, and finally, notifies the customer of the delivery details. This is what we call an imperative programming model. If we compare this model to our café analogy, each customer represents a task, while the staff member symbolizes a thread. The sequential nature of the service mirrors how imperative programming handles tasks one by one, with limited parallelism reflecting how some operations can occur concurrently, but the primary focus remains on a single task. Much like the small corner café, the imperative programming model is simple, easy to understand, and familiar to developers. It is also perfectly suited for most applications we develop today. So, why should we consider looking beyond this model? To answer that, let us explore a real-world analogy related to our corner café.

Now, imagine a fast-food restaurant located in the main subway station of the city center. Obviously, it cannot operate the same way as a small corner café. The fast-food restaurant needs to serve a large number of people efficiently, with minimal wait time, while making the best use of its staff and

equipment.

In this setting, customers place orders through kiosks, and the preparation of each order is handled by several staff members, each assigned a specific task. For example, one staff member ensures there are enough fries and adds them to orders, while another makes sure there are enough ingredients for burgers and prepares them.

Unlike the corner café, no single staff member is responsible for an entire order. Instead, each person focuses on a specific task and moves on to the next one. This allows them to scale and serve a larger number of customers, reducing wait times and making more efficient use of both staff and equipment. This model directly mirrors how reactive programming operates: it is event-driven, with kiosk orders representing events that trigger the system; it allows for concurrent processing as multiple staff work on different aspects of orders simultaneously; it offers scalability by enabling the addition of more specialized workers; and it enhances efficiency through optimized resource utilization. In essence, just like in this fast-food restaurant, reactive programming breaks tasks down into smaller, manageable pieces that can be processed independently and concurrently, resulting in more responsive and efficient applications.

However, the model used by the fast-food restaurant is much more complex, requiring specialized equipment, and would clearly be too much for the small corner café we discussed earlier. Similarly, the reactive programming model is not a one-size-fits-all solution. It should only be used in situations where its benefits outweigh those of the imperative programming model. The amount of data generated in modern social media platforms and data generated from IoT devices from large industrial and agricultural sites can be good candidates to apply the reactive programming model. For example, the vast amounts of data generated by modern social media platforms and IoT devices from large industrial and agricultural sites make them ideal candidates for applying the reactive programming model.

Before we further our discussion, there are a couple of important points regarding the reactive programming model. Let us return to our fast-food restaurant analogy. Imagine a situation where a new staff member approaches a customer, takes their order, and then prepares the entire order by themselves instead of focusing on the specific task assigned to them. This

would disrupt the whole system, causing the fast-food restaurant to operate more like the small corner café.

Similarly, when writing a reactive application, you must ensure that all tasks, such as validating inputs, querying a database, or calling an HTTP endpoint, are performed in the reactive processing. Additionally, you should use libraries that support reactive processing. Otherwise, your application will not truly be reactive, and you will not get the full benefit of the reactive programming model.

Another important concept related to the reactive programming model is support for back pressure. Again, taking a real-world situation, imagine you are trying to take notes while someone is speaking rapidly. If they talk too fast, you might miss information or get overwhelmed. Back pressure is your ability to say, *Slow down, please!* to the speaker. In technical terms, it is a way for the part of the system receiving data (the consumer) to tell the part sending data (the producer) to slow down when it is getting too much to handle. This is important because it prevents the system from becoming overloaded, which could lead to crashes or lost data. Given applications of reactive programming involve a massive amount of data processing, the support for back pressure is very important.

Technically, we can define reactive programming as an asynchronous, non-blocking, and event-driven programming model. Some of the important characteristics of reactive programming include:

- **Event-driven:** Reactive programming reacts to changes and events rather than waiting for them to happen.
- **Asynchronous data streams:** Reactive programming is centered around data streams or sequences of events that are processed asynchronously.
- **Non-blocking:** Reactive programming allows for concurrent processing of events without blocking the main thread.
- **Data-oriented and declarative:** Reactive programming emphasizes the flow and transformation of data rather than control flow.
- **Responsiveness:** Reactive programming reacts and responds in a timely manner to incoming events or user interactions.

Although the concept of reactive programming dates back several decades,

the creation of *The Reactive Manifesto* in 2013 by a group of developers marked a significant milestone. The next major development came in 2015 with the Reactive Streams initiative, which aimed to provide a standard for asynchronous stream processing with non-blocking and backpressure.

Support for reactive programming in Java began with the introduction of the `java.util.concurrent.Flow` interface in Java 9. Today, there are popular reactive Java libraries such as RxJava and Reactor. The Spring project introduced support for reactive programming in Spring Framework 5.0 with the addition of the WebFlux module.

## Spring reactive services

Spring WebFlux is a reactive web framework introduced in Spring 5, designed to build scalable, non-blocking web applications. It provides an alternative to the traditional Spring MVC, offering full support for reactive programming paradigms. WebFlux supports both annotation-based and functional programming models, allowing developers to choose their preferred style. It is compatible with reactive databases and can run on both servlet and non-servlet containers like Netty, enhancing its versatility.

Spring uses Reactor as its default implementation of the Reactive Streams specification, which is what we will use in this chapter. However, once you grasp the core concepts, you can easily switch to RxJava for your Spring Services as well.

In reactive programming, a stream is a sequence of ongoing events ordered in time. It serves as the core abstraction in reactive programming, representing a flow of data that can be observed and manipulated over time. When you apply an operation to items in a stream, that operation is performed on each element as it flows through, resulting in the creation of a new stream. Additionally, multiple operations can be chained together, forming a pipeline of transformations that each element undergoes.

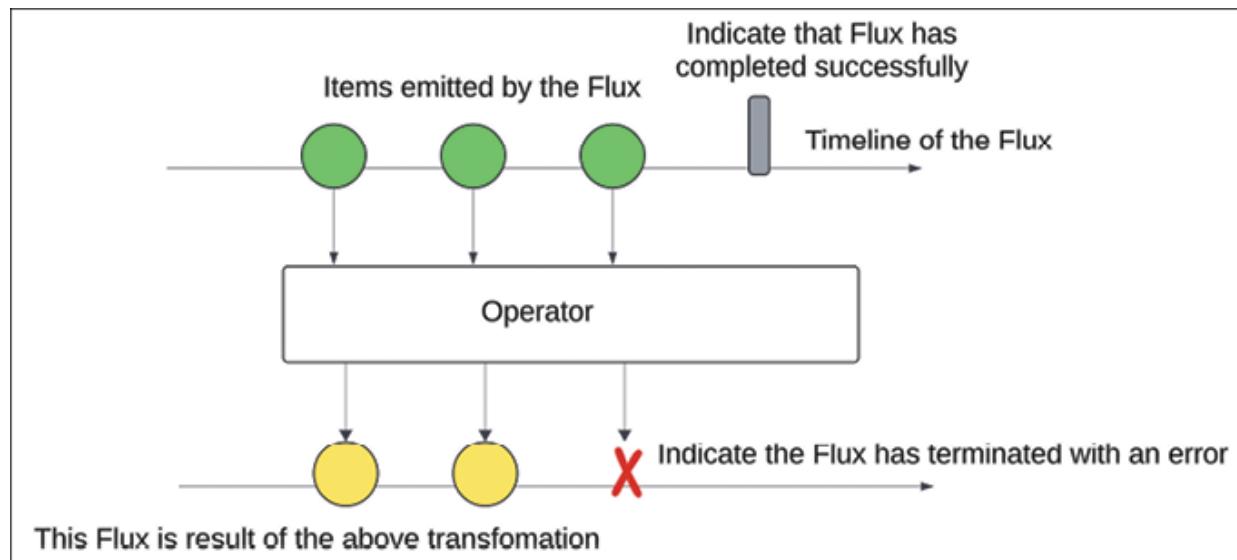
Streams can also be configured as parallel streams, allowing operations to be applied concurrently to different elements, potentially improving performance on multi-core systems.

In real-world terms, streams are similar to underground subway systems. You board a train at your local station heading in one direction, then transfer

to another train at an intersecting station, which is going in a different direction. You might repeat this process multiple times to reach your destination. In this analogy, streams are like trains heading in a particular direction, and whenever you make a change, it is akin to boarding a different train, representing a new stream.

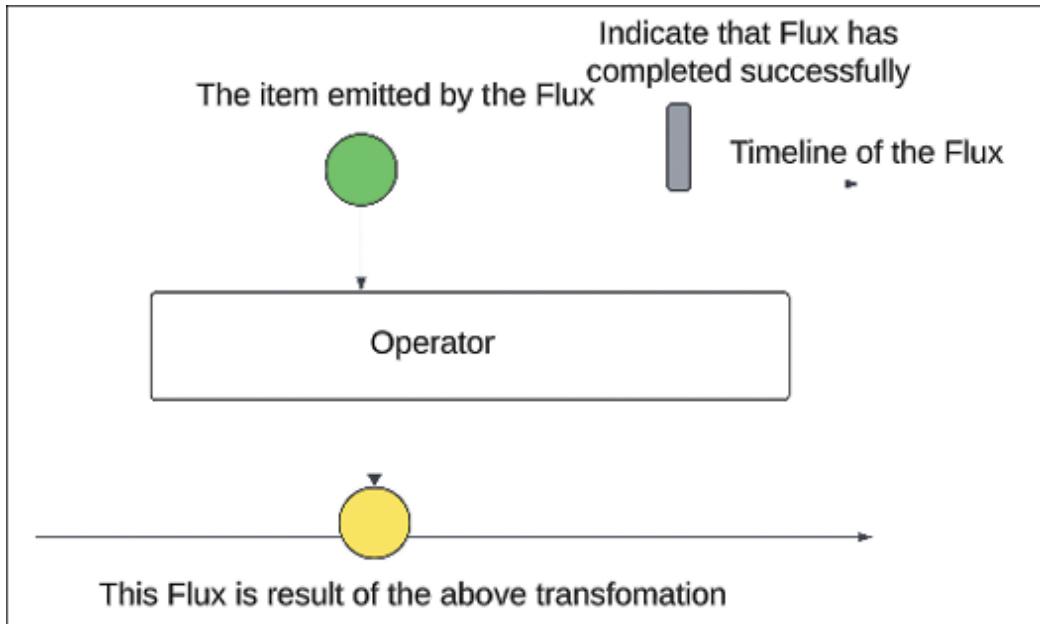
There are two main types of streams in the Reactor project, which is used by Spring WebFlux as the default implementation: *Flux* and *Mono*.

Flux represents a stream of 0 to N elements. When you apply operations to a Flux, the result is a new Flux, as shown in the following figure:



**Figure 8.1:** Visualization of applying an operator on a Flux

Mono represents a stream of 0 or 1 element. Similarly, applying operations to a Mono results in a new Mono, as depicted in the following figure:



*Figure 8.2: Visualization of applying an operator on a Mono*

The Reactive Stream specification defines two main interfaces that are very important for our discussion: **Publisher** and **Subscriber**. First, let us look at the **Publisher** interface:

```

1. public interface Publisher<T> {
2.
3.     public void subscribe(Subscriber<? super T> s);
4.
5. }
```

The **Publisher** is a generic interface where **T** represents the type of elements that the **Publisher** will produce. The `subscribe` method is used to request the **Publisher** to start streaming data to the provided **Subscriber**. This interface is a key component of reactive programming, enabling the creation of data streams that can be consumed asynchronously while providing support for back pressure:

```

1. public interface Subscriber<T> {
2.
3.     public void onSubscribe(Subscription s);
4.     public void onNext(T t);
5.     public void onError(Throwable t);
6.     public void onComplete();
7.
8. }
```

As shown in the above code segment, the **Subscriber** interface is another key

component of the Reactive Streams specification. It is a generic interface where **T** represents the type of elements that the **Subscriber** will consume. The **Subscriber** interface defines the methods that a subscriber must implement to receive data from a **Publisher** and provides a mechanism for the **Publisher** to push data to the Subscriber.

A brief description of each method is provided as follows:

- **onSubscribe**: This method is called when the **Subscriber** is first subscribed to a **Publisher**. It provides a **Subscription** object, allowing the **Subscriber** to control the flow of data, including back pressure. It is always the first method called and is invoked exactly once per subscription.
- **onNext**: This method is called when the **Publisher** emits a new item. It may be called 0 or more times, depending on the available data elements.
- **onError**: This method is invoked when the **Publisher** encounters an error. The **Throwable t** represents the error that occurred.
- **onComplete**: This method is called when the **Publisher** has finished emitting all items, signaling that no more data elements will be emitted.

As we covered most of the important points related to reactive programming in Spring, now let us try to build our very first reactive service using Spring Boot.

First, visit Spring Initializr at <https://start.spring.io> and provide the required details as follows. Once you download the generated project as a zip file, you and extract it and open it using VS Code.

**Project : Maven**

**Language : Java**

**Spring Boot : Latest stable version ( e.g. – 3.3.0)**

**Project Metadata :**

**Group : com.bpb.hssb.ch8**

**Artifact : helloworld**

**Name : helloworld**

**Packaging : Jar (Default)**

**Java : 17 (Default)**

**Dependencies: webflux**

Create a class called **HelloworldFlux.java** with the following code:

```
1. package com.bpb.hssb.ch8.helloworld;
2.
3. import reactor.core.publisher.Flux;
4.
5. public class HelloworldFlux {
6.
7.     public static void main(String[] args) {
8.
9.         Flux<String> frutFlux = Flux.just("Apples", "Oranges", "Grapes", "Mangoes", "Bananas" );
10.
11.        frutFlux.log().subscribe();
12.
13.        frutFlux.subscribe(frut -> System.out.println(frut));
14.
15.        frutFlux.take(3).subscribe(frut -> System.out.println(frut));
16.
17.    }
18. }
19.
```

In *line 9*, we have created a reactive stream of type **Flux** using the **just** method, which creates a **Flux** from a fixed set of values. There are a couple of similar methods are available to create Flux from various data sources. In our case, the **frutFlux** contains a set of String elements representing names of fruits.

Then, in *line 11*, we log all events in the Flux and then subscribe to it. The **subscribe** method without arguments means it will consume the items but not do anything with them. Executing this line results in printing all the events in the console, as shown in the following figure, this would be a good approach to see what events are occurring in the stream:



The screenshot shows a terminal window with the following content:

```
PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL PORTS
● + 1-helloworld git:(master) ✘ /usr/bin/env /Library/Java/JavaVirtualMachines/jdk-17.0.2.jdk/Contents/Home/bin/java @var/folders/2g/tj3r6v
s13vz808xnrqkh6wr0000gn/T/cp_bf84qlv4layd77ilbuvprwz.argfile com.bpb.hssb.ch8.helloworld.HelloworldFlux
19:33:29.644 [main] INFO reactor.Flux.Array.1 — | onSubscribe([Synchronous Fuseable] FluxArray.ArraySubscription)
19:33:29.648 [main] INFO reactor.Flux.Array.1 — | request(unbounded)
19:33:29.648 [main] INFO reactor.Flux.Array.1 — | onNext(Apples)
19:33:29.648 [main] INFO reactor.Flux.Array.1 — | onNext(Oranges)
19:33:29.648 [main] INFO reactor.Flux.Array.1 — | onNext(Grapes)
19:33:29.648 [main] INFO reactor.Flux.Array.1 — | onNext(Mangoes)
19:33:29.648 [main] INFO reactor.Flux.Array.1 — | onNext(Bananas)
19:33:29.648 [main] INFO reactor.Flux.Array.1 — | onComplete()
○ + 1-helloworld git:(master) ✘
```

**Figure 8.3:** Logs showing the events that occurred in a Flux

Then, in *line 13*, we subscribe to the Flux again, this time with a **lambda** function that prints each item to the console.

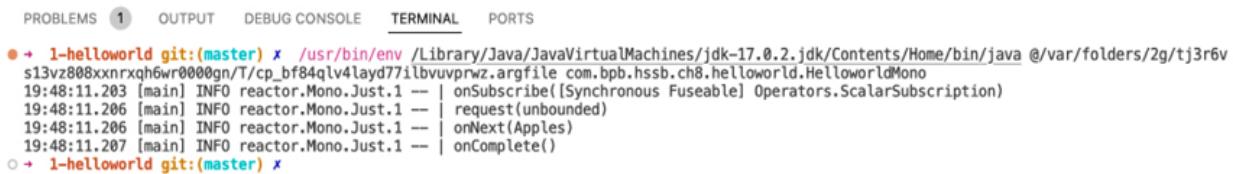
In *line 15*, we use the `take` function to pick the first three elements from the **frutFlux**, which results in a new Flux with three elements, and then subscribe with the same lambda function that prints each item to the console. If you are familiar with Java Streams, you can find several similarities in the reactive programming model with the Java Streams processing model.

Next, let us try to write a simple example to see how Mono works. Create a Java class called **HelloworldMono.java** with the following code:

```
1. package com.bpb.hssb.ch8.helloworld;
2.
3. import reactor.core.publisher.Mono;
4.
5. public class HelloworldMono {
6.
7.     public static void main(String[] args) {
8.
9.         Mono<String> frutMono = Mono.just("Apples");
10.
11.        frutMono.log().subscribe();
12.
13.        frutMono.subscribe(frut -> System.out.println(frut));
14.    }
15. }
```

In *line 9*, in contrast to the previous example, we define a reactive stream of type Mono with only one element using the **just** method.

In *line 11*, we log all events in Mono and then subscribe to them. The **subscribe** method without arguments means it will consume the items but not do anything with them. Executing this line prints all the events in the console, as shown in the following figure:



The figure shows a terminal window with the following content:

```
PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL PORTS
● + 1-helloworld git:(master) ✘ /usr/bin/env /Library/Java/JavaVirtualMachines/jdk-17.0.2.jdk/Contents/Home/bin/java @var/folders/2g/tj3r6v
s13vz800xxnrqhq6wr0000gn/T/cp_bf84qlv4lloyd77ilbvvprwz.argfile com.bpb.hssb.ch8.helloworld.HelloworldMono
19:48:11.203 [main] INFO reactor.Mono.Just.1 --- | onSubscribe([Synchronous Fuseable] Operators.ScalarSubscription)
19:48:11.206 [main] INFO reactor.Mono.Just.1 --- | request(unbounded)
19:48:11.206 [main] INFO reactor.Mono.Just.1 --- | onNext(Apples)
19:48:11.207 [main] INFO reactor.Mono.Just.1 --- | onComplete()
○ + 1-helloworld git:(master) ✘
```

**Figure 8.4:** Logs showing the events that occurred in a Flux

Then in *line 13*, we subscribe to the Mono again, this time with a **lambda**

function that prints each item to the console.

At this point, you have gotten hands-on with the basics of reactive programming in Java. Now, let us try to implement a sample use case for the BookClub project that we have been building throughout this book. Assume we want to implement a comment feature for the BookClub and anticipate massive data volume through this service and qualify to be implemented using the React programming model. Using the sample we created in the previous section as a starting point, go ahead and add the following dependencies to your project:

```
1. <dependency>
2.   <groupId>org.springframework.boot</groupId>
3.   <artifactId>spring-boot-starter-data-r2dbc</artifactId>
4. </dependency>
5. <dependency>
6.   <groupId>io.r2dbc</groupId>
7.   <artifactId>r2dbc-h2</artifactId>
8.   <scope>runtime</scope>
9. </dependency>
10. <dependency>
11.   <groupId>com.h2database</groupId>
12.   <artifactId>h2</artifactId>
13.   <scope>runtime</scope>
14. </dependency>
15. <dependency>
16.   <groupId>org.projectlombok</groupId>
17.   <artifactId>lombok</artifactId>
18.   <optional>true</optional>
19. </dependency>
```

These dependencies collectively set up a Spring Boot application with reactive database access using R2DBC and H2 databases. The **spring-boot-starter-data-r2dbc** dependency provides Spring Data R2DBC support, which offers a reactive programming model for database operations. The **r2dbc-h2** dependency includes the R2DBC driver for H2, allowing reactive database connectivity to H2 databases. Finally, the **h2** dependency includes the H2 database itself, which we use for the development and testing of the service. Both **r2dbc-h2** and **h2** are set to runtime scope, meaning they are only needed when the application is running, not during compilation.

Now, let us define a domain object called **Comment.java** using the following code:

```
1. package com.bpb.hssb.ch8.bookclub.domain;
2.
3. import java.time.LocalDateTime;
4.
5. import org.springframework.data.annotation.Id;
6.
7. import lombok.AllArgsConstructor;
8. import lombok.Builder;
9. import lombok.Getter;
10. import lombok.NoArgsConstructor;
11. import lombok.Setter;
12. import lombok.ToString;
13.
14. @Getter
15. @Setter
16. @AllArgsConstructor
17. @NoArgsConstructor
18. @Builder
19. @ToString
20. public class Comment {
21.
22.     @Id
23.     private Integer id;
24.     private int bookId;
25.     private String username;
26.     private String description;
27.     private LocalDateTime createdAt;
28.
29. }
```

30.

Next, we can define **CommentRepository.java** with the following code:

```
1. package com.bpb.hssb.ch8.bookclub.repository;
2.
3. import org.springframework.data.domain.Sort;
4. import org.springframework.data.repository.
   reactive.ReactiveCrudRepository;
```

```
5.
6. import com.bpb.hssb.ch8.bookclub.domain.Comment;
7.
8. import reactor.core.publisher.Flux;
9.
10. public interface CommentRepository extends
11.     ReactiveCrudRepository<Comment, Integer> {
12.     Flux<Comment> findAll(Sort sort);
13.
14. }
```

The above code is almost identical to the repository classes that we have defined in the previous chapters except for a couple of noticeable differences: the above interface extended from **ReactiveCrudRepository**, which is an interface provided by Spring Data that extends the reactive repository support for CRUD operations. Here are the methods defined in this interface.

- **save(S entity): Mono<S>**
- **saveAll(Iterable<S> entities): Flux<S>**
- **findById(ID id): Mono<T>**
- **findAll(): Flux<T>**
- **deleteById(ID id): Mono<Void>**
- **delete(T entity): Mono<Void>**
- **deleteAll(): Mono<Void>**

As you can see, these methods are similar to the methods that we have seen in the other responsory interface, but as the return types, **ReactiveCrudRepository** uses Flux or Mono streams. In addition to that, we have defined the **findAll** method that takes Sort options as a method parameter.

Now, we can define a **CommentService.java** interface as follows:

```
1. package com.bpb.hssb.ch8.bookclub.service;
2.
3. import com.bpb.hssb.ch8.bookclub.domain.Comment;
4.
5. import reactor.core.publisher.Flux;
6. import reactor.core.publisher.Mono;
```

```
7.
8. public interface CommentService {
9.
10. Flux<Comment> findRecentComments();
11.
12. Mono<Comment> createComment(Comment comment);
13.
14. Mono<Comment> findById(int id);
15.
16. Mono<Void> deleteComment(Comment comment);
17.
18. }
19.
```

We can also define the **CommentServiceImpl.java** implementation class with the following code. In fact, we do not need this service layer in this sample, but to follow the convention we used in this book, we can keep the service layer:

```
1. package com.bpb.hssb.ch8.bookclub.service;
2.
3. import org.springframework.data.domain.Sort;
4.
5. import com.bpb.hssb.ch8.bookclub.domain.Comment;
6. import com.bpb.hssb.ch8.bookclub.repository.CommentRepository;
7.
8. import reactor.core.publisher.Flux;
9. import reactor.core.publisher.Mono;
10.
11. public class CommentServiceImpl implements CommentService {
12.
13.     private CommentRepository commentRepository;
14.
15.     public CommentServiceImpl(CommentRepository commentRepository) {
16.         this.commentRepository = commentRepository;
17.     }
18.
19.     @Override
20.     public Flux<Comment> findRecentComments() {
21.         return commentRepository.findAll(Sort.by
22.             (Sort.Direction.DESC, "created_at")));
22.     }
23. }
```

```

23.
24. @Override
25. public Mono<Comment> createComment(Comment comment) {
26.     return commentRepository.save(comment);
27. }
28.
29. @Override
30. public Mono<Comment> findById(int id) {
31.     return commentRepository.findById(id);
32. }
33.
34. @Override
35. public Mono<Void> deleteComment(Comment comment) {
36.     return commentRepository.delete(comment);
37. }
38.

```

At this point, we can start implementing the **CommentController.java** class. Let us start with the following minimal code and gradually improve it:

```

1. package com.bpb.hssb.ch8.bookclub.controller;
2.
3. import org.springframework.web.bind.annotation.RequestMapping;
4. import org.springframework.web.bind.annotation.RestController;
5.
6. import com.bpb.hssb.ch8.bookclub.service.CommentService;
7.
8. import reactor.core.publisher.Flux;
9. import reactor.core.publisher.Mono;
10.
11. @RestController
12. @RequestMapping("/api/comments")
13. public class CommentController {
14.
15.     private CommentService commentService;
16.
17.     public CommentController(CommentService commentService) {
18.
19.         this.commentService = commentService;
20.     }
21.

```

```
22. }
```

The code is identical to the other RESTful controllers that we developed. The **CommentService** is injected into the **CommentController** using the constructor injection. Now, let us start adding our first handler method by adding the **findRecentComments** method given in the following code segment:

```
1. @GetMapping
2. public Flux<Comment> findRecentComments() {
3.
4.     return commentService.findRecentComments()
5.         .take(10)
6.         .switchIfEmpty(Flux.empty());
7. }
```

The code defines a GET endpoint in a Spring WebFlux controller that retrieves recent comments. The method **findRecentComments** returns a **Flux<Comment>**, which is a reactive stream of **Comment** objects. It calls a **commentService.findRecentComments()** method, used to fetch the comments from a database. The **take(10)** operator is then applied to limit the result to the first ten comments. Finally, **switchIfEmpty(Flux.empty())** is used to handle cases where no comments are found, returning an empty Flux instead of a null. This ensures that the endpoint always returns a Flux, even if it is empty.

Now, we can define **findById** method using the following code:

```
1. @GetMapping("/{id}")
2. public Mono<ResponseEntity<Comment>> findById(@PathVariable int id) {
3.
4.     return commentService.findById(id)
5.         .map(comment -> ResponseEntity.ok(comment))
6.         .defaultIfEmpty(ResponseEntity.notFound().build());
7. }
```

In the above code, the method **findById()** takes an id parameter from the URL path using **@PathVariable** and returns a **Mono<ResponseEntity<Comment>>**. It calls **commentService.findById(id)** to fetch the comment, then uses **map()** to wrap the found comment in a **ResponseType** with an **OK** status if it exists. The **defaultIfEmpty()** operator handles cases where no comment is found,

returning a **404 Not Found** response. This approach ensures that the endpoint always returns a proper HTTP response, either with the found comment or a not found status.

In reactive programming, the **map** method is a fundamental operator used for transforming data streams. It applies a specified function to each element emitted by a source stream, creating a new stream with the transformed elements while preserving the original stream's integrity. This one-to-one transformation maintains the order of elements and operates in a non-blocking manner, making it ideal for asynchronous data processing. A map is versatile and used for various tasks such as data conversion, information extraction, calculations, and formatting. It can be chained with other reactive operators to build complex data processing pipelines, respecting backpressure to ensure efficient data flow. In our example, the map method is used to create a stream of **ResponseType** objects from the stream of **Comment** objects.

Let us also create the **createComment** method using the following code:

```
1. @PostMapping
2. public Mono<ResponseType<Comment>> createComment(@RequestBody Comment com
   ment) {
3.     return commentService.createComment(comment)
4.         .map(savedComment -> ResponseEntity
5.             .created(URI.create("/api/comments/" + savedComment.getId()))
6.             .body(savedComment))
7.         .onErrorResume(e -> Mono.just(ResponseEntity.badRequest().build()));
8. }
```

The above code snippet defines a POST that accepts a **Comment** object as the request body and returns a **Mono<ResponseType<Comment>>**. It utilizes a **commentService** to persist the new comment in the database and then transforms the result using the map operator again. This transformation creates a **ResponseType** with a *Created* (201) status, sets the Location header to the newly created resource's URI, and includes the saved comment in the response body. Error handling is implemented via **onErrorResume**, which gracefully manages any exceptions by returning a *Bad Request* (400) response.

Finally, we can complete the controller by adding the **deleteComment** method:

```
1.
```

```

1. @DeleteMapping("/{id}")
2. public Mono<ResponseEntity<Comment>> deleteComment(@PathVariable int id) {
3.
4.     return commentService.findById(id).flatMap(
5.         comment -> commentService.deleteComment(comment).then(Mono.just(ResponseEntity.ok(comment))))
6.         .defaultIfEmpty(ResponseEntity.notFound().build());
7.     }

```

This code is similar to the way we defined the previous methods, deleted a **Comment** object from the database, and returned Mono with the deleted element.

Before we run our application, we need to define the Spring configuration using the following code:

```

1. package com.bpb.hssb.ch8.bookclub.config;
2.
3. import org.springframework.context.annotation.Bean;
4. import org.springframework.context.annotation.Configuration;
5.
6. import com.bpb.hssb.ch8.bookclub.repository.CommentRepository;
7. import com.bpb.hssb.ch8.bookclub.service.CommentService;
8. import com.bpb.hssb.ch8.bookclub.service.CommentServiceImpl;
9.
10. @Configuration
11. public class BookclubConfiguration {
12.
13.     @Bean
14.     public CommentService commentService
15.         (CommentRepository commentRepository) {
16.         return new CommentServiceImpl(commentRepository);
17.     }
18. }
19.

```

Let us also create the database schema by adding the following schema into the **src/main/resources/schema.sql** location:

```

1. DROP TABLE COMMENT IF EXISTS;
2.
3. CREATE TABLE COMMENT (

```

```
4. id INT AUTO_INCREMENT PRIMARY KEY,  
5. BOOK_ID INT NOT NULL,  
6. USERNAME VARCHAR(100) NOT NULL,  
7. DESCRIPTION VARCHAR(100) NOT NULL,  
8. CREATED_AT TIMESTAMP DEFAULT CURRENT_TIMESTAMP  
9. );
```

We can add some sample data to the database by adding the following data file to the src/main/resources/data.sql location:

1. `INSERT INTO COMMENT (BOOK_ID, USERNAME, DESCRIPTION, CREATED_AT) VALUES (1,'John Doe','This book was a fantastic read','2024-09-02 10:30:00');`
2. `INSERT INTO COMMENT (BOOK_ID, USERNAME, DESCRIPTION, CREATED_AT) VALUES (2,'John Doe','This book is a must read','2024-09-02 10:40:00');`
3. `INSERT INTO COMMENT (BOOK_ID, USERNAME, DESCRIPTION, CREATED_AT) VALUES (3,'John Doe','I like this book','2024-09-02 10:50:00');`
4. `INSERT INTO COMMENT (BOOK_ID, USERNAME, DESCRIPTION, CREATED_AT) VALUES (4,'John Doe','Very helpful book','2024-09-02 11:30:00');`
5. `INSERT INTO COMMENT (BOOK_ID, USERNAME, DESCRIPTION, CREATED_AT) VALUES (1,'Jane Smith','Best book for the subject','2024-09-04 10:30:00');`
6. `INSERT INTO COMMENT (BOOK_ID, USERNAME, DESCRIPTION, CREATED_AT) VALUES (2,'Jane Smith','Best book for the subject','2024-09-04 11:30:00');`
7. `INSERT INTO COMMENT (BOOK_ID, USERNAME, DESCRIPTION, CREATED_AT) VALUES (3,'Jane Smith','Very hard to read','2024-09-04 12:30:00');`
8. `INSERT INTO COMMENT (BOOK_ID, USERNAME, DESCRIPTION, CREATED_AT) VALUES (4,'Jane Smith','I like this book','2024-09-04 12:40:00');`
9. `INSERT INTO COMMENT (BOOK_ID, USERNAME, DESCRIPTION, CREATED_AT) VALUES (1,'Emily Brown','Everyone should read this book','2024-09-06 10:30:00');`
10. `INSERT INTO COMMENT (BOOK_ID, USERNAME, DESCRIPTION, CREATED_AT) VALUES (2,'Emily Brown','This book was a fantastic read','2024-09-06 10:40:00');`
11. `INSERT INTO COMMENT (BOOK_ID, USERNAME, DESCRIPTION, CREATED_AT) VALUES (3,'Emily Brown','This book was a fantastic read','2024-09-06 10:50:00');`
12. `INSERT INTO COMMENT (BOOK_ID, USERNAME, DESCRIPTION, CREATED_AT) VALUES (1,'Alex Wilson','Very easy to read','2024-09-12 10:30:00');`
- 13.

Now, you can run the service and test the above endpoints using an HTTP testing tool like Postman or cURL.

Spring WebFlux supports two programming models. We have used the annotated controllers model, which is very close to the programming model that we used to develop RESTful services and is easy to understand as well.

We will look into the other programming model, functional endpoints, later in this chapter.

## Writing test for reactive services

In this section, let us look at how to write test cases for reactive services. First, add the following dependency to your project:

1. <dependency>
2. <groupId>io.projectreactor</groupId>
3. <artifactId>reactor-test</artifactId>
4. <scope>test</scope>
5. </dependency>

Next, add the following **test** class with the minimal code:

```
1. package com.bpb.hssb.ch8.bookclub.controller;
2.
3. import org.springframework.beans.factory.annotation.Autowired;
4. import org.springframework.boot.test.autoconfigure.web.reactive.WebFluxTest;
5. import org.springframework.boot.test.mock.mockito.MockBean;
6. import org.springframework.test.web.reactive.server.WebTestClient;
7.
8. import com.bpb.hssb.ch8.bookclub.service.CommentService;
9.
10. @WebFluxTest(CommentController.class)
11. public class BookControllerTest {
12.
13.     @Autowired
14.     private WebTestClient webTestClient;
15.
16.     @MockBean
17.     private CommentService commentService;
18. }
```

In the above code, **@WebFluxTest** annotation is used to test Spring WebFlux controllers. It disables full auto-configuration and only applies configuration relevant to WebFlux tests. It is specifically configured to test **CommentController.class**.

Then, we have autowired WebTestClient, which is a non-blocking, reactive client for testing WebFlux server endpoints. It is automatically configured by the **@WebFluxTest** annotation. We have also added a **MockBean** to create a

mock of the **CommentService**. In tests, this mock is used to simulate service behavior without actually calling the real service methods.

Now, we can define our first test case with the following code:

```
1.  @Test
2.  void testGetComment() throws Exception {
3.
4.      Comment comment = Comment.builder()
5.          .bookId(1)
6.          .username("Sagara")
7.          .description("Very good book")
8.          .build();
9.
10.     when(commentService.findById(100)).thenReturn(Mono.just(comment));
11.
12.     webTestClient.get()
13.         .uri("/api/comments/100")
14.         .exchange().expectStatus()
15.         .isOk()
16.         .expectBody(Comment.class)
17.         .value(cm -> {
18.             assertNotNull(cm);
19.             assertEquals(1, cm.getBookId());
20.             assertEquals("Sagara", cm.
getUsername());
21.             assertEquals("Very good book", cm.getDescription());
22.         });
23.     }
```

In the above code, first, we configure the mock service to return the **comment** object when the **findById** method is called with **100** as the parameter value. Then, we use **webTestClient** to call the get comment by ID endpoint and verify using the assert methods.

Let us also write a test case for the create comment endpoint. Add the following test case to the above class:

```
1.  @Test
2.  void testCreateComment() throws Exception {
3.
4.      Comment newComment = Comment.builder()
```

```

5.     .bookId(1)
6.     .username("Sagara")
7.     .description("Excellent read!")
8.     .build();
9.

10.    Comment savedComment = Comment.builder()
11.        .id(1)
12.        .bookId(1)
13.        .username("Sagara")
14.        .description("Excellent read!")
15.        .build();
16.
17.
18.    when(commentService.createComment(any(Comment.class))).  

thenReturn(Mono.just(savedComment));
19.
20.    webTestClient.post()
21.        .uri("/api/comments")
22.        .contentType(MediaType.APPLICATION_JSON)
23.        .bodyValue(newComment)
24.        .exchange()
25.        .expectStatus().isCreated()
26.        .expectBody(Comment.class)
27.        .value(comment -> {
28.            assertNotNull(comment);
29.            assertEquals(1, comment.getBookId());
30.            assertEquals("Sagara", comment.getUsername());
31.            assertEquals("Excellent read!",
comment.getDescription());
32.        });
33.    }

```

Similar to the previous test case, we configured the mock service first. Then, use the **webTestClient** to create a comment endpoint and verify the results.

## Functional endpoint model

Previously, we developed a service using the annotated controller model, which is similar to the RESTful services we have worked on before. In this

section, we will rewrite the same service using the functional endpoint model to explore an alternative approach.

Let us first replace the **CommentController** of the previous example with **CommentHandler** using the following code:

```
1. package com.bpb.hssb.ch8.bookclub.handler;
2.
3. import com.bpb.hssb.ch8.bookclub.service.CommentService;
4.
5. public class CommentHandler {
6.
7.     private final CommentService commentService;
8.
9.     public CommentHandler(CommentService commentService) {
10.         this.commentService = commentService;
11.     }
12.
13. }
```

As you can easily understand, in the above code, we have injected **CommentService** into **CommentHandler** using constructor injection; other than that, this is just a simple Java bean.

Now, we can add our first handler method to implement find recent comments:

```
1. public Mono<ServerResponse> findRecentComments(ServerRequest request) {
2.     return ServerResponse.ok()
3.         .body(commentService.findRecentComments().take(10),
4.               Comment.class);
5. }
```

The above method takes a **ServerRequest** as input and returns a **Mono<ServerResponse>** as the return. It constructs the response by chaining methods, starting with **ServerResponse.ok()** to set a **200 OK** status, then using **.body()** to populate the response content. The body is filled with data from **commentService.findRecentComments()**, which returns a Flux of **Comment** objects. The **.take(10)** operator is applied to limit the result to only the ten most recent comments. The **Comment.class** parameter in the **body** method specifies the type of elements in the response.

Let us add a method to find comments by ID as well:

```
1. public Mono<ServerResponse> findById(ServerRequest request) {
```

```
2.     int id = Integer.parseInt(request.pathVariable("id"));
3.     return commentService.findById(id)
4.         .flatMap(comment ->
5.             ServerResponse.ok().bodyValue(comment))
6.         .switchIfEmpty(ServerResponse.notFound().build());
7. }
```

The above **findById** method takes a **ServerRequest** as input and returns a **Mono<ServerResponse>**. It first extracts the comment ID from the request's path variable, converting it to an integer. The handler then calls **commentService.findById(id)** to fetch the comment asynchronously. Using **flatMap**, it transforms the result into a **ServerResponse**: if a comment is found, it creates an **OK** response with the comment as the body. The **switchIfEmpty** operator handles the case where no comment is found, returning a **404 Not Found** response.

The **flatMap** is a powerful operator used for asynchronous, one-to-many transformations of data streams. It is particularly useful when dealing with operations that return nested Publishers. The **flatMap** operator applies a function to each element emitted by the source stream, where this function returns another **Publisher**. **flatMap** then flattens these inner Publishers into a single output stream, effectively merging the results. This makes it ideal for scenarios involving asynchronous operations, such as database queries or network calls, where each input element might produce multiple output elements or require further asynchronous processing. Unlike **map**, which maintains a one-to-one relationship between input and output elements, **flatMap** can produce a variable number of output elements for each input, allowing for more complex transformations and compositions of asynchronous operations.

Next, we can add the following method to create a comment:

```
1. public Mono<ServerResponse> createComment(ServerRequest request) {
2.     return request.bodyToMono(Comment.class)
3.         .flatMap(commentService::createComment)
4.         .flatMap(savedComment ->
5.             ServerResponse.created(URI.create
6.                 ("/api/comments/" + savedComment.getId())))
7.             .bodyValue(savedComment))
8.         .onErrorResume(e -> ServerResponse.
9.             badRequest().build());
10. }
```

The above method takes a **ServerRequest** as the input and returns a **Mono<ServerResponse>**. It begins by extracting the comment data from the request body using **bodyToMono(Comment.class)**. The extracted comment is then passed to the **commentService**'s **createComment** method using **flatMap**, which handles the asynchronous operation of saving the comment. Upon successful creation, another **flatMap** is used to construct the response. It creates a **201 Created** response with the **Location** header set to the URI of the new comment and includes the saved comment in the response body. The **onErrorResume** operator is used for error handling, returning a **400 Bad Request** response if any error occurs during the process.

Finally, we can add a method to remove a comment:

```
1. public Mono<ServerResponse> deleteComment
   (ServerRequest request) {
2.     int id = Integer.parseInt(request.pathVariable("id"));
3.     return commentService.findById(id)
4.         .flatMap(comment -> commentService.deleteComment(comment))
5.             .then(ServerResponse.ok().bodyValue(comment)))
6.         .switchIfEmpty(ServerResponse.notFound().build());
7. }
```

So far, we have added **handler** methods, but it is still just a bean. In order to use the handler for server requests, we need to register the handler with each route. This can be done by adding the following configuration to the **BookclubConfiguration** class:

```
1.
2.     @Bean
3.     public CommentHandler commentHandler(CommentService service){
4.         return new CommentHandler(service);
5.     }
6.
7.     @Bean
8.     public RouterFunction<ServerResponse>
9.         commentRoutes(CommentHandler handler) {
10.         return route()
11.             .path("/api/comments", builder -> builder
12.                 .GET("", handler::findRecentComments)
13.                 .POST("", handler::createComment)
14.                 .GET("/{id}", handler::findById)
15.                 .DELETE("/{id}", handler::deleteComment))
```

```
14.     .build();
15. }
```

In the above code, we have configured a **RouterFunction** bean and registered each **handler** method with the correct request path.

If you run and test this service, you can see identical results as the previous example. It is up to you to pick one programming model between annotated controllers and functional preferences based on your preferences.

## Building reactive clients

So far, we have explored how to develop reactive services using two main programming models. This section will focus on developing a reactive client for the previously created services.

Start by creating a new project using Spring Initializr. Be sure to add WebFlux as a dependency. Next, create a domain class called **Comment** with the following code:

```
1. package com.bpb.hssb.ch8.bookclub.client;
2.
3. import java.time.LocalDateTime;
4.
5. import lombok.AllArgsConstructor;
6. import lombok.Builder;
7. import lombok.Getter;
8. import lombok.NoArgsConstructor;
9. import lombok.Setter;
10. import lombok.ToString;
11.
12. @Getter
13. @Setter
14. @AllArgsConstructor
15. @NoArgsConstructor
16. @Builder
17. @ToString
18. public class Comment {
19.
20.     private Integer id;
21.     private int bookId;
```

```
22. private String username;  
23. private String description;  
24. private LocalDateTime createdAt;  
25.  
26. }  
27.
```

Next, create a Java class called **BookClubReactiveClient** with the following code:

```
1. package com.bpb.hssb.ch8.bookclub.client;  
2.  
3. import org.springframework.beans.factory.annotation.Autowired;  
4. import org.springframework.boot.CommandLineRunner;  
5. import org.springframework.boot.SpringApplication;  
6. import org.springframework.boot.autoconfigure.SpringBootApplication;  
7.  
8. import java.util.List;  
9.  
10. import org.slf4j.Logger;  
11. import org.slf4j.LoggerFactory;  
12. import org.springframework.web.reactive.function.client.WebClient;  
13.  
14. @SpringBootApplication  
15. public class BookClubReactiveClient implements CommandLineRunner {  
16.  
17. private static final Logger logger =  
    LoggerFactory.getLogger(BookClubReactiveClient.class);  
18.  
19. @Autowired  
20. private WebClient webClient;  
21. public static void main(String[] args) {  
22.  
23. SpringApplication.run(BookClubReactiveClient.class, args);  
24. }  
25.  
26. @Override  
27. public void run(String... args) throws Exception {  
28.  
29. List<Comment> comments = webClient.get()  
30. .uri("/comments")  
31. .retrieve()  
32. .bodyToFlux(Comment.class)
```

```
33.     .collectList()
34.     .block();
35.
36.     logger.info("Recent Comments : {}", comments);
37.
38. }
39.
40. }
```

In the above code, we have autowired WebClient as a dependency, which means we have to configure and provide it through our Spring configuration. The WebClient is configured to send a request to the **/comments** endpoint using HTTP GET. The retrieve method initiates the request, and **bodyToFlux(Comment.class)** is used to deserialize the response body into a Flux of **Comment** objects. The Flux is then converted to a **List** using **collectList**. The **block** method at the end makes this reactive operation blocking, waiting for the entire operation to complete and returning the result as a **List<Comment>**; without the **block** method, the client would not wait for results. Finally, the retrieved comments are logged using a logger, providing visibility into the fetched data.

The **block** method is a terminal operator that serves as a bridge between reactive and imperative programming styles by converting an asynchronous reactive stream into a synchronous operation. When called on a Mono or Flux, the **block** subscribes to the stream and waits for it to complete, returning the emitted values or throwing an exception if the stream errors. While it provides a straightforward way to extract results from reactive streams, its use is generally discouraged in fully reactive applications as it negates the benefits of non-blocking execution. The **block** method is primarily useful in testing scenarios or when integrating reactive code with non-reactive parts of an application like in the above code.

The **collectList** method transforms a Flux into a Mono containing a **List** of all the elements emitted by the original Flux. This method is particularly useful when you need to aggregate all elements from a reactive stream into a single collection. It subscribes to the source Flux, collects all emitted items into a **List**, and emits this **List** as a single item when the source Flux completes. The **collectList()** operation is non-blocking and lazy, meaning it will not start collecting items until the resulting Mono is subscribed to. This

makes it efficient in handling large or infinite streams, as it does not immediately materialize the entire stream into memory. However, it is important to note that while **collectList()** itself is non-blocking, it does introduce a form of backpressure, as it needs to collect all elements before emitting the list. This method is particularly useful in scenarios where you need to process or analyze the entire dataset as a whole rather than handling elements individually in a streaming fashion.

As the final step, we can add client configuration using the following code:

```
1. package com.bpb.hssb.ch8.bookclub.client;
2.
3. import org.springframework.beans.factory.annotation.Value;
4. import org.springframework.context.annotation.Bean;
5. import org.springframework.context.annotation.Configuration;
6. import org.springframework.web.reactive.function.client.WebClient;
7.
8. @Configuration
9. public class BookclubConfiguration {
10.
11.     @Value("${bookclub.api.baseurl}")
12.     private String baseurl;
13.
14.     @Bean
15.     public WebClient webClient() {
16.         return WebClient.create(baseurl);
17.     }
18.
19. }
20.
```

In the above code, the **@Value** annotation is used to inject the base URL of the API from a configuration property named **bookclub.api.baseurl** and used to create a **WebClient** the **WebClient.create()** factory method.

Before testing this client application, make sure to run one of the services that we created in the previous sections.

## Conclusion

By the end of this chapter, you have gained a clear understanding of reactive

programming and how it fundamentally differs from imperative programming. You have explored how to build reactive services using Spring WebFlux, written effective test cases for these services, explored the functional endpoint model, and developed client applications that leverage the reactive paradigm. With these skills, you are now well-equipped to apply reactive programming in your application development.

In the next chapter, we will learn how to use messaging protocols such as JMS and MQTT with Spring Boot.

## **Join our book's Discord space**

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

[https://discord\(bpbonline\).com](https://discord(bpbonline).com)



*OceanofPDF.com*

# CHAPTER 9

# Working with Spring Messaging

## Introduction

This chapter introduces messaging and explores how it can be effectively applied in real-world scenarios. We begin by discussing the key messaging patterns, focusing on queue-based and topic-based messaging and their respective use cases. From there, we dive into **Java Message Service (JMS)**, explaining how messages are transmitted and received using Spring Framework with ActiveMQ as the message broker. Finally, the chapter introduces the **Advanced Message Queuing Protocol (AMQP)**, comparing its features with JMS and demonstrating how to implement AMQP-based messaging with Spring and RabbitMQ.

## Structure

The chapter covers the following topics:

- Introduction to messaging
- Spring Messaging with ActiveMQ
- Spring Messaging with RabbitMQ

## Objectives

By the end of this chapter, you will have a solid understanding of messaging, including when and where it is most effective. You will also grasp key

messaging patterns, such as queue-based and topic-based communication, along with their distinct use cases. Additionally, you will learn how to send and receive JMS messages using Spring with ActiveMQ Artemis as the broker and how to work with AMQP messages using RabbitMQ, gaining hands-on experience with both technologies.

## Introduction to messaging

We have come a long way since *Alexander Graham Bell* invented the telephone over a century ago, passing several milestones such as the mobile phone, satellite-based communication, and internet-based platforms, and this evolution will continue. Throughout these developments, two primary ways of communication remain unchanged: calling and messaging.

Calling has been around since the beginning and remains an effective way to communicate, especially when real-time interaction is needed. If you need to discuss something urgently with someone remotely, calling is the best approach because it is inherently bidirectional and immediate. This is why emergency services rely on short-code numbers for quick, real-time response. However, calling comes with a limitation: the recipient must be available to answer in real-time, and various factors, like network issues or the recipient's phone being off, can hinder communication.

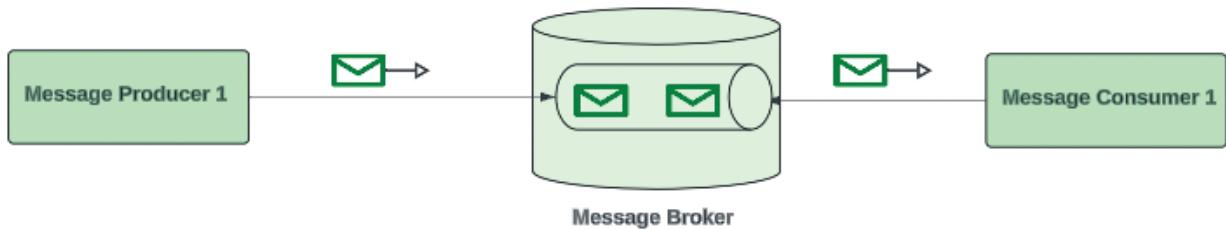
This scenario closely mirrors the default communication model among software systems: the real-time peer-to-peer request-response model. Like phone calls, both systems must be connected over a network (like the Internet or a LAN) for successful communication. This model is highly effective for many use cases, such as online pizza orders, where real-time confirmation and actions, like payment processing and delivery scheduling, are required. However, the downside is that if communication is not real-time, reliable, or efficient, it could lead to business failures, making this model unsuitable for certain cases where reliability is critical.

On the other hand, **messaging** offers distinct advantages. Often, we prefer messaging over calling because the recipient does not need to be available in real-time. You simply send a message, and the recipient receives it when they can. This makes messaging more **reliable** and **fault tolerant**. For example, if a mobile tower goes offline, the message is not lost, it is stored and delivered when the tower is back online. Similarly, if the recipient's

phone is off, they will still receive the message once it is turned on. Another advantage is that messaging is unidirectional, meaning the recipient is not forced to respond immediately. They can choose when to reply, offering flexibility.

This is analogous to messaging in software systems, which removes the need for recipient systems to handle requests and responses in real-time. Additionally, messaging provides increased reliability and fault tolerance, making it a better option for certain types of transactions. This pattern is classified as an asynchronous communication mechanism among software systems.

In software systems, messaging often relies on an intermediary called a broker. Much like databases, there are both proprietary and open-source message brokers available to use. As shown in the diagram below, the communication begins when the sender, sometimes called the producer, transmits a message to the broker, including the intended recipient's details, similar to how we write an address on a postcard before dropping it into a mailbox. The representation of the same is as follows:

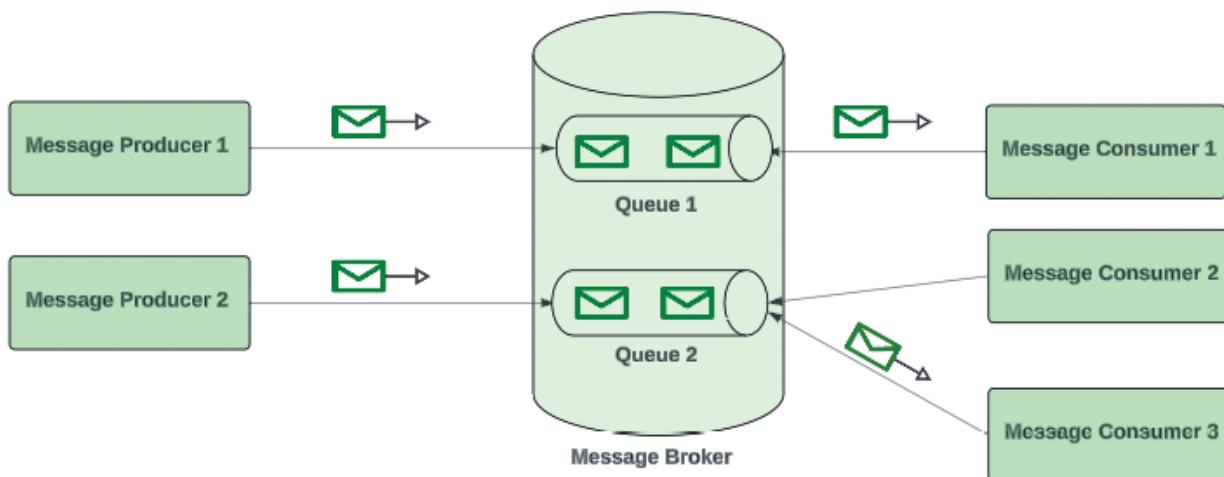


*Figure 9.1: Message broker-based communication*

Once the broker acknowledges receipt of the message, the sender can safely terminate the communication link without worrying about further delivery. The broker is now responsible for securely storing the message and ensuring it is reliable, much like the postal system handles letters. The recipient, sometimes called the message consumer, can connect to the broker to retrieve them and, if necessary, respond using the same mechanism via the broker as the intermediary.

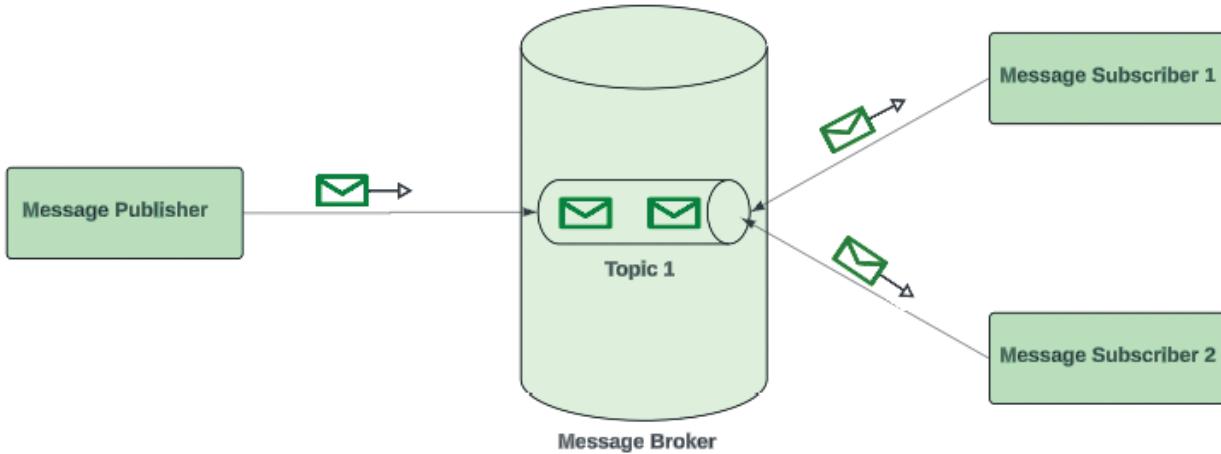
In messaging systems, two primary approaches are commonly used: queue-based and topic-based messaging. Queue-based messaging, often referred to as point-to-point messaging, typically involves structures known as queues or message queues. These act as ordered collections of messages, where

messages are enqueued and dequeued in a **first-in, first-out (FIFO)** order. Messages are sent to a specific queue, and each message is consumed by only one consumer, even if multiple consumers are connected to the queue. A queue-based system usually consists of three key concepts: the queue itself, created in the message broker to store the messages; the producer, which is the application responsible for sending messages to the queue; and the consumer, which retrieves and processes those messages from the queue. This model ensures that messages are handled reliably while allowing multiple producers and consumers to interact with the queue. The following figure depicts queue-based messaging:



*Figure 9.2: Queue-based messaging pattern*

In topic-based messaging, also known as publish-subscribe (pub/sub) messaging, messages are sent to entities called *topics* or *exchanges*. In this model, a single message can be delivered to multiple subscribers. Messages are published to a topic, and all subscribers to that topic receive copies of the same message, enabling multiple consumers to process the same data simultaneously. The key elements of topic-based messaging include the topic, which serves as the messaging endpoint where messages are published; the publisher, which is the application responsible for sending messages to the topic; the subscriber, which is the application that receives messages from the topic; and the subscription, a virtual queue that receives copies of the messages sent to the topic. This approach is especially useful for scaling applications to large numbers of recipients. The following figure depicts topic-based messaging:



*Figure 9.3: Topic-based messaging pattern*

## Spring Messaging with ActiveMQ

**Java Message Service (JMS)** is a Java API that provides a standardized interface for creating, sending, receiving, and reading messages between distributed software components, ensuring portability and interoperability across different systems. JMS specification was originally developed as part of the Java EE, but it can be used separately as well. JMS enables asynchronous and reliable communication between Java-based applications. It supports both queue-based and topic-based models, allowing for flexible and scalable communication patterns.

As we have explored the concept of messaging in detail, let us move on to building our first messaging application using JMS and Spring Boot. We can integrate a messaging use case into our BookClub sample application. Imagine that the BookClub plans to initiate an exchange program with other similar clubs, allowing members from different clubs to request book reservations. This service would also be available to the BookClub members, enabling a seamless flow of book exchanges across clubs.

There is an initial project available in the companion GitHub repository of this book to help you get started with this exercise. It includes the repository and service classes we have previously developed. Your first step will be to add the following dependencies to the project's POM file:

1. <dependency>
2. <groupId>org.springframework.boot</groupId>
3. <artifactId>spring-boot-starter-artemis</artifactId>

```
4. </dependency>
5. <dependency>
6.   <groupId>org.springframework.boot</groupId>
7.   <artifactId>spring-boot-starter-json</artifactId>
8. </dependency>
9. <dependency>
10.  <groupId>org.apache.activemq</groupId>
11.  <artifactId>artemis-server</artifactId>
12. </dependency>
13. <dependency>
14.  <groupId>org.apache.activemq</groupId>
15.  <artifactId>artemis-jakarta-server</artifactId>
16. <scope>runtime</scope>
17. </dependency>
```

By including these dependencies, we are setting up the project to use Apache ActiveMQ Artemis as an embedded JMS broker, along with the necessary Spring dependencies to support JMS messaging. The **spring-boot-starter-artemis** starter provides everything needed to connect to an existing Artemis instance and integrates Spring's JMS functionality. Then, the **spring-boot-starter-json** starter enables JSON processing capabilities, including serialization and deserialization, which we will use to handle message payloads in JSON format. We have also included a JakartaEE-compatible version of the Apache ActiveMQ Artemis embedded server as a runtime dependency. This eliminates the need to configure a separate message broker, making it easier to manage within the application.

Next, we will define a new domain class called **ReservationRequest** to represent book reservation requests using the following code:

```
1. package com.bpb.hssb.ch9.bookclub.domain;
2.
3. import jakarta.persistence.Entity;
4. import jakarta.persistence.GeneratedValue;
5. import jakarta.persistence.GenerationType;
6. import jakarta.persistence.Id;
7. import lombok.AllArgsConstructor;
8. import lombok.Builder;
9. import lombok.Getter;
10. import lombok.NoArgsConstructor;
11. import lombok.Setter;
```

```
12. import lombok.ToString;
13.
14. @Getter
15. @Setter
16. @AllArgsConstructor
17. @NoArgsConstructor
18. @Builder
19. @ToString
20. @Entity
21. public class ReservationRequest {
22.
23.     @Id
24.     @GeneratedValue(strategy = GenerationType.AUTO)
25.     private int reservationId;
26.     private String referenceId;
27.     private String club;
28.     private String book;
29.
30. }
```

We can also introduce a JPA repository for **ReservationRequest** using the following code:

```
1. package com.bpb.hssb.ch9.bookclub.repository;
2.
3. import org.springframework.data.repository.CrudRepository;
4.
5. import com.bpb.hssb.ch9.bookclub.domain.ReservationRequest;
6.
7. public interface ReservationRequestRepository extends
   CrudRepository<ReservationRequest, Integer> {
8.
9. }
```

Now, with the following code, we can define a service interface for reservation requests:

```
1. package com.bpb.hssb.ch9.bookclub.service;
2.
3. import com.bpb.hssb.ch9.bookclub.domain.ReservationRequest;
4.
5. public interface ReservationService {
6.
7.     public ReservationRequest createReservationRequest
```

```
    (ReservationRequest reservationRequest);
8.
9. }
```

In the above code, we have defined a method called **createReservationRequest** to accept a **ReservationRequest** as a parameter and rerun the updated **ReservationRequest** with the reservation. We can simply define the implementation of this interface as follows:

```
1. package com.bpb.hssb.ch9.bookclub.service;
2.
3. import com.bpb.hssb.ch9.bookclub.domain.ReservationRequest;
4. import com.bpb.hssb.ch9.bookclub.repository.
   ReservationRequestRepository;
5.
6. public class ReservationServiceImpl implements ReservationService {
7.
8.     private ReservationRequestRepository
   reservationRequestRepository;
9.
10.    public ReservationServiceImpl(ReservationRequestRepository
   reservationRequestRepository) {
11.        this.reservationRequestRepository =
   reservationRequestRepository;
12.    }
13.
14.    @Override
15.    public ReservationRequest createReservationRequest
   (ReservationRequest reservationRequest) {
16.        return reservationRequestRepository.save(reservationRequest);
17.    }
18.
19. }
```

The above service implementation expects a **ReservationRequestRepository**, and we have used constructor injection to supply this dependency.

So far, we have not encountered any JMS-related code. As the next step, let us create a receiver implementation that waits for JMS messages from the embedded message broker.

To do this, create a new class called **ReservationRequestReceiver.java** and use the following code:

```

1. package com.bpb.hssb.ch9.bookclub.receiver;
2.
3. import org.springframework.jms.annotation.JmsListener;
4.
5. import org.slf4j.Logger;
6. import org.slf4j.LoggerFactory;
7.
8. import com.bpb.hssb.ch9.bookclub.domain.ReservationRequest;
9. import com.bpb.hssb.ch9.bookclub.service.ReservationService;
10. import com.bpb.hssb.ch9.bookclub.service.ReservationServiceImpl;
11.
12. public class ReservationRequestReceiver {
13.
14.     private static final Logger logger =
15.         LoggerFactory.getLogger(ReservationServiceImpl.class);
16.
17.     private ReservationService reservationService;
18.
19.     public ReservationRequestReceiver
20.         (ReservationService reservationService) {
21.         this.reservationService = reservationService;
22.     }
23.
24.     @JmsListener(destination = "bookreservation",
25.         containerFactory = "myFactory")
26.     public void receiveMessage
27.         (ReservationRequest reservationRequest) {
28.
29.         ReservationRequest created = reservationService.
30.             createReservationRequest(reservationRequest);
31.         logger.info("ReservationRequest created : " + created);
32.     }
33. }
```

Aside from using the **@JmsListener** annotation in the **receiveMessage** method, there is hardly any visible JMS-related code in the above class. This demonstrates the simplicity of the Spring Framework, which allows developers to work with POJOs. The **receiveMessage** method behaves just like any other Java method—it takes a **ReservationRequest** object as a parameter and returns the persisted request.

The **@JmsListener** annotation streamlines the process of creating **message-driven POJOs** by marking methods to receive messages from specified

queues or topics. It abstracts away the complexity of setting up the JMS infrastructure by automatically configuring everything needed to listen for incoming messages and invoking the annotated method when a message arrives.

In our example, the `@JmsListener` annotation specifies the destination name as `bookreservation` and uses a custom `JmsListenerContainerFactory` named `myFactory`. This annotation is flexible, supporting different method signatures to access the raw message, extract the payload, or retrieve header information.

Finally, we will create a configuration class named `JMSConfiguration` using the following code:

```
1. package com.bpb.hssb.ch9.bookclub.config;
2.
3. import org.springframework.beans.factory.annotation.Value;
4. import org.springframework.boot.autoconfigure.jms.
   DefaultJmsListenerContainerFactoryConfigurer;
5. import org.springframework.boot.autoconfigure.jms.
   artemis.ArtemisConfigurationCustomizer;
6. import org.springframework.context.annotation.Bean;
7. import org.springframework.context.annotation.Configuration;
8. import org.springframework.jms.config.
   DefaultJmsListenerContainerFactory;
9. import org.springframework.jms.config.JmsListenerContainerFactory;
10. import org.springframework.jms.support.converter.
   MappingJackson2MessageConverter;
11. import org.springframework.jms.support.converter.MessageConverter;
12. import org.springframework.jms.support.converter.MessageType;
13.
14. import jakarta.jms.ConnectionFactory;
15.
16. @Configuration
17. public class JMSConfiguration {
18.
19.   @Value("${spring.artemis.embedded.port:61616}")
20.   private int artemisPort;
21.
22.   @Bean
23.   public ArtemisConfigurationCustomizer
```

```

    artemisConfigurationCustomizer() {
24.      return (org.apache.activemq.artemis.core.
    config.Configuration configuration) -> {
25.        try {
26.          configuration.addAcceptorConfiguration
    ("tcp", "tcp://0.0.0.0:" + artemisPort);
27.
28.        } catch (Exception e) {
29.          throw new RuntimeException
    ("Failed to add TCP acceptor to Artemis configuration", e);
30.        }
31.      };
32.    }
33.
34.    @Bean
35.    public JmsListenerContainerFactory<?> myFactory
    (ConnectionFactory connectionFactory,
36.      DefaultJmsListenerContainerFactoryConfigurer configurer) {
37.      DefaultJmsListenerContainerFactory factory =
    new DefaultJmsListenerContainerFactory();
38.      configurer.configure(factory, connectionFactory);
39.      return factory;
40.    }
41.
42.    @Bean
43.    public MessageConverter jacksonJmsMessageConverter() {
44.      MappingJackson2MessageConverter converter =
    new MappingJackson2MessageConverter();
45.      converter.setTargetType(MessageType.TEXT);
46.      converter.setTypeIdPropertyName("_type");
47.      return converter;
48.    }
49.  }

```

In the above code, we first define an **ArtemisConfigurationCustomizer** bean to modify the configuration of an embedded **Apache Artemis message broker**. This functional interface allows us to add custom configurations, specifically, a **TCP acceptor** that enables the broker to accept connections from all network interfaces (0.0.0.0) on a specified port (**artemisPort**). This setup allows external clients to connect to the embedded broker via TCP. If

the in-VM transport acceptor, which works only within the same runtime, was sufficient, we could omit this configuration, but the TCP acceptor makes the example more realistic.

Next, we define a bean that creates and configures a **JmsListenerContainerFactory**, which is crucial for setting up JMS message listeners in a Spring application. The method takes a **ConnectionFactory** and a **DefaultJmsListenerContainerFactoryConfigurer** as parameters, both of which are typically auto-configured by Spring Boot. It creates a new instance of **DefaultJmsListenerContainerFactory** and uses the provided configurator to apply default configurations, ensuring it is properly set up with the given connection factory. This approach allows customization of the JMS listener container factory while still leveraging Spring Boot's auto-configuration capabilities. The resulting factory can be used to create JMS listener containers to manage the lifecycle of message listeners and handle the reception of JMS messages. This factory is later referenced in the **ReservationRequestReceiver** class via the **@JmsListener** annotation, which enables message-driven POJOs to handle incoming messages.

The third bean creates a **Jackson-based message converter** for JMS. The **MappingJackson2MessageConverter** converts between **JMS messages and Java objects**, leveraging Jackson's JSON processing capabilities. The converter is configured to treat the **message payload as text (MessageType.TEXT)**, which is ideal for handling JSON content. Additionally, it sets the **\_type** property to facilitate the deserialization of polymorphic types, ensuring seamless conversion between **JSON and Java objects** during message transmission and reception.

At this point, you are ready to run your first JMS application. When the application runs, the embedded Artemis broker should start, and the JMS listener will wait for incoming messages. If everything is correctly configured, you should see logs indicating the broker's activity and message handling.

If you notice the default logging is too verbose, you can reduce it by adding the following line to the **application.properties** file:

```
1. logging.level.org.apache.activemq=ERROR
```

In the next section, we will create a JMS client-side application to test this

setup.

Head to Spring Initializr and generate a project named **bookclub-jms-client** by adding **spring-boot-starter-artemis**, **spring-boot-starter-json**, and **lombok** as dependencies. Ideally, the POM file of the generated project should contain the following dependencies:

1. <dependency>
2. <groupId>org.springframework.boot</groupId>
3. <artifactId>spring-boot-starter-artemis</artifactId>
4. </dependency>
5. <dependency>
6. <groupId>org.springframework.boot</groupId>
7. <artifactId>spring-boot-starter-json</artifactId>
8. </dependency>

As the next step, add the same **ReservationRequest** domain class to the client project to maintain consistency with the server-side application:

1. package com.bpb.hssb.ch9.bookclub.domain;
- 2.
3. `import lombok.AllArgsConstructor;`
4. `import lombok.Builder;`
5. `import lombok.Getter;`
6. `import lombok.NoArgsConstructor;`
7. `import lombok.Setter;`
8. `import lombok.ToString;`
- 9.
10. `@Getter`
11. `@Setter`
12. `@NoArgsConstructor`
13. `@AllArgsConstructor`
14. `@Builder`
15. `@ToString`
16. `public class ReservationRequest {`
- 17.
18.  `private String referenceId;`
19.  `private String club;`
20.  `private String book;`
- 21.
22. `}`

23.

In the previous sample, we created a JMS listener that waits for new JMS messages. In this client-side application, we will implement a JMS sender to send messages.

Start by creating a class named **ReservationRequestSender.java** with the following code:

```
1. package com.bpb.hssb.ch9.bookclub.client;
2.
3. import org.springframework.jms.core.JmsTemplate;
4.
5. import com.bpb.hssb.ch9.bookclub.domain.ReservationRequest;
6.
7. public class ReservationRequestSender {
8.
9.     private JmsTemplate jmsTemplate;
10.
11.    public ReservationRequestSender(JmsTemplate jmsTemplate) {
12.        this.jmsTemplate = jmsTemplate;
13.    }
14.
15.    public void send(ReservationRequest reservationRequest) {
16.        jmsTemplate.convertAndSend
17.            ("bookreservation", reservationRequest);
18.    }
19. }
```

In the above code, you have noticed the use of **JmsTemplate**, which is a crucial class in the Spring Framework for simplifying JMS processing. It abstracts away the complexity of the JMS API, managing tasks like creating connections, sessions, and message producers. **JmsTemplate** simplifies both sending and receiving messages by providing several convenient methods, including the `send` method for basic message dispatching, the `convertAndSend` method for automatic message conversion, `receive()`, and `receiveAndConvert()` for message consumption.

**JmsTemplate** supports both queue and topic models and integrates with Spring's transaction management, ensuring JMS operations can participate in managed transactions. While it uses a **SimpleMessageConverter** by default for message conversion, it can be customized for more complex

object serialization. Typically, it is configured with a **ConnectionFactory**, allowing pooled or shared connections to optimize performance.

In our example, **JmsTemplate** is injected via constructor injection, and we define a **send** method that takes a **ReservationRequest** object. This method utilizes **convertAndSend** to send the object to a JMS destination named **bookreservation**. This is the same queue destination our JMS listener is configured to listen to.

As the next step, we define the following JMS configuration for the client application. The configuration is relatively simple; we only need to define a **MessageConverter** bean. You can create a configuration class named **BookclubClientConfiguration.java** for this purpose:

```
1. package com.bpb.hssb.ch9.bookclub.client;
2. import org.springframework.context.annotation.Bean;
3. import org.springframework.context.annotation.Configuration;
4. import org.springframework.jms.core.JmsTemplate;
5. import org.springframework.jms.support.converter.
   MappingJackson2MessageConverter;
6. import org.springframework.jms.support.converter.MessageConverter;
7. import org.springframework.jms.support.converter.MessageType;
8.
9. @Configuration
10. public class BookclubClientConfiguration {
11.
12.     @Bean
13.     public ReservationRequestSender reservationRequestSender
   (JmsTemplate jmsTemplate) {
14.         return new ReservationRequestSender(jmsTemplate);
15.     }
16.
17.     @Bean
18.     public MessageConverter jacksonJmsMessageConverter() {
19.
20.         MappingJackson2MessageConverter converter =
   new MappingJackson2MessageConverter();
21.         converter.setTargetType(MessageType.TEXT);
22.         converter.setTypeIdPropertyName("_type");
23.         return converter;
24.     }
}
```

```
25.  
26. }  
27.
```

In the previous section, we created an application with embedded Apache ActiveMQ Artemis and configured it to start during the application start-up. For this sample, our objective is to connect to the embedded broker initialized by the previous application and send a message.

To achieve this, we need to provide the broker's connection details. This can be done by adding the following configuration to the **application.properties** file:

```
1. spring.artemis.mode=native  
2. spring.artemis.host=localhost  
3. spring.artemis.port=61616  
4. spring.artemis.user=artemis  
5. spring.artemis.password=simetraehcapa  
6. logging.level.org.apache.activemq=ERROR
```

As the final step, we can modify the application class to send a message to the broker. The complete code for the application class is given as follows:

```
1. package com.bpb.hssb.ch9.bookclub.client;  
2.  
3. import org.slf4j.Logger;  
4. import org.slf4j.LoggerFactory;  
5. import org.springframework.boot.CommandLineRunner;  
6. import org.springframework.boot.SpringApplication;  
7. import org.springframework.boot.autoconfigure.SpringBootApplication;  
8.  
9. import com.bpb.hssb.ch9.bookclub.domain.ReservationRequest;  
10.  
11. @SpringBootApplication  
12. public class BookclubClientApplication implements CommandLineRunner {  
13.  
14.     private static final Logger logger =  
        LoggerFactory.getLogger(BookclubClientApplication.class);  
15.  
16.     private ReservationRequestSender sender;  
17.  
18.     public BookclubClientApplication(ReservationRequestSender sender) {  
19.         this.sender = sender;  
20.     }
```

```

21.
22. public static void main(String[] args) {
23.     SpringApplication.run(BookclubClientApplication.class, args);
24. }
25.
26. @Override
27. public void run(String... args) throws Exception {
28.
29.     ReservationRequest request = ReservationRequest.builder()
30.         .club("NYC")
31.         .referenceId("R4551524")
32.         .book("Master Java17")
33.         .build();
34.
35.     sender.send(request);
36.     logger.info("ReservationRequest published successfully");
37. }

```

In the above code, we simply created a **ReservationRequest** object and used the **ReservationRequestSender** to send this request to the broker. If you run this application, you should see the following output in the console. One important thing to note is to ensure that the listener sample from the previous section is running and remains in the running state.



```

PROBLEMS 4 OUTPUT DEBUG CONSOLE TERMINAL PORTS
● + 3-bookclub-jms-client git:(master) ✘ cd /Users/sagara/Dev/gdrive/Book-Spring6/code/ch9/3-bookclub-jms-client ; /usr/bin/env /Library/Java/JavaVirtualMachines/jdk-17
8.2.jdk/Contents/Home/bin/java @var/folders/2g/tj3r6vs13vz808xxnrqhb6wr000gn/T/cp_5ojtkpm9sqzeom1019yintax.argfile com.bpb.hssb.ch9.bookclub.client.BookclubClientAp
plication
:: Spring Boot :: (v3.3.0)
2024-10-12T17:11:42.394+13:00  INFO 2162 --- [helloworld] [main] c.b.h.c.b.c.BookclubClientApplication : Starting BookclubClientApplication using Java 17.
0.2 with PID 2162 (/Users/sagara/Dev/gdrive/Book-Spring6/code/ch9/3-bookclub-jms-client/target/classes started by sagara in /Users/sagara/Dev/gdrive/Book-Spring6/code/c
h9/3-bookclub-jms-client)
2024-10-12T17:11:42.396+13:00  INFO 2162 --- [helloworld] [main] c.b.h.c.b.c.BookclubClientApplication : No active profile set, falling back to 1 default
profile: "default"
2024-10-12T17:11:42.865+13:00  INFO 2162 --- [helloworld] [main] c.b.h.c.b.c.BookclubClientApplication : Started BookclubClientApplication in 8.66 seconds
2024-10-12T17:11:43.036+13:00  INFO 2162 --- [helloworld] [main] c.b.h.c.b.c.BookclubClientApplication : ReservationRequest published successfully

```

*Figure 9.4: Publishing a JMS message to ActiveMQ Artemis*

Once the listener is running, you should be able to see a console output similar to the following screenshot from the listener application:



```

PROBLEMS 10 OUTPUT DEBUG CONSOLE TERMINAL PORTS

Hibernate:
  insert
  into
    reservation_request
    (book, club, reference_id, reservation_id)
  values
    (?, ?, ?, ?)
2024-10-12T17:15:36.533+13:00  INFO 78547 — [bookclub] [IntContainer#0-1] c.b.h.c.b.s.ReservationServiceImpl
created : ReservationRequest(reservationId=7, referenceId=R4551524, club=NYC, book=Master Java17)  : ReservationRequest

```

*Figure 9.5: Receiving a JMS message from ActiveMQ Artemis*

The **JmsTemplate** we used here can also be utilized for writing test cases. You can set up an embedded message broker by defining a Spring configuration similar to the one we developed for our examples.

Regarding security, beyond application-level security, you can also configure authentication and authorization at the message broker level. This can be done using username/password credentials or SSL/TLS certificates to secure the communication between the applications and the broker. Then, you can configure the **JmsTemplate** with a secured **ConnectionFactory** that incorporates these authentication schemas.

In this section, we successfully created both a JMS listener and sender and tested their functionality. In the next section, we will try converting this application to use AMQP with RabbitMQ as the message broker.

## Spring Messaging with RabbitMQ

**Advanced Message Queuing Protocol (AMQP)** and **Java Message Service (JMS)** are both used for messaging in distributed systems, but they differ significantly in their design and scope.

AMQP is a binary, application-layer protocol designed for efficient message exchange between distributed systems. Operating over TCP/IP, AMQP defines robust messaging capabilities that involve exchanges, queues, and bindings, which together support flexible routing. The protocol specifies both the wire-level format for message transfer and the broker behavior for message handling. AMQP also features credit-based flow control for managing congestion and offers transactions and publisher confirms. The platform-independent type system allows the serialization of complex data across various environments.

JMS, on the other hand, is limited to Java environments. While JMS supports only two messaging models (queues and topics), AMQP provides

more versatility with four message models. AMQP also comes with native security features, such as SASL and TLS, whereas security in JMS depends on the provider. AMQP messages are encoded in binary format, while JMS supports formats like **BytesMessage** and **TextMessage**. Additionally, AMQP facilitates distributed transactions and broker-specific extensions without compromising protocol integrity. In summary, JMS is well-suited for Java-centric systems that need standard messaging capabilities, while AMQP excels in cross-platform communication and interoperability, especially in scenarios requiring advanced routing.

As our next step, let us rewrite the previous example to use AMQP with RabbitMQ as the broker. To get started, install RabbitMQ by following the official RabbitMQ documentation. After installation, start the RabbitMQ broker.

Begin by adding the following dependencies to your project's POM file and make sure the broker is running:

1. `<dependency>`
2. `<groupId>org.springframework.boot</groupId>`
3. `<artifactId>spring-boot-starter-amqp</artifactId>`
4. `</dependency>`
5. `<dependency>`
6. `<groupId>org.springframework.boot</groupId>`
7. `<artifactId>spring-boot-starter-json</artifactId>`
8. `</dependency>`

In the above code, the first dependency, **spring-boot-starter-amqp**, brings in all the essential libraries for working with AMQP in our sample project. It configures auto-configuration for connection factories and message listeners and provides a template to simplify the integration of RabbitMQ with Spring Boot.

The second dependency, **spring-boot-starter-json**, incorporates JSON processing libraries like **Jackson**. This enables automatic serialization and deserialization of objects into JSON, which is particularly useful when handling message payloads in Spring applications.

Following the same pattern as in the JMS application, create a class named **ReservationRequestReceiver.java** with the following code:

1. `package com.bpb.hssb.ch9.bookclub.receiver;`
- 2.

```

3. import org.slf4j.Logger;
4. import org.slf4j.LoggerFactory;
5.
6. import com.bpb.hssb.ch9.bookclub.domain.ReservationRequest;
7. import com.bpb.hssb.ch9.bookclub.service.ReservationService;
8. import com.bpb.hssb.ch9.bookclub.service.ReservationServiceImpl;
9.
10. public class ReservationRequestReceiver {
11.
12.     private static final Logger logger =
13.         LoggerFactory.getLogger(ReservationServiceImpl.class);
14.
15.     private ReservationService reservationService;
16.
17.     public ReservationRequestReceiver
18.         (ReservationService reservationService) {
19.         this.reservationService = reservationService;
20.     }
21.
22.     public void receiveMessage (ReservationRequest reservationRequest) {
23.         ReservationRequest created =
24.             reservationService.createReservationRequest(reservationRequest);
25.         logger.info("ReservationRequest created : " + created);
26.     }

```

The above class is a simple Java class with a single method called **receiveMessage**, which accepts a **ReservationRequest** object and returns the persisted request. There is no need to include any AMQP-specific code within this class, maintaining simplicity and focus on business logic.

Next, we can remove the **JMSConfiguration** from the project and replace it with a new class called **AMQPConfiguration**, containing the following configuration code:

```

1. package com.bpb.hssb.ch9.bookclub.config;
2.
3. import org.springframework.amqp.core.BindingBuilder;
4. import org.springframework.amqp.core.TopicExchange;
5. import org.springframework.context.annotation.Bean;

```

```
6. import org.springframework.context.annotation.Configuration;
7.
8. import com.bpb.hssb.ch9.bookclub.receiver.ReservationRequestReceiver;
9.
10. import org.springframework.amqp.core.Binding;
11. import org.springframework.amqp.core.Queue;
12. import org.springframework.amqp.rabbit.connection.ConnectionFactory;
13. import org.springframework.amqp.rabbit.listener.
    SimpleMessageListenerContainer;
14. import org.springframework.amqp.rabbit.listener.
    adapter.MessageListenerAdapter;
15. import org.springframework.amqp.support.converter.
    Jackson2JsonMessageConverter;
16. import org.springframework.amqp.support.
    converter.MessageConverter;
17.
18. @Configuration
19. public class AMQPConfiguration {
20.
21.     static final String queueName = "bookreservation";
22.
23.     static final String topicExchangeName =
    "bookreservation-exchange";
24.
25.     @Bean
26.     Queue queue() {
27.         return new Queue(queueName, false);
28.     }
29.
30.     @Bean
31.     TopicExchange exchange() {
32.         return new TopicExchange(topicExchangeName);
33.     }
34.
35.     @Bean
36.     Binding binding(Queue queue, TopicExchange exchange) {
37.         return BindingBuilder.bind(queue).to
    (exchange).with("foo.bar.#");
38.     }
39.
40.     @Bean
41.     SimpleMessageListenerContainer container
```

```

        (ConnectionFactory connectionFactory,
42.      MessageListenerAdapter listenerAdapter) {
43.      SimpleMessageListenerContainer container =
44.          new SimpleMessageListenerContainer();
45.      container.setConnectionFactory(connectionFactory);
46.      container.setQueueNames(queueName);
47.      container.setMessageListener(listenerAdapter);
48.      return container;
49.
50.  @Bean
51.  MessageListenerAdapter listenerAdapter
52.  (ReservationRequestReceiver receiver) {
53.      MessageListenerAdapter adapter =
54.          new MessageListenerAdapter(receiver, "receiveMessage");
55.      adapter.setMessageConverter(jsonMessageConverter());
56.
57.  @Bean
58.  public MessageConverter jsonMessageConverter() {
59.      return new Jackson2JsonMessageConverter();
60.  }
61. }

```

In the above code, the first bean creates a non-durable queue with a specific name. The second bean establishes a topic exchange, which is an AMQP exchange type that routes messages to queues based on wildcard matches in a routing key. The third bean binds the queue to the topic exchange using the "**foo.bar.#**" routing pattern. This pattern ensures that messages with routing keys starting with "**foo.bar**". are routed to the appropriate queue. Together, these beans configure a basic messaging setup where messages are published to the topic exchange and delivered to the queue based on the routing key.

The last three beans set up a message-listening infrastructure in the Spring application. The **SimpleMessageListenerContainer** creates a listener that connects to the broker using the provided **ConnectionFactory** and listens for messages from the designated queue. The **MessageListenerAdapter** is equipped with a JSON message converter, enabling automatic deserialization of JSON messages into Java objects. Lastly, the **jsonMessageConverter** bean creates a **Jackson2JsonMessageConverter** for JSON processing.

As the final step, add the following broker connection details to the **application.properties** file:

1. `spring.rabbitmq.password=guest`
2. `spring.rabbitmq.username=guest`

Now, you can run the application that listens for messages. Make sure you have started the RabbitMQ broker before you run the application. Next, let us try to rewrite the client-side application to send to the AMQP broker, and we can test end to end.

First, update the **ReservationRequestSender** class with the following code:

```
1. package com.bpb.hssb.ch9.bookclub.client;
2.
3. import org.springframework.amqp.rabbit.core.RabbitTemplate;
4. import com.bpb.hssb.ch9.bookclub.domain.ReservationRequest;
5.
6. public class ReservationRequestSender {
7.
8.     private String topicExchangeName;
9.
10.    private RabbitTemplate rabbitTemplate;
11.
12.    public ReservationRequestSender(RabbitTemplate rabbitTemplate,
13.                                     String topicExchangeName) {
14.        this.rabbitTemplate = rabbitTemplate;
15.        this.topicExchangeName = topicExchangeName;
16.    }
17.    public void send(ReservationRequest reservationRequest) {
18.        rabbitTemplate.convertAndSend(topicExchangeName,
19.                                     "foo.bar.baz", reservationRequest);
20.    }
21. }
```

In the above code, similar to **JMSTemplate**, you can see the **RabbitTemplate**. This class is central to the Spring integration with RabbitMQ simplifying interactions. It also abstracts away the complexities of the AMQP protocol, handling common operations such as establishing connections, creating channels, and managing transactions. With **RabbitTemplate**, developers can focus on the business logic instead of low-level messaging details.

**RabbitTemplate** supports various messaging patterns, including simple sends, publish/subscribe, and request/reply. It also integrates seamlessly with Spring's dependency injection and transaction management capabilities. Moreover, it provides built-in message conversion between Java objects and AMQP messages, making it easier to manage complex data structures.

The **ReservationRequestSender** class leverages **RabbitTemplate** to send reservation requests to the RabbitMQ broker. When the send method is invoked, it uses RabbitTemplate's **convertAndSend** method to convert a **ReservationRequest** object into a message and transmit it to a topic exchange using the routing key "**foo.bar.baz**". The **convertAndSend** method simplifies the process by handling both the message conversion and transmission. This class serves as a client-side component in the messaging architecture, enabling other parts of the application to publish reservation requests easily to the RabbitMQ exchange.

As the next step, we can define the client configuration by creating the **BookclubClient Configuration.java** class with the following code:

```
1. package com.bpb.hssb.ch9.bookclub.client;
2.
3. import org.springframework.amqp.core.BindingBuilder;
4. import org.springframework.amqp.core.TopicExchange;
5. import org.springframework.context.annotation.Bean;
6. import org.springframework.context.annotation.Configuration;
7.
8. import org.springframework.amqp.core.Binding;
9. import org.springframework.amqp.core.Queue;
10. import org.springframework.amqp.rabbit.core.RabbitTemplate;
11. import org.springframework.amqp.support.converter.Jackson2JsonMessageConverter;
12. import org.springframework.amqp.support.converter.MessageConverter;
13.
14. @Configuration
15. public class BookclubClientConfiguration {
16.
17.     static final String queueName = "bookreservation";
18.
19.     static final String topicExchangeName = "bookreservation-exchange";
20.
21.     @Bean
22.     Queue queue() {
```

```

23.     return new Queue(queueName, false);
24. }
25.
26. @Bean
27. TopicExchange exchange() {
28.     return new TopicExchange(topicExchangeName);
29. }
30.
31. @Bean
32. Binding binding(Queue queue, TopicExchange exchange) {
33.     return BindingBuilder.bind(queue).to
34.         (exchange).with("foo.bar.#");
35. }
36. @Bean
37. public MessageConverter jsonMessageConverter() {
38.     return new Jackson2JsonMessageConverter();
39. }
40.
41. @Bean
42. public ReservationRequestSender sender
43.     (RabbitTemplate rabbitTemplate) {
44.     return new ReservationRequestSender
45.         (rabbitTemplate, topicExchangeName);
46. }

```

The above code defines several beans: a non-durable queue named **"bookreservation"**, a topic exchange named **"bookreservation-exchange"**, and a binding between them using the routing key pattern **"foo.bar.#"**. This configuration ensures that messages with routing keys starting with **"foo.bar."** will be routed to the defined queue.

Additionally, the configuration sets up a **Jackson2JsonMessageConverter** to handle serialization and deserialization of messages to and from JSON format, making it easier to send and receive structured data. A **ReservationRequestSender** bean is also created and configured with a **RabbitTemplate** and the topic exchange name, providing a convenient mechanism for sending reservation requests to the broker.

As the final step, add the following broker connection details to the

## application.properties file:

1. `spring.rabbitmq.password`=guest
2. `spring.rabbitmq.username`=guest

Now, if you run the **BookclubClientApplication**, you should see the following output in the console. Ensure that both the listener application from the previous section and the RabbitMQ broker are running before executing the client application.



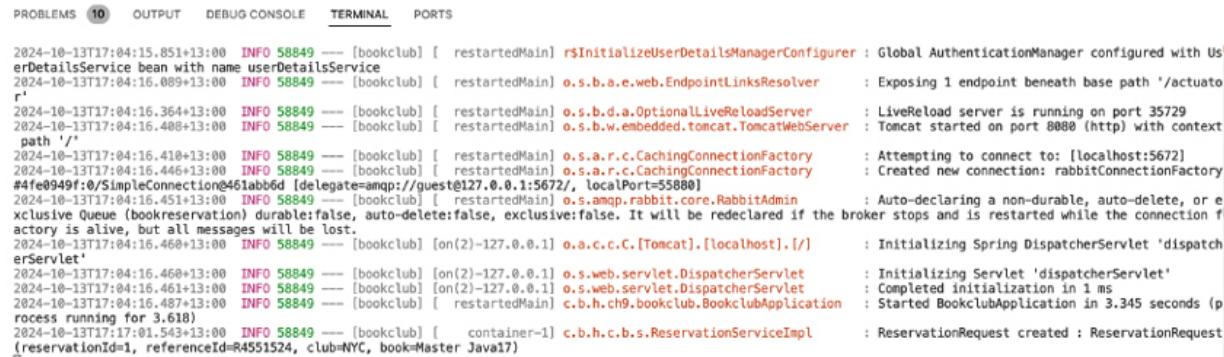
```
PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL PORTS

:: Spring Boot ::      (v3.3.0)

2024-10-13T17:17:00.615+13:00 INFO 75547 --- [helloworld] [      main] c.b.h.c.b.c.BookclubClientApplication : Starting BookclubClientApplication using Java 17.0.2
with PID 75547 (/Users/sagara/Dev/gdrive/Book-Spring6/code/ch9/5-bookclub-amqp-client/target/classes started by sagara in /Users/sagara/Dev/gdrive/Book-Spring6/code/ch9/5-bo
okclub-amqp-client)
2024-10-13T17:17:00.617+13:00 INFO 75547 --- [helloworld] [      main] c.b.h.c.b.c.BookclubClientApplication : No active profile set, falling back to 1 default prof
ile: "default"
2024-10-13T17:17:01.309+13:00 INFO 75547 --- [helloworld] [      main] c.b.h.c.b.c.BookclubClientApplication : Started BookclubClientApplication in 0.868 seconds (p
rocess running for 1.066)
2024-10-13T17:17:01.329+13:00 INFO 75547 --- [helloworld] [      main] o.s.a.r.c.CachingConnectionFactory : Attempting to connect to: [localhost:5672]
2024-10-13T17:17:01.389+13:00 INFO 75547 --- [helloworld] [      main] o.s.a.r.c.CachingConnectionFactory : Created new connection: rabbitConnectionFactory#c6e0f
32:0/SimpleConnection@216e9ca3 [delegate=amqp://guest@127.0.0.1:5672/, localPort=56528]
2024-10-13T17:17:01.393+13:00 INFO 75547 --- [helloworld] [      main] o.s.amqp.rabbit.core.RabbitAdmin : Auto-declaring a non-durable, auto-delete, or exclusi
ve Queue (bookreservation) durable:false, auto-delete:false, exclusive:false. It will be redeclared if the broker stops and is restarted while the connection factory is aliv
e, but all messages will be lost.
2024-10-13T17:17:01.411+13:00 INFO 75547 --- [helloworld] [      main] c.b.h.c.b.c.BookclubClientApplication : ReservationRequest request published successfully
```

Figure 9.6: Publishing an AMQP message to RabbitMQ

At the same time, you should be able to see the received message logs in the listener application that we created in the previous step:



```
PROBLEMS 10 OUTPUT DEBUG CONSOLE TERMINAL PORTS

2024-10-13T17:04:15.851+13:00 INFO 58849 --- [bookclub] [ restartedMain] r$InitializeUserDetailsManagerConfigurer : Global AuthenticationManager configured with Us
erDetailsService bean with name userDetailsService
2024-10-13T17:04:16.089+13:00 INFO 58849 --- [bookclub] [ restartedMain] o.s.b.a.e.web.EndpointLinksResolver : Exposing 1 endpoint beneath base path '/actuator'
2024-10-13T17:04:16.364+13:00 INFO 58849 --- [bookclub] [ restartedMain] o.s.b.d.a.OptionalLiveReloadServer : LiveReload server is running on port 35729
2024-10-13T17:04:16.408+13:00 INFO 58849 --- [bookclub] [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 8080 (http) with context
path '/'
2024-10-13T17:04:16.410+13:00 INFO 58849 --- [bookclub] [ restartedMain] o.s.a.r.c.CachingConnectionFactory : Attempting to connect to: [localhost:5672]
2024-10-13T17:04:16.446+13:00 INFO 58849 --- [bookclub] [ restartedMain] o.s.a.r.c.CachingConnectionFactory : Created new connection: rabbitConnectionFactory
#4fe0949f:0/SimpleConnection@461abb6d [delegate=amqp://guest@127.0.0.1:5672/, localPort=55880]
2024-10-13T17:04:16.451+13:00 INFO 58849 --- [bookclub] [ restartedMain] o.s.amqp.rabbit.core.RabbitAdmin : Auto-declaring a non-durable, auto-delete, or e
xclusive Queue (bookreservation) durable:false, auto-delete:false, exclusive:false. It will be redeclared if the broker stops and is restarted while the connection f
actory is alive, but all messages will be lost.
2024-10-13T17:04:16.460+13:00 INFO 58849 --- [bookclub] [on(2)-127.0.0.1] o.a.c.c.C[Tomcat].[localhost].[] : Initializing Spring DispatcherServlet 'dispatch
erServlet'
2024-10-13T17:04:16.460+13:00 INFO 58849 --- [bookclub] [on(2)-127.0.0.1] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
2024-10-13T17:04:16.461+13:00 INFO 58849 --- [bookclub] [on(2)-127.0.0.1] o.s.web.servlet.DispatcherServlet : Completed initialization in 1 ms
2024-10-13T17:04:16.487+13:00 INFO 58849 --- [bookclub] [ restartedMain] c.b.h.chp.bookclub.BookclubApplication : Started BookclubApplication in 3.345 seconds (p
rocess running for 3.618)
2024-10-13T17:04:16.543+13:00 INFO 58849 --- [bookclub] [ container=1] c.b.h.c.b.s.ReservationServiceImpl : ReservationRequest created : ReservationRequest
(reservationId=1, referenceId=R4551524, club=NYC, book=Master Java17)
```

Figure 9.7: Receiving an AMQP message from RabbitMQ

Similar to the **JmsTemplate** discussed in the previous section, you can also use **RabbitTemplate** to write test cases for AMQP-related message applications. It allows you to send and receive messages in tests, ensuring your messaging logic functions correctly without requiring a fully running RabbitMQ instance. You can leverage embedded brokers or mock RabbitMQ configurations to simplify testing scenarios.

In terms of security measures, similar to JMS, you can configure TLS/SSL

for RabbitMQ connections to encrypt data in transit, ensuring communication between the application and broker remains secure. Additionally, you can configure Spring Boot to use these secure connections by specifying the appropriate settings in your application configuration files. RabbitMQ also provides built-in authorization mechanisms to control user permissions over exchanges, queues, and vhosts. For message-level security, consider implementing message signing and encryption using Spring AMQP's **MessagePostProcessor**. This ensures that messages are both tamper-proof and confidential, providing an extra layer of security in distributed messaging systems.

## Conclusion

This chapter has provided a foundational understanding of messaging, emphasizing when and where messaging can be effectively applied. We explored essential messaging patterns, such as queue-based and topic-based communication, highlighting their use cases and differences. Additionally, we covered the practical aspects of sending and receiving JMS messages using Spring with ActiveMQ Artemis, along with an introduction to AMQP messaging with RabbitMQ. With these insights and hands-on techniques, you can leverage messaging systems efficiently in real-world applications.

As we are gradually reaching the end of this book, in the next chapter, we will learn some important topics related to running a Spring Boot application in production, such as deploying an application into different environments using profiles, packaging options such as Docker and Spring Native.

# CHAPTER 10

# Running Spring Boot in Production

## Introduction

As we reach the final chapters of this book, this chapter will introduce several essential features you need to know to run a Spring Boot application in production. We will start with the profile concept, which serves as the foundation for running a Spring Boot application in multiple environments. Next, we will discuss packaging options, such as Docker and GraalVM native images. We will also cover how to enable SSL for Spring Boot applications and integrate with identity providers to offer enterprise-grade security for your users. Finally, we will discuss how you can collect, process, and visualize logs, metrics, and traces in a production system.

## Structure

The chapter covers the following topics:

- Spring Profiles
- Spring Boot with Docker
- Spring Boot with GraalVM
- Use Jetty and Undertow with Spring Boot
- Configuring SSL for Spring Boot applications
- Identity provider integration for Spring Boot
- Working with logs, metrics, and traces

## Objectives

By the end of this chapter, you will understand how to use the Spring Profile concept to run your applications in different environments, create Docker and GraalVM native images, and customize the default web server to use Jetty and Undertow. You will also learn how to enable SSL for your application and integrate it with identity providers to provide enhanced security features. Lastly, you will gain an understanding of how to collect, analyze, and visualize logs, metrics, and traces in a production system.

## Spring Profiles

In real-world application development, you do not push an application directly from your development machine into production. Instead, it goes through a deployment pipeline defined by your team and organization. Typically, you start by writing and testing your application code thoroughly on your local machine. Once satisfied with the functionality and quality, you commit it to a source code repository like Git. From there, you use a build script or tool, such as Jenkins or GitHub Actions, to build a fresh application and deploy it to a shared development environment used by your team. This environment is often called **dev** or **development**. If the application works well here, you promote it to the next environment, commonly known as **quality assurance (QA)** or Test, where it undergoes manual or automated testing. In practice, there may be multiple test environments, each focusing on specific quality aspects like **user experience (UX)**, performance, or security. Once quality checks are complete, the application is moved to production. Some workflows may also include additional environments, such as pre-production or staging, before reaching production.

As the application moves across environments, certain configuration parameters need to be adjusted. For instance, different environments should use separate database instances, meaning database URLs and credentials vary by environment. In a typical Spring Boot application, you have to mainly deal with two types of environments-specific configuration elements:

- **Configuration properties:** The external dependencies of your application, such as databases, message brokers, and file servers, vary across environments. Database pool configurations, caching settings,

and resource allocation parameters are also unique to each environment. Instead of hardcoding these values in the code, we externalize them as injectable configuration parameters so that different values can be injected per environment. In previous examples, we used the **@Value** annotation to achieve this.

- **Spring bean configuration:** Security configurations may differ from one environment to another, and sometimes, additional or unique functional components are needed for each environment. For example, in a shopping cart application, a mock or sandbox implementation of a payment gateway may be used in non-production environments. These situations lead to different or additional Spring bean configurations for each environment.

To support environment-specific configurations, Spring supports a concept called profile that goes hand in hand with the Spring **@Value** annotation that we discussed earlier. As our next exercise, let us try to apply the profile concept to an application that we developed earlier.

We use the RESTful service sample that we created in [Chapter 2](#) to implement the new concepts that we are discussing in this chapter. The initial project is available in the companion GitHub repository to make it easy for you.

First, let us introduce a new domain class called **ServiceInfo.java** using the following code; this class represents some environment-specific information that we are going to return through our RESTful service:

```
1. package com.bpb.hssb.ch10.bookclub.domain;  
2.  
3. import lombok.AllArgsConstructor;  
4. import lombok.Builder;  
5. import lombok.Getter;  
6. import lombok.NoArgsConstructor;  
7. import lombok.Setter;  
8. import lombok.ToString;  
9.  
10. @Getter  
11. @Setter  
12. @NoArgsConstructor  
13. @NoArgsConstructor  
14. @Builder
```

```
15. @ToString
16. public class ServiceInfo {
17.
18.     private String environment;
19.     private String version;
20.
21. }
```

In the above class, we have defined two fields to represent the environment name and version of the application deployed in the current environment.

As the second step, let us also create another domain class called **ProductionRevision** to represent the production revision value of the application enforced by the platform security team:

```
1. package com.bpb.hssb.ch10.bookclub.domain;
2.
3. import lombok.AllArgsConstructor;
4. import lombok.Builder;
5. import lombok.Getter;
6. import lombok.NoArgsConstructor;
7. import lombok.Setter;
8. import lombok.ToString;
9.
10. @Getter
11. @Setter
12. @NoArgsConstructor
13. @AllArgsConstructor
14. @Builder
15. @ToString
16. public class ProductionRevision {
17.
18.     private String revision;
19.
20. }
```

Next, we can add the **ServiceInfoController.java** class to expose this information as a RESTful endpoint:

```
1. package com.bpb.hssb.ch10.bookclub.controller;
2.
3. import org.springframework.web.bind.annotation.GetMapping;
4. import org.springframework.web.bind.annotation.RequestMapping;
5. import org.springframework.web.bind.annotation.RestController;
```

```

6.
7. import com.bpb.hssb.ch10.bookclub.domain.ServiceInfo;
8. import com.bpb.hssb.ch10.bookclub.service.BookServiceException;
9.
10. @RestController
11. @RequestMapping("/api")
12. public class ServiceInfoController {
13.
14.     private ServiceInfo serviceInfo;
15.
16.     public ServiceInfoController(ServiceInfo serviceInfo) {
17.
18.         this.serviceInfo = serviceInfo;
19.     }
20.
21.     @GetMapping("/info")
22.     public ServiceInfo getInfo() throws BookServiceException {
23.         return serviceInfo;
24.     }
25.
26. }

```

In the above code, we just injected the **ServiceInfo** bean using the constrictor injection and then returned it using the **getInfo** method. As the final step, let us modify the bean configuration with the **ServiceInfo** bean; we can do that by modifying the **BookclubConfiguration** class; the full code of the updated **BookclubConfiguration** class is given as follows:

```

1. package com.bpb.hssb.ch10.bookclub.config;
2.
3. import org.springframework.context.annotation.Bean;
4. import org.springframework.context.annotation.Configuration;
5.
6. import com.bpb.hssb.ch10.bookclub.domain.ServiceInfo;
7. import com.bpb.hssb.ch10.bookclub.repository.BookCopyJPARepository;
8. import com.bpb.hssb.ch10.bookclub.repository.BookJPARepository;
9. import com.bpb.hssb.ch10.bookclub.service.BookService;
10. import com.bpb.hssb.ch10.bookclub.service.BookServiceImpl;
11.
12. @Configuration
13. public class BookclubConfiguration {
14.

```

```

15.  @Bean
16.  public BookService bookService(BookJPARespository
17.    bookRepository, BookCopyJPARespository bookCopyJPARespository) {
18.      return new BookServiceImpl(bookRepository,
19.        bookCopyJPARespository);
20.  }
21.  @Bean
22.  public ServiceInfo serviceInfo() {
23.      return new ServiceInfo("production", "2.0.0");
24.  }

```

If you run the application and hit the **serviceInfo** endpoint using an HTTP client such as **Postman**, you will be able to see the result as given in the following figure:

The screenshot shows the Postman interface with a GET request to the endpoint `http://127.0.0.1:8080/api/info`. The response body is a JSON object:

```

1: {
2:   "environment": "production",
3:   "version": "2.0.0"
4: }

```

*Figure 10.1: Calling info endpoint using Postman*

If you move this application from one environment to another, nothing will change because we have hard-coded the environment-specific values in the **BookclubConfiguration**. To avoid confusion, we have now implemented the problem that we intended to solve using the Spring profile concept. To start solving the problem, first, let us remove the **ServiceInfo** bean from the **BookclubConfiguration**. The updated **BookclubConfiguration** class should look as follows:

1. package com.bpb.hssb.ch10.bookclub.config;

```

2.
3. import org.springframework.context.annotation.Bean;
4. import org.springframework.context.annotation.Configuration;
5.
6. import com.bpb.hssb.ch10.bookclub.domain.ServiceInfo;
7. import com.bpb.hssb.ch10.bookclub.repository.BookCopyJPARepository;
8. import com.bpb.hssb.ch10.bookclub.repository.BookJPARepository;
9. import com.bpb.hssb.ch10.bookclub.service.BookService;
10. import com.bpb.hssb.ch10.bookclub.service.BookServiceImpl;
11.
12. @Configuration
13. public class BookclubConfiguration {
14.
15.     @Bean
16.     public BookService bookService(BookJPARepository
17.         bookRepository, BookCopyJPARepository bookCopyJPARepository) {
18.         return new BookServiceImpl(bookRepository,
19.             bookCopyJPARepository);
20.     }
21. }

```

Next, let us introduce two environment-specific Bean configuration classes into the project. Go ahead and add a new class called **DevelopmentConfiguration** to add the following code-specific to the development environment:

```

1. package com.bpb.hssb.ch10.bookclub.config;
2.
3. import org.springframework.beans.factory.annotation.Value;
4. import org.springframework.context.annotation.Bean;
5. import org.springframework.context.annotation.Configuration;
6. import org.springframework.context.annotation.Profile;
7.
8. import com.bpb.hssb.ch10.bookclub.domain.ServiceInfo;
9.
10. @Configuration
11. @Profile("development")
12. public class DevelopmentConfiguration {
13.
14.     @Value("${bookclub.service.info.environment}")

```

```

15. private String environment;
16.
17. @Value("${bookclub.service.info.version}")
18. private String version;
19.
20. @Bean
21. public ServiceInfo serviceInfo() {
22.     return new ServiceInfo(environment, version);
23. }
24.

```

In the above code, we have used **@Value** annotation to inject values from the **application.properties** file and specified names of the configuration parameters using the  **\${...}**  syntax. Then, we used these configuration parameter values to create the **ServiceInfo** bean. You cannot see any hard-coded values in this configuration.

The **@Profile** annotation in Spring Boot is the most important annotation when it comes to handling environment-specific configurations. It allows developers to conditionally include or exclude components, beans, or entire configuration classes based on which profiles are active. The **@Profile** annotation can be applied at the class level or method level. When used on a **@Configuration** class, it determines whether all **@Bean** methods within that class should be processed. When applied to individual **@Bean** methods, it controls the creation of specific beans. The annotation accepts one or more profile names as arguments. In our code, we have used this annotation at the class level and passed development as the value specifying this configuration should only be active when the development profile is active. You can also use logical operators for more complex conditions, such as **@Profile("production & !test")**, to activate a component in production but not in test environments. Profiles can be activated through various means, including **application.properties**, command-line arguments, or programmatically. This flexibility allows for easy switching between different environments or configurations without changing the application code.

Next, we can add the **ProductionConfiguration** using the following code that represents the production environment-specific configuration of the application:

```

1. package com.bpb.hssb.ch10.bookclub.config;

```

```

2.
3. import org.springframework.beans.factory.annotation.Value;
4. import org.springframework.context.annotation.Bean;
5. import org.springframework.context.annotation.Configuration;
6. import org.springframework.context.annotation.Profile;
7.
8. import com.bpb.hssb.ch10.bookclub.domain.ProductionRevision;
9. import com.bpb.hssb.ch10.bookclub.domain.ServiceInfo;
10.
11. @Configuration
12. @Profile("production")
13. public class ProductionConfiguration {
14.
15.     @Value("${bookclub.service.info.environment}")
16.     private String environment;
17.
18.     @Value("${bookclub.service.info.version}")
19.     private String version;
20.
21.     @Value("${bookclub.service.info.revision}")
22.     private String revision;
23.
24.     @Bean
25.     ProductionRevision productionRevision() {
26.         return new ProductionRevision(revision);
27.     }
28.
29.     @Bean
30.     public ServiceInfo serviceInfo(ProductionRevision
31.         productionRevision) {
32.         String versionStr = version + "_" +
33.             productionRevision.getRevision();
34.         return new ServiceInfo(environment, versionStr);
35.     }
36. }

```

This code is pretty identical to **DevelopmentConfiguration** class that we created previously. The main difference is that we have defined an additional bean called **productionRevision** that is only specific to the production environment and is used to calculate the version of the application. Although this is a bit of an artificial case, this is good enough to demonstrate the

environment-specific bean configurations.

Now, we can define the actual values by creating environment-specific configuration properties files. First, create a file called **application-development.properties** in the same directory where you have the default **application.properties** file with the following values:

1. `bookclub.service.info.environment=development`
2. `bookclub.service.info.version=2.2.0-Beta`

Then, we can create a file called **application-production.properties** with the following values:

1. `bookclub.service.info.environment=production`
2. `bookclub.service.info.version=2.0.0`
3. `bookclub.service.info.revision=R20241015`

Now, as we are all set, go ahead and run the application.

If you have done everything correctly, you will be able to see the following error message:

```
*****
APPLICATION FAILED TO START
*****
Description:
Parameter 0 of constructor in com.bpb.hssb.ch10.bookclub.controller.ServiceInfoController required a bean of type 'com.bpb.hssb.ch10.bookclub.domain.ServiceInfo' that could not be found.

Action:
Consider defining a bean of type 'com.bpb.hssb.ch10.bookclub.domain.ServiceInfo' in your configuration.
```

*Figure 10.2: Error message caused without picking the right profile*

It makes sense to get this error message because we have not activated any profile yet, although we have configured two profiles. The application only loads the **BookclubConfiguration** that is common for all the profiles, but there is no bean definition for **ServiceInfo** bean.

First, let us activate the development profile by adding the following line to the default **application.properties** file. This setting would activate the development profile, which results in the load of the configuration from the **DevelopmentConfiguration** class:

1. `spring.profiles.active=development`

Now, if you rerun the application and hit the endpoint using the Postman, you will be able to see the details related to the development environment.

Next, we can change the active profile into production by changing the same

setting as follows:

```
1. spring.profiles.active=production
```

If you rerun and test the application endpoint using Postman again, you can now see details related to the production environment. You can pass the whole **application.properties** file externally using the **spring.config.location** environment variables.

Alternative to defining the active profiles using the **application.properties** file and also to override the default profile, you can pass the active profile using **spring.profile.active** parameter using one of the following approaches:

- Using command-line parameter:

```
1. java -jar bookclub-env-0.0.1-SNAPSHOT.jar --spring.profiles.active=development
```

- Using environment variable:

```
1. export SPRING_PROFILES_ACTIVE=development
```

```
2. java -jar bookclub-env-0.0.1-SNAPSHOT.jar --spring.profiles.active=development
```

- Using system property:

```
1. java -Dspring.profiles.active=development -jar bookclub-env-0.0.1-SNAPSHOT.jar
```

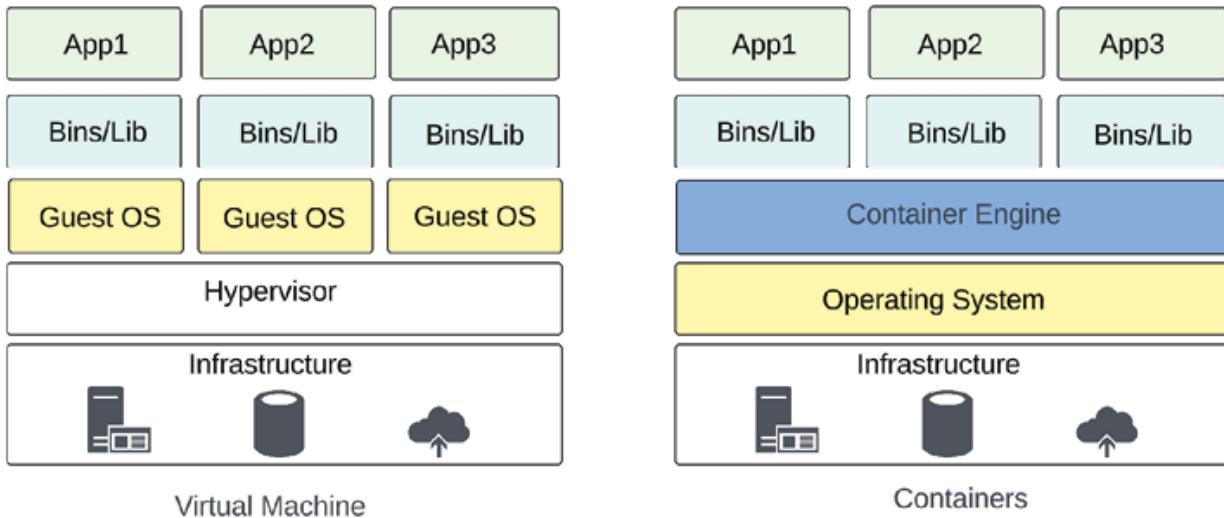
In this section, we looked at one of the important features required to run Spring Boot applications in different environments. In the next section, we will discuss packaging options important in running Spring Boot applications in production.

## Spring Boot with Docker

During the last decade, containers have revolutionized software development and deployment by providing a lightweight, portable, and efficient way to package and run applications. Containers are standalone, executable units that encapsulate an application along with all its dependencies, libraries, and configuration files. This encapsulation ensures consistent application performance across different computing environments, from development to testing and production.

Docker, introduced in 2013, is the most popular toolset for creating and managing containers. While Docker dominates the container ecosystem, alternative options such as Podman and Containerd are also available. Docker containers share the host system's kernel, making them more lightweight and resource-efficient compared to traditional virtual machines.

Containers start up almost instantly and consume fewer system resources, enabling optimized use of hardware resources. The following figure illustrates how Docker's architecture differs from traditional virtual machines:



*Figure 10.3: Virtual machines vs. containers*

**Virtual machines (VMs)** and Docker containers are both virtualization technologies, but they differ significantly in their architecture and resource utilization. VMs use a hypervisor to create fully isolated environments with dedicated operating systems, virtualizing both the OS kernel and application layer. This provides strong isolation but at the cost of higher resource consumption and slower boot times. Docker containers, in contrast, share the host OS kernel and virtualize only the application layer using isolated user-space instances called containers. This lightweight approach allows containers to start in milliseconds, consume fewer resources, and achieve near-native performance. While VMs offer better security through complete isolation, containers provide superior portability and efficiency, making them ideal for microservices architectures and applications requiring rapid scaling.

In this section, we will package our sample application as a Docker image and run it as a container. As a prerequisite, you need to have Docker installed on your development machine. Depending on your operating system, you can install Docker Desktop or alternatives such as Rancher Desktop.

To package our application as a Docker image, we need to create a

descriptor file called a **Dockerfile**. The **Dockerfile** is a text-based script that contains instructions for building a Docker container image. It uses a simple, declarative syntax to specify a series of steps that Docker follows to assemble the image. These steps typically include selecting a base image, copying application files, installing dependencies, setting environment variables, and defining the command to run when the container starts. For our sample application, we will create a Dockerfile with the following code:

```
1. FROM eclipse-temurin:17-jdk-focal
2. RUN groupadd -r spring && useradd -r -g spring spring
3. ARG JAR_FILE=target/*.jar
4. COPY ${JAR_FILE} app.jar
5. RUN chown spring:spring app.jar
6. USER spring
7. ENTRYPOINT ["java", "-jar", "/app.jar"]
```

The explanation of the code is as follows:

- *Line 1* specifies the base image for the Docker container. Every Docker image we create in the real world is based on a base image. In this case, the base image provides both the operating system and the Java runtime required to run the application. We have specified Eclipse Temurin version 17, which is based on Ubuntu Focal.
- *Line 2* adds a new group and user named **spring**. The **-r** flag creates a system account. This is a security best practice to ensure the application runs as a non-root user.
- *Lines 3 and 4* copy the JAR file from the build context into the container, renaming it to **app.jar**.
- *Line 5* changes the ownership of the **app.jar** file to the **spring** user and group. This ensures that the application has the necessary permissions to run the JAR file.
- *Line 6* sets the user context for any subsequent RUN, CMD, or ENTRYPOINT instructions to the **spring** user. This enhances security by preventing the application from running as root.
- *Line 7* defines the command to be executed when the container starts. It runs the Java application by executing the JAR file.

First, let us compile and package the application as a Jar file:

```
1. ./mvnw package
```

Now, we can create a Docker image and tag it as **bookclub-svc**:

```
1. docker build -t bookclub-svc .
```

The above command initiates Docker's build process, searching for a Dockerfile in the current directory and sequentially processing each instruction within it. This process creates layers that together form the final image. This layered approach enables efficient caching and incremental builds, significantly speeding up subsequent image creations when only minor changes are made. The resulting image encapsulates the BookClub service application, its dependencies, and its runtime environment.

Next, we can run the image using the following command:

```
1. docker run --rm -p 8080:8080 -e "SPRING_PROFILES_ACTIVE=production" -t bookclub-svc
```

The above command initiates a new container instance based on the **bookclub-svc** image. The **--rm** flag ensures that the container is automatically removed when it exits, which is particularly useful during development. The **-p 8080:8080** option maps port 8080 from the container to port 8080 on the host machine, enabling external access to the application running inside the container. Without this port mapping, the application would not be reachable. Additionally, we pass the active Spring profile as an environment variable. Upon execution, you should see the following logs in the console:

```

2-bookclub-docker git:(master) ✘ docker run --rm -p 8080:8080 -e "SPRING_PROFILES_ACTIVE=production" -t bookclub-svc

:
: Spring Boot :: (v3.3.1)

2024-10-27T03:21:23.834Z INFO 1 --- [bookclub] [main] c.b.h.ch10.bookclub.BookclubApplication : Starting BookclubApplication v0.0.1-SNAPSHOT using Java 17.0.13 with PID 1 (/app.jar started by spring in
)
2024-10-27T03:21:23.837Z INFO 1 --- [bookclub] [main] c.b.h.ch10.bookclub.BookclubApplication : The following 1 profile is active: "production"
2024-10-27T03:21:26.113Z INFO 1 --- [bookclub] [main] s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data JPA repositories in DEFAULT mode.
2024-10-27T03:21:26.226Z INFO 1 --- [bookclub] [main] s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository scanning in 92 ms. Found 2 JPA repository interfaces.
2024-10-27T03:21:27.483Z INFO 1 --- [bookclub] [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port: 8080 (http)
2024-10-27T03:21:27.505Z INFO 1 --- [bookclub] [main] o.s.apache.catalina.core.StandardService : Starting service [Tomcat]
2024-10-27T03:21:27.506Z INFO 1 --- [bookclub] [main] o.s.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/10.1.25]
2024-10-27T03:21:27.570Z INFO 1 --- [bookclub] [main] o.s.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2024-10-27T03:21:27.573Z INFO 1 --- [bookclub] [main] w.s.e.WebApplicationContext : Root WebApplicationContext: initialization completed in 3601 ms
2024-10-27T03:21:28.183Z INFO 1 --- [bookclub] [main] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Starting...
2024-10-27T03:21:28.495Z INFO 1 --- [bookclub] [main] com.zaxxer.hikari.pool.HikariPool : HikariPool-1 - Added connection conn0: url=jdbc:h2:mem:41dd059e3-f3e9-42af-bccc-1dd9613dcc50 (H2 2.2)
2024-10-27T03:21:28.582Z INFO 1 --- [bookclub] [main] org.flywaydb.core.FlywayExecutor : Database: jdbc:h2:mem:41dd059e3-f3e9-42af-bccc-1dd9613dcc50 (H2 2.2)
2024-10-27T03:21:28.589Z WARN 1 --- [bookclub] [main] o.f.c.internal.database.base.Database : Flyway upgrade recommended: H2 2.2.224 is newer than this version of Flyway and support has not been test
1. The latest supported version of H2 is 2.2.228.
2024-10-27T03:21:28.626Z INFO 1 --- [bookclub] [main] o.f.c.i.s.JdbcTableSchemaHistory : Schema history table "PUBLIC"."flyway_schema_history" does not exist yet
2024-10-27T03:21:28.629Z INFO 1 --- [bookclub] [main] o.f.core.internal.command.DdlValidate : Successfully validated 2 migrations (execution time 00:00:025s)
2024-10-27T03:21:28.644Z INFO 1 --- [bookclub] [main] o.f.c.i.s.JdbcTableSchemaHistory : Creating Schema History table "PUBLIC"."flyway_schema_history" ...
2024-10-27T03:21:28.721Z INFO 1 --- [bookclub] [main] o.f.core.internal.command.DdlMigrate : Current version of schema "PUBLIC": <> Empty Schema >>
2024-10-27T03:21:28.738Z INFO 1 --- [bookclub] [main] o.f.core.internal.command.DdlMigrate : Migrating Schema "PUBLIC" to version "1 - Init"
2024-10-27T03:21:28.779Z INFO 1 --- [bookclub] [main] o.f.core.internal.command.DdlMigrate : Migrating schema "PUBLIC" to version "2 - Add initial book data"
2024-10-27T03:21:28.819Z INFO 1 --- [bookclub] [main] o.f.core.internal.command.DdlMigrate : Successfully applied 2 migrations to schema "PUBLIC", now at version v2 (execution time 00:00:058s)
2024-10-27T03:21:29.007Z INFO 1 --- [bookclub] [main] o.h.HibernateJpaInterposerUtil$LogHelper : HHHHHHHH04: PROCESSING PERSISTENCEUNITINFO [Name: default]
2024-10-27T03:21:29.168Z INFO 1 --- [bookclub] [main] org.hibernate.Version : HHHHHHHH042: Hibernate ORM core version 6.2.5.Final
2024-10-27T03:21:29.248Z INFO 1 --- [bookclub] [main] o.h.i.Registration : HHHHHHHH026: Second-level cache disabled
2024-10-27T03:21:29.827Z INFO 1 --- [bookclub] [main] o.s.o.j.p.SpringPersistenceUnitInfo : No LoadTimeWeaver setup: ignoring JPA class transformer
2024-10-27T03:21:31.593Z INFO 1 --- [bookclub] [main] o.n.e.j.p.i.JtaPlatformInitiator : HHHHHHHH049: No JTA platform available (set 'hibernate.transaction.jta.platform' to enable JTA platform int
ortion)
2024-10-27T03:21:31.595Z INFO 1 --- [bookclub] [main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
2024-10-27T03:21:32.357Z WARN 1 --- [bookclub] [main] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, database queries may be performed during view rendering. Explicitly configure spring.jpa.open-in-view to disable this warning
2024-10-27T03:21:33.807Z INFO 1 --- [bookclub] [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 8080 (http) with context path '/'
2024-10-27T03:21:34.012Z INFO 1 --- [bookclub] [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port: 8081 (http)
2024-10-27T03:21:34.072Z INFO 1 --- [bookclub] [main] o.s.apache.catalina.core.StandardService : Starting service [Tomcat]
2024-10-27T03:21:34.073Z INFO 1 --- [bookclub] [main] o.s.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/10.1.25]
2024-10-27T03:21:34.099Z INFO 1 --- [bookclub] [main] o.s.c.c.C.[Tomcat-1].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2024-10-27T03:21:34.099Z INFO 1 --- [bookclub] [main] w.s.e.WebApplicationContext : Root WebApplicationContext: initialization completed in 280 ms
2024-10-27T03:21:34.158Z INFO 1 --- [bookclub] [main] o.s.b.a.e.web.EndpointLinksResolver : Exposing 3 endpoints beneath base path '/actuator'
2024-10-27T03:21:34.312Z INFO 1 --- [bookclub] [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 8081 (http) with context path '/'
2024-10-27T03:21:34.344Z INFO 1 --- [bookclub] [main] c.b.h.ch10.bookclub.BookclubApplication : Started BookclubApplication in 11.47 seconds (process running for 12.565)

```

Figure 10.4: Application starts as a Docker container

You can reuse the same Postman request from the previous section to test this application

In production systems, while it is possible to run containers directly using Docker, the common practice is to use a container orchestration solution like Docker Swarm or Kubernetes.

In the exercise we just completed, we used two separate tools: Maven to compile and package the application as a JAR file and Docker to build the Docker image. This two-phase build process is often preferred by developers. However, if you would like to streamline the process and perform everything in one step, you can easily do so. The application can be compiled, packaged as a JAR file, and built into a Docker image using the following Docker command:

```
1. ./mvnw spring-boot:build-image -Dspring-boot.build-image.imageName=bookclub-svc
```

In addition to the build plugin provided by the Spring project, the Jib plugin created by Google is also quite popular among developers. You can enable this plugin by adding the following dependency to the pom file of your project:

1. <plugin>
2. <groupId>com.google.cloud.tools</groupId>

```
3. <artifactId>jib-maven-plugin</artifactId>
4. <version>3.4.1</version>
5. <configuration>
6.   <to>
7.     <image>bookclub-svc</image>
8.   </to>
9. </configuration>
10. </plugin>
```

Then, you can use the following command to build the Docker image:

```
1. ./mvnw clean package jib:dockerBuild
```

In this section, we have learned how to create Docker images for your Spring Boot projects. In the next section, we will look at how to create native images for Spring Boot applications using GraalVM.

## Spring Boot with GraalVM

GraalVM is a high-performance **Java Development Kit (JDK)** designed to improve the performance and resource efficiency of Java applications. It addresses several challenges faced by traditional Java applications, such as slow start-up times, high memory consumption, and performance limitations in containerized environments. A key feature of GraalVM is its ability to compile Java applications ahead of time into standalone native binaries, resulting in significantly faster start-up times, reduced memory footprint, and improved CPU utilization. Additionally, GraalVM offers polyglot capabilities, allowing seamless integration of multiple programming languages, such as JavaScript, within a single application.

Spring Boot 3.0 introduced full support for GraalVM Native Image, marking a significant step forward in Spring Boot application development. This integration allows developers to easily compile their Spring Boot applications ahead of time using GraalVM. Previously incubated in the Spring Native project, this support is now fully integrated into Spring Boot's core.

The **spring-boot-starter-parent**, which we have been using throughout this book, has declared a Maven profile called the native profile. This profile configures the executions required to create a native image. You can activate profiles using the **-P** flag on the command line. Additionally, you need to

add the following to the plugin section of your project's **pom.xml** file:

1. <plugin>
  2. <groupId>org.graalvm.buildtools</groupId>
  3. <artifactId>native-maven-plugin</artifactId>
  4. </plugin>

Then, you can use the following Maven command to build your Spring Boot project as native images:

- ```
1. ./mvnw -Pnative native:compile
```

Sometimes, running the above command results in a build failure, as given in the following figure. This is a very common error that indicates that you have not configured GraalVM supported JDK. You can solve this by installing GraalVM or Liberica Native Image Kit as per your operating system.

**Figure 10.5:** Error message caused using a JDK without GraalVM support

The native image created during the build process is available in the target directory of your project, and you can execute the native image as you would execute any other native application. A couple of examples for each of the operating systems are given as follows:

For Windows:

- ## 1. target\bookclub-svc.exe

For Linux:

1. ./target/bookclub-svc

For macOS:

1. [/target/bookclub-svc](#)

In this section, we looked at how to generate native images for Spring Boot applications. In the next section, we will look into how we can use Jetty and

Undertow as embedded web servers with Spring Boot instead of Tomcat.

## Use Jetty and Undertow with Spring Boot

Spring Boot uses Tomcat as the default embedded web server, and Tomcat is an excellent choice due to its security, lightweight nature, and great performance. However, Spring Boot does not limit you to Tomcat; you can also use other web servers, such as Jetty and Undertow. In this section, let us explore how easy it is to switch from Tomcat to another embedded web server, starting with Jetty.

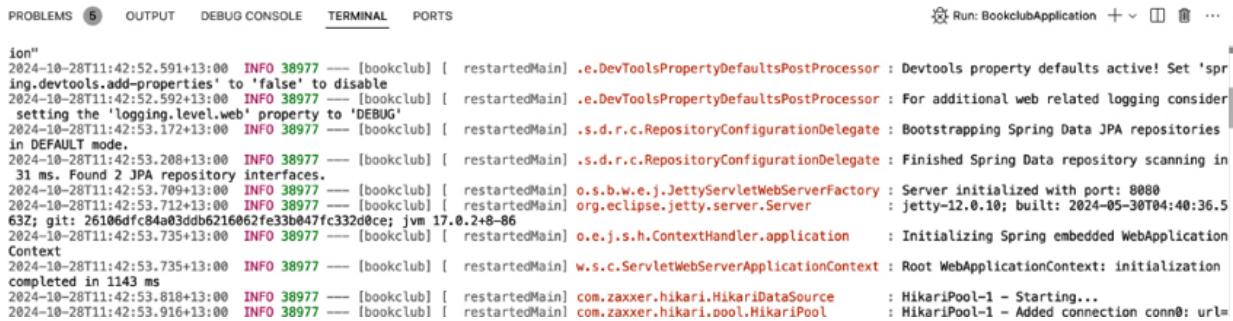
Jetty is a highly versatile and lightweight open-source web server and servlet container developed by the Eclipse Foundation. It is known for its efficiency, modularity, and flexibility, making it a popular choice for both standalone server deployments and embedded applications within larger software frameworks. Jetty supports the latest web standards, including HTTP/2, WebSocket protocols, and non-blocking I/O.

To replace Tomcat with Jetty, all you need to do is remove **spring-boot-starter-tomcat** from your project's **pom.xml** and add **spring-boot-starter-jetty**. Spring Boot's auto-configuration mechanism will handle the rest. You can achieve this with the following code:

```
1. <dependency>
2.   <groupId>org.springframework.boot</groupId>
3.   <artifactId>spring-boot-starter-web</artifactId>
4.   <exclusions>
5.     <exclusion>
6.       <groupId>org.springframework.boot</groupId>
7.       <artifactId>spring-boot-starter-tomcat</artifactId>
8.     </exclusion>
9.   </exclusions>
10.  </dependency>
11.  <dependency>
12.    <groupId>org.springframework.boot</groupId>
13.    <artifactId>spring-boot-starter-jetty</artifactId>
14.  </dependency>
15.
16.
```

Now, if you run the application, you should be able to see Jetty has started

instead of Tomcat as shown in the following figure:



The screenshot shows a terminal window with the title 'Run: BookclubApplication'. The window displays a series of INFO log messages from a Spring Boot application named 'bookclub'. The logs indicate the application is starting up, with various components like 'DevToolsPropertyDefaultsPostProcessor', 'RepositoryConfigurationDelegate', and 'ServletWebServerApplicationContext' being initialized. It also shows the configuration of a Jetty server on port 8080, the use of Hikari for database connections, and the scanning of JPA repository interfaces. The logs are timestamped from 2024-10-28T11:42:53.208+13:00 to 2024-10-28T11:42:53.735+13:00.

```
ion"
2024-10-28T11:42:52.591+13:00  INFO 38977 --- [bookclub] [ restartedMain] .e.DevToolsPropertyDefaultsPostProcessor : Devtools property defaults active! Set 'spring.devtools.add-properties' to 'false' to disable
2024-10-28T11:42:52.592+13:00  INFO 38977 --- [bookclub] [ restartedMain] .e.DevToolsPropertyDefaultsPostProcessor : For additional web related logging consider setting the 'logging.level.web' property to 'DEBUG'
2024-10-28T11:42:53.172+13:00  INFO 38977 --- [bookclub] [ restartedMain] .s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data JPA repositories in DEFAULT mode.
2024-10-28T11:42:53.208+13:00  INFO 38977 --- [bookclub] [ restartedMain] .s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository scanning in 31 ms. Found 2 JPA repository interfaces.
2024-10-28T11:42:53.709+13:00  INFO 38977 --- [bookclub] [ restartedMain] org.s.b.w.e.j.JettyServletWebServerFactory : Server initialized with port: 8080
2024-10-28T11:42:53.712+13:00  INFO 38977 --- [bookclub] [ restartedMain] org.eclipse.jetty.server.Server : jetty-12.0.10; built: 2024-05-30T04:40:36.563Z; git: 26106dfc84a03db6216062fe33b47fc332d0ce; jvm: 17.0.2+8-86
2024-10-28T11:42:53.735+13:00  INFO 38977 --- [bookclub] [ restartedMain] o.e.j.s.h.ContextHandler.application : Initializing Spring embedded WebApplicationContext
2024-10-28T11:42:53.735+13:00  INFO 38977 --- [bookclub] [ restartedMain] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 1143 ms
2024-10-28T11:42:53.818+13:00  INFO 38977 --- [bookclub] [ restartedMain] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Starting...
2024-10-28T11:42:53.916+13:00  INFO 38977 --- [bookclub] [ restartedMain] com.zaxxer.hikari.HikariPool : HikariPool-1 - Added connection conn0: url=
```

Figure 10.6: Jetty as the embedded web server

Next, let us look at how to use Undertow as the embedded web server with Spring Boot applications.

Undertow is a high-performance, fully embeddable web server developed by Red Hat as part of the JBoss project. Designed to be lightweight and flexible, Undertow focuses on providing excellent performance for both blocking and non-blocking tasks. It fully supports the Java Servlet 3.1 specification, as well as HTTP/2 and WebSocket protocols.

As with Jetty, Undertow can be configured as the embedded web server by replacing the **spring-boot-starter-tomcat** dependency with **spring-boot-starter-undertow**, as shown in the following code. Everything else is taken care of by Spring Boot's auto-configuration mechanism.

1. <dependency>
2. <groupId>org.springframework.boot</groupId>
3. <artifactId>spring-boot-starter-web</artifactId>
4. <exclusions>
5. <exclusion>
6. <groupId>org.springframework.boot</groupId>
7. <artifactId>spring-boot-starter-tomcat</artifactId>
8. </exclusion>
9. </exclusions>
10. </dependency>
11. <dependency>
12. <groupId>org.springframework.boot</groupId>
13. <artifactId>spring-boot-starter-undertow </artifactId>
14. </dependency>
- 15.
- 16.

Once you run the application, you should be able to see that Spring Boot has configured and started Undertow as the embedded web server instead of Tomcat:



```
PROBLEMS 5 OUTPUT DEBUG CONSOLE TERMINAL PORTS Run: BookclubApplication + ▾ □ ☰ ...  
2024-10-28T11:54:28.322+13:00 INFO 49942 — [bookclub] [ restartedMain] o.s.b.w.e.undertow.UndertowWebServer : Undertow started on port 8080 (http) with c  
ontext path '/'  
2024-10-28T11:54:28.373+13:00 INFO 49942 — [bookclub] [ restartedMain] io.undertow.servlet : Initializing Spring embedded WebApplication  
Context  
2024-10-28T11:54:28.373+13:00 INFO 49942 — [bookclub] [ restartedMain] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization  
completed in 49 ms  
2024-10-28T11:54:28.383+13:00 INFO 49942 — [bookclub] [ restartedMain] o.s.b.a.e.web.EndpointLinksResolver : Exposing 3 endpoints beneath base path '/ac  
tuator'  
2024-10-28T11:54:28.413+13:00 INFO 49942 — [bookclub] [ restartedMain] io.undertow : starting server: Undertow - 2.3.13.Final  
2024-10-28T11:54:28.415+13:00 INFO 49942 — [bookclub] [ restartedMain] o.s.b.w.e.undertow.UndertowWebServer : Undertow started on port 8081 (http) with c  
ontext path '/'  
2024-10-28T11:54:28.425+13:00 INFO 49942 — [bookclub] [ restartedMain] c.b.h.ch10.bookclub.BookclubApplication : Started BookclubApplication in 3.643 second  
s (process running for 4.082)  
2024-10-28T11:54:31.099+13:00 INFO 49942 — [bookclub] [on(2)-127.0.0.1] io.undertow.servlet : Initializing Spring DispatcherServlet 'disp  
atcherServlet'  
2024-10-28T11:54:31.100+13:00 INFO 49942 — [bookclub] [on(2)-127.0.0.1] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'  
2024-10-28T11:54:31.102+13:00 INFO 49942 — [bookclub] [on(2)-127.0.0.1] o.s.web.servlet.DispatcherServlet : Completed initialization in 2 ms
```

Figure 10.7: Undertow as the embedded web server

In the next section, we will look at how to configure SSL for your Spring Boot applications.

## Configuring SSL for Spring Boot applications

So far, in almost every application we have developed in this book, we have been using plain HTTP because it is straightforward and easy to debug at the transport layer if required. However, when you deploy your applications into production, you must configure and expose your applications and services only over HTTPS. We do not expose any production HTTP endpoints over plain HTTP.

In real-world situations, your organization needs to purchase a CA-signed certificate from trusted **certificate authorities (CAs)** such as DigiCert, Sectigo, or GoDaddy. Alternatively, you can obtain a free automated certificate from *Let's Encrypt*. However, in this section, we will generate our own certificate for testing. These types of certificates, generated without involvement from a trusted certificate authority, are known as **self-signed certificates** and are perfectly fine for non-production use.

First, let us generate a self-signed certificate using Java's keytool by running the following command:

1. keytool -genkeypair -keyalg RSA -keysize 2048 -storetype PKCS12 -keystore cert.p12 -validity 365

The above command prompts you to a couple of questions and generates the certificate, as shown in the following figure:

```

PROBLEMS 3 OUTPUT DEBUG CONSOLE TERMINAL PORTS
● → 7-bookclub-SSL git:(master) ✘ keytool -genkeypair -keyalg RSA -keysize 2048 -storetype PKCS12 -keystore cert.p12 -validity 365
Enter keystore password:
Re-enter new password:
What is your first and last name?
[Unknown]: Sagara
What is the name of your organizational unit?
[Unknown]: BookClub
What is the name of your organization?
[Unknown]: BookClub
What is the name of your City or Locality?
[Unknown]: Auckland
What is the name of your State or Province?
[Unknown]: Auckland
What is the two-letter country code for this unit?
[Unknown]: nz
Is CN=Sagara, OU=BookClub, O=BookClub, L=Auckland, ST=Auckland, C=nz correct?
[no]: yes

Generating 2,048 bit RSA key pair and self-signed certificate (SHA256withRSA) with a validity of 365 days
for: CN=Sagara, OU=BookClub, O=BookClub, L=Auckland, ST=Auckland, C=nz
○ → 7-bookclub-SSL git:(master) ✘

```

**Figure 10.8:** self-signed certificate generation using Java keytool

Once it generates the certificate, you can just add it to the resource directory. Then, we can update our **application.properties** file to enable SSL for our application by adding the following properties:

1. `# SSL`
2. `server.port=8443`
3. `server.ssl.key-store=classpath:cert.p12`
4. `server.ssl.key-store-password=123456`
5.
6. `# JKS or PKCS12`
7. `server.ssl.keyStoreType=PKCS12`

Once you add this property, Spring Boot makes sure to disable HTTP transport and only enable HTTPS for your application. When you run the application again, you should be able to see the console logs, as shown in the following figure:

```

PROBLEMS 5 OUTPUT DEBUG CONSOLE TEST RESULTS TERMINAL PORTS
Run: BookclubApplication + - × ...
rsistence unit 'default'
2024-10-28T12:12:48.087+13:00  WARN 64719 --- [bookclub] [ restartedMain] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, database queries may be performed during view rendering. Explicitly configure spring.jpa.open-in-view to disable this warning
2024-10-28T12:12:48.557+13:00  INFO 64719 --- [bookclub] [ restartedMain] o.s.b.d.a.OptionalLiveReloadServer : LiveReload server is running on port 35729
2024-10-28T12:12:48.744+13:00  INFO 64719 --- [bookclub] [ restartedMain] o.a.t.util.net.NioEndpoint$certificate : Connector [https-jsse-nio-8443], TLS virtual host [__default__], certificate type [UNDEFINED] configured from keystore [/Users/sagara/.keystore] using alias [tomcat] with trust store [null]
2024-10-28T12:12:48.752+13:00  INFO 64719 --- [bookclub] [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 8443 (https) with context path '/'
2024-10-28T12:12:48.819+13:00  INFO 64719 --- [bookclub] [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port 8081 (https)
2024-10-28T12:12:48.820+13:00  INFO 64719 --- [bookclub] [ restartedMain] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2024-10-28T12:12:48.826+13:00  INFO 64719 --- [bookclub] [ restartedMain] o.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/10.1.25]
2024-10-28T12:12:48.827+13:00  INFO 64719 --- [bookclub] [ restartedMain] o.a.c.c.C.[Tomcat-1].[/] : Initializing Spring embedded WebApplication Context
2024-10-28T12:12:48.827+13:00  INFO 64719 --- [bookclub] [ restartedMain] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 73 ms
2024-10-28T12:12:48.836+13:00  INFO 64719 --- [bookclub] [ restartedMain] o.s.b.a.e.web.EndpointLinksResolver : Exposing 3 endpoints beneath base path '/ac'

```

**Figure 10.9:** Application starts with SSL

Now, if you try to reach the HTTP endpoint using Postman, you will realize that it is no longer available; instead, now, you should use the HTTPS endpoint as follows:

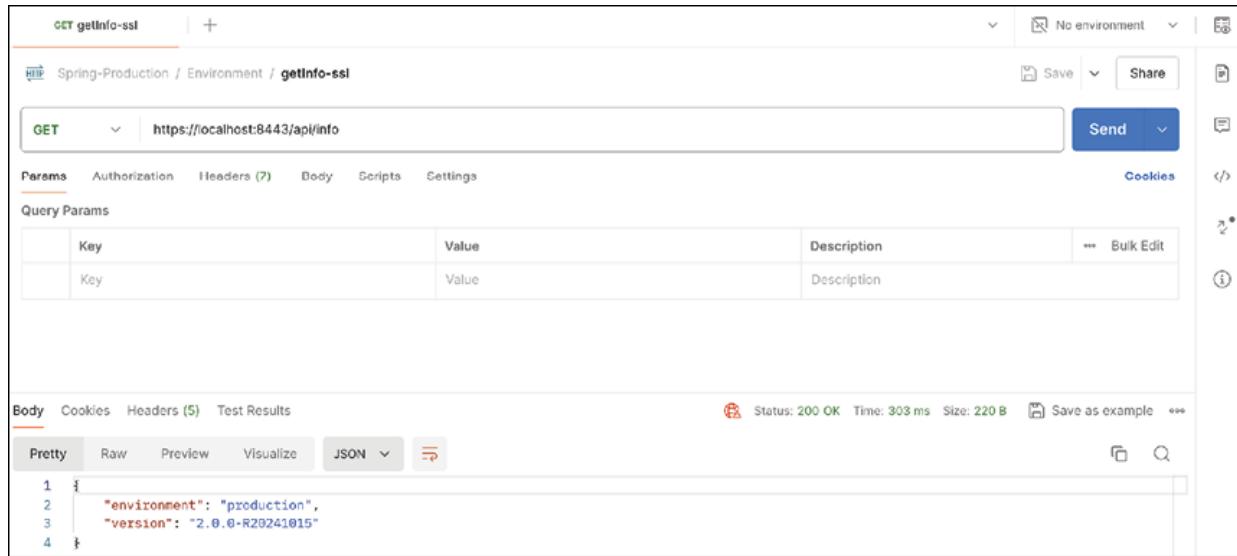


Figure 10.10: Calling info endpoint using HTTPS

## Identity provider integration for Spring Boot

Earlier in this book, we discussed the Spring Security project, which can be used to secure Spring Boot web applications and services. We implemented security for a few web applications and RESTful services using Spring Security features. However, if you recall, we used **InMemoryUserDetailsManager** with hard-coded credentials to test the security controls of our sample applications. Naturally, in production applications, we cannot rely on in-memory databases or hard-coded credentials.

One option would be to use **JdbcUserDetailsManager** instead of **InMemoryUserDetailsManager**, where user profiles and hashed passwords are stored in a database, just like other business data. Alternatively, you can develop a custom **UserDetailsService** tailored to your own database schema. During authentication, Spring Security will attempt to locate the provided username in the database to validate the password. This approach requires a mechanism to persist user profiles with passwords before authentication, a process commonly known as user onboarding. To support onboarding, you could develop a user sign-up page (similar to a sign-in page) with persistence logic so that users can self-register. This approach, known as user self-onboarding, is typical in customer-centric applications. Alternatively, you could provide a secure API endpoint to facilitate user

onboarding from another system.

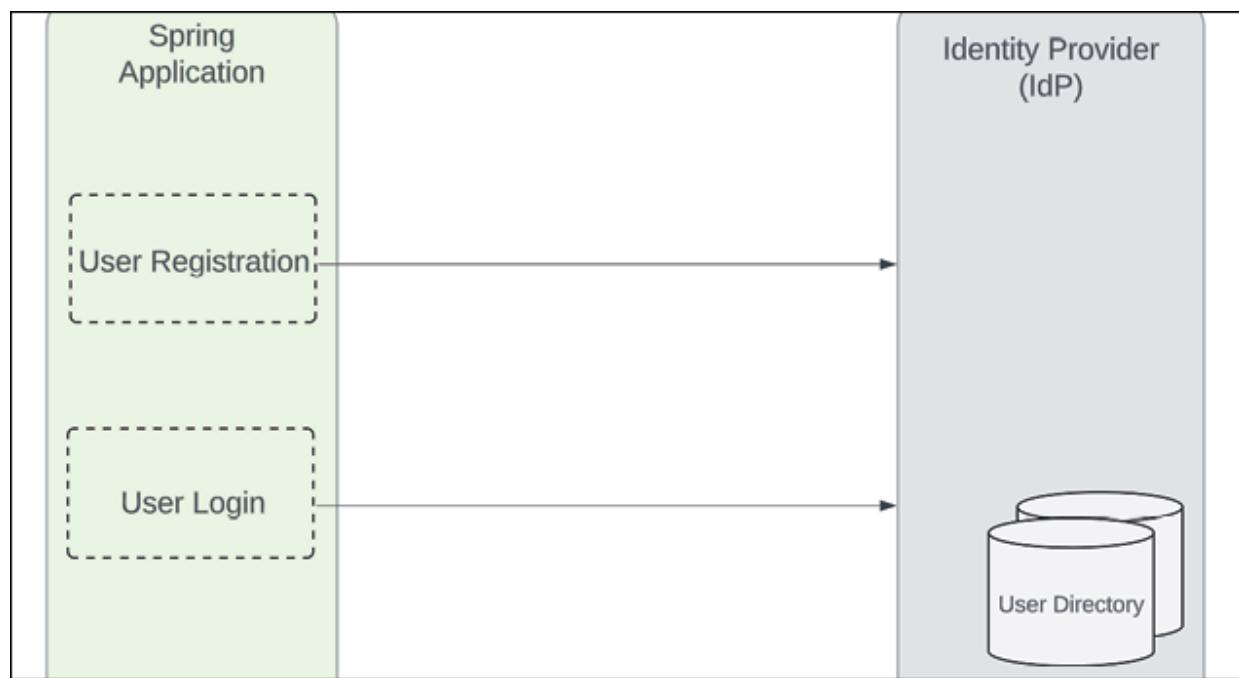
In summary, once you have implemented **JdbcUserDetailsManager** and user onboarding features, you will have the basic capability to onboard customers and allow them to secure access to your application after authentication. However, in real-world applications, this is just the beginning. Here are a few practical situations you will need to address:

- Over time, users may forget their password, username, or both, necessitating account recovery features like **forgot password** or **forgot username** options. Account recovery often relies on additional verifications, such as email or phone number verification.
- As privacy becomes increasingly important, much of the data collected from users is categorized as **personally identifiable information (PII)**. Special care is needed when collecting and storing PII. You should provide users with the ability to view and update their profiles, download a copy of their profile data, or even delete their profiles if required.
- So far in our discussion, we have focused on password-based authentication. However, password-based authentication alone may not be sufficient for securing your application. To provide strong authentication, you need to consider implementing **multi-factor authentication (MFA)**, which can be based on SMS, email, push notifications, or passkeys.
- Enabling users to log in from their preferred social platforms, such as Google or Facebook, can greatly improve the user experience of your application. This is known as social login. In addition to that, instead of presenting a lengthy registration form, you can use a social platform to create the initial user profile, known as social onboarding.

While it is possible to implement and maintain these features by yourself, you should carefully consider whether this is the best investment of your time and resources. Specialized proprietary, open-source, and cloud-based solutions exist that provide all the above-mentioned user management and strong authentication features out of the box, with options for customization and rebranding. These are commonly referred to as **identity and access management (IAM)** solutions. However, the intention of this discussion is not to discourage you from developing identity management features

yourself but to make you aware of the out-of-the-box options available so that you can make an informed decision.

That being said, identity solutions are not a silver bullet. While they provide every user management and strong authentication capability that you could ever require, they introduce complexity to your application architecture and add latency to your login flows. The following figure illustrates the application architecture after introducing the identity solution. From here onward, we will use the term **identity provider (IdP)** as it clearly represents their role within our application architecture.



*Figure 10.11: Identity provider integration with an application*

Here are the typical high-level steps involved when an application interacts with an identity provider:

1. The end user accesses your application through a browser and clicks the **Login** button, initiating the authentication process.
2. The application recognizes the user's intent to authenticate and redirects them to the identity provider. This redirection generally occurs through an HTTP redirect, though other mechanisms may be used. This is commonly referred to as an authentication request. The application includes identification data in this request, allowing the identity provider to recognize which application is requesting authentication on

behalf of the user.

3. The identity provider presents a login form to the user and may challenge them with **multi-factor authentication (MFA)** options, such as SMS or email OTP, push notification, or passkeys. This process can vary from a single step to multiple steps, depending on the required security level.
4. Once authentication is complete, the identity provider sends the user back to the application, along with a security token. The application then validates this token to confirm it was issued by the trusted identity provider. The token may contain user details directly, or it could be a reference token that the application exchanges with the identity provider to retrieve user information.

Spring Security supports two commonly used authentication protocols: **Security Assertion Markup Language (SAML)** and **OpenID Connect (OIDC)**:

- **SAML**: This is an older protocol that uses XML and operates by sending assertions between an IdP, which verifies the user's identity, and a **service provider (SP)**, which provides access to the application the user wants to use. SAML is widely used in enterprise environments.
- **OIDC**: Built on top of OAuth 2.0, OIDC is a newer protocol designed for modern web and mobile applications. It uses simpler **JSON Web Tokens (JWTs)** to share information about user identity. OIDC not only handles authentication but also allows applications to access additional user details, making it ideal for consumer applications and social logins. OIDC is more popular in consumer-facing applications.

In this book, we focus only on OIDC. For information on using SAML with Spring Boot, refer to the Spring Security documentation.

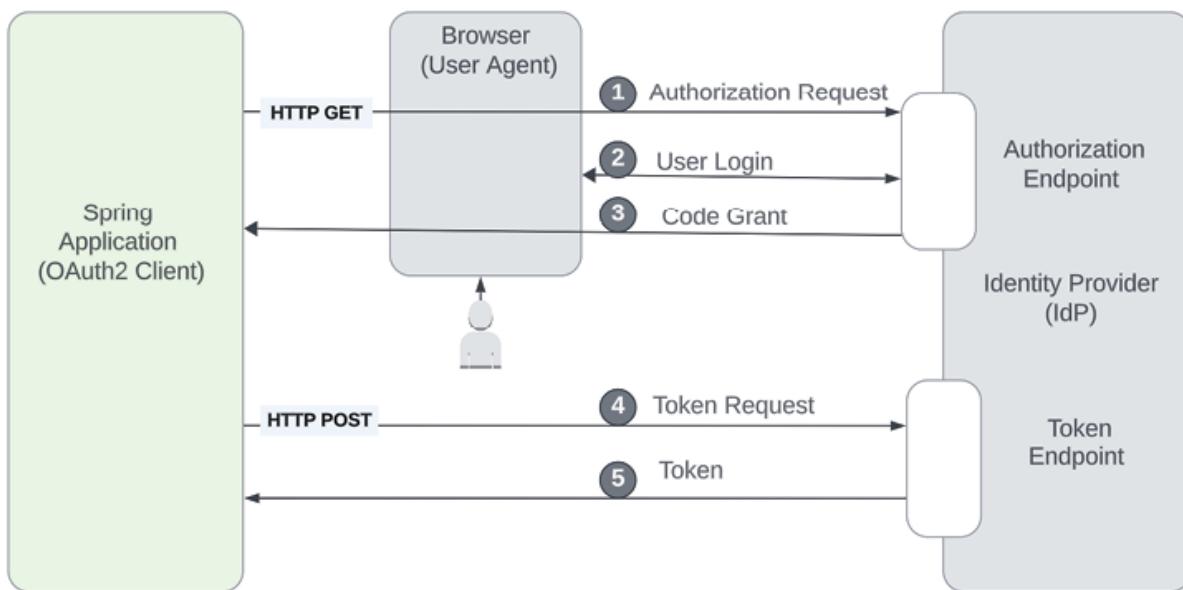
OIDC was developed as an extension of the popular OAuth 2.0 framework, which is an access delegation protocol. According to the OAuth 2.0 standard, there are several mechanisms, known as grant types, to retrieve security tokens. In this section, we will use the authorization code grant type or, simply, the code grant type. This grant type is the most commonly used and mostly recommended for real-world applications, and some of the other grant types are no longer recommended.

Since we have already discussed the generic authentication flow, we will

now look at the OAuth2/OIDC authorization code grant flow, as it is essential to understand it. Before we go further, it is important to clarify some key terms used in OIDC and OAuth2 protocols to avoid any confusion:

- **End user:** Often referred to as the subject.
- **Application:** Known as the **Relying Party (RP)** in OIDC and as the client in OAuth2.
- **Identity provider:** Called the **OpenID Provider (OP)** in OIDC and the **Authorization Server (AS)** in OAuth2.

Now, let us look at the OAuth2/OIDC authorization code grant flow shown in the following figure and description of each step:



**Figure 10.12: OAuth2 authorization code grant type flow**

1. The end user accesses your application through a browser and clicks the **Login** button, initiating the authentication process.
2. The application recognizes the user's intent to authenticate and redirects them to the identity provider with an OIDC authorization request in URL-encoded format. This includes an identifier called **client\_id**, which is used by the identity provider to recognize the application. The application sends this authorization request via HTTP redirect to a specific endpoint on the identity provider, known as the authorization endpoint.
3. The identity provider validates the **client\_id** and other metadata and

then presents a login form to the user. Depending on the required security level, it may also challenge the user with MFA options. While the application can request a certain **level of authentication assurance (LoA)**, it is ultimately up to the identity provider to determine the authentication method.

4. Once authentication is complete, the identity provider redirects the user back to the application with a special token called the code grant, representing an authorization response. The code grant is simply a reference and contains no user details—this is important because it travels through the browser, where it is visible to the user.
5. After receiving the code grant, the application creates a token request and sends it directly to the identity provider. This is a server-to-server communication, so the browser and user are not involved. The application also needs to authenticate itself to the identity provider, typically by sending **client\_id** and **client\_secret** in an HTTP Authorization header, a process known as client authentication. Note that client authentication is completely different from user authentication and is a critical security step in both OIDC and OAuth2.
6. If client authentication succeeds and the code grant is valid, the identity provider responds with a security token, known as the **ID Token**, containing details about the authenticated user. This step is referred to as the token response.
7. The application validates the ID Token and may use the information it contains to establish a session for the user.

The ID Token is in JWT format. The following figure shows a sample JWT token. While it is theoretically possible to have a plain JWT, in practice, JWT tokens must either be signed by the identity provider (known as a **JWS**) or both signed and encrypted by the identity provider (known as a **JWE**).

```
{
  "isk": "7b8bfe9f4ed1396eb777b0c6d8e111af6e0c18b416074c6ee8ca3f424fcc6c66",
  "at_hash": "znkxxOhMFPQ6P9W74IJJMA",
  "sub": "3b7cbde5-bfc7-474a-b79e-4d9e03c4c55f",
  "amr": [
    "BasicAuthenticator"
  ],
  "iss": "https://api.asgardeo.io/t/bifrost/oauth2/token",
  "sid": "de96a883-9baf-409e-9711-eaa7ca841f51",
  "aud": "hyfl0ShhPybRU7zP2nnm3jEtJ4sa",
  "c_hash": "Jsg6vv4Q4lzpPmTtAoxImA",
  "nbf": 1697445072,
  "azp": "hyfl0ShhPybRU7zP2nnm3jEtJ4sa",
  "org_id": "332080fd-338e-4269-88e4-a565c98d74cb",
  "exp": 1697448672,
  "org_name": "bifrost",
  "iat": 1697445072,
  "jti": "3807f58a-70e3-41a8-96d5-1c394fc8cf55"
  "username": john_doe
}
}
```

*Figure 10.13: Sample ID Token*

The main advantage of using a JWT in an ID Token is that the application can self-validate the token using signature verification. Identity providers that support OIDC expose an endpoint called the JWKS endpoint, which the application can use to obtain a copy of the public certificate associated with the private key used by the identity provider to sign the ID Token. Once the identity provider's public key is obtained, the application can validate the ID Token itself. Additionally, if needed, the application can validate the ID Token and request further details by presenting it to a special endpoint on the identity provider called the UserInfo endpoint.

You can use open-source identity providers like Spring Authorization Server, Keycloak, and WSO2 Identity Server. Alternatively, cloud-based **identity as a service (IDaaS)** providers with free tiers, such as Okta, Auth0, Azure AD, and Asgardeo, are also available.

Spring Boot applications can be configured to work with any OIDC-supported identity provider. Spring Security provides predefined client properties for providers such as Google, GitHub, Facebook, and Okta.

As our next exercise, we will integrate our BookClub application with GitHub. First, visit the following GitHub documentation page and create a

GitHub application:

<https://docs.github.com/en/apps/oauth-apps/building-oauth-apps/creating-an-oauth-app>

As shown in the following figure, in the create app page, you will be asked to provide the application name and the callback URL and provide the following values:

**Name:** Any available name

**Callback URL:** <http://localhost:8080/login/oauth2/code/github>

Register new GitHub App

**GitHub App name \***

The name of your GitHub App.

[Write](#) [Preview](#) [Markdown supported](#)

This is displayed to users of your GitHub App

**Homepage URL \***

The full URL to your GitHub App's website.

**Identifying and authorizing users**

The full URL to redirect to after a user authorizes an installation.

[Read our Callback URL documentation for more information.](#)

[Add Callback URL](#)

**Callback URL**

 [Delete](#)

**Expire user authorization tokens**  
This will provide a `refresh_token` which can be used to request an updated access token when this access token expires.

**Request user authorization (OAuth) during installation**  
Requests that the installing user grants access to their identity during installation of your App

*Figure 10.14: GitHub application registration page*

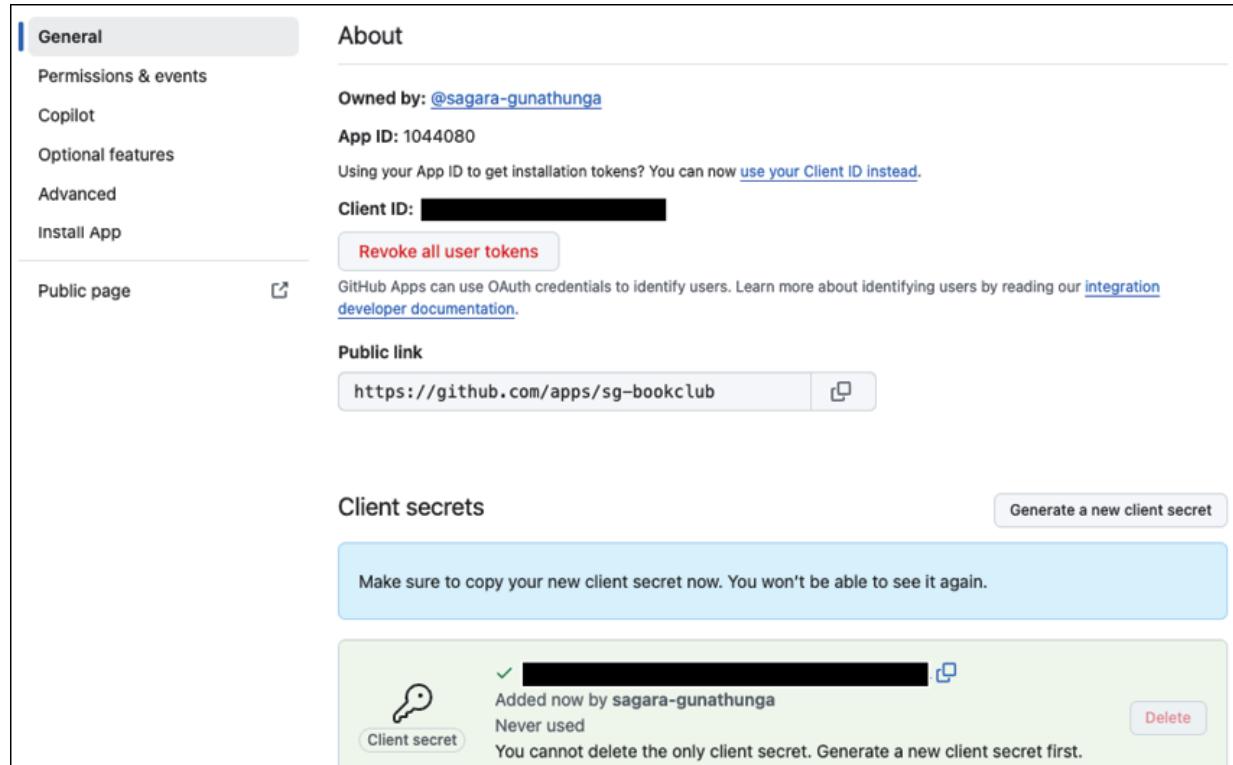
The most important piece of information here is the callback URL. This URL is an endpoint in your application where the identity provider redirects the user after successful authentication, along with the authorization code grant. According to the OAuth2 specification, the callback URL encoded in the authorization request from the application must match one of the callback URLs registered during application setup; otherwise, the identity

provider will return an error. Therefore, it is very important to ensure this value is accurate.

Spring Security uses the following format to define callback URLs:

**http://localhost:8080/login/oauth2/code/{provider-name}**

After creating the application, make sure to copy the client ID and client secret, as these will be needed in the next steps:



The screenshot shows the GitHub application metadata page. On the left, a sidebar lists 'General', 'Permissions & events', 'Copilot', 'Optional features', 'Advanced', and 'Install App'. The 'General' tab is selected. The main content area shows the following details:

- About**
  - Owned by: [@sagara-gunathunga](#)
  - App ID: 1044080
  - Using your App ID to get installation tokens? You can now [use your Client ID instead](#).
  - Client ID: [REDACTED]
  - [Revoke all user tokens](#)
- Public page**
  - GitHub Apps can use OAuth credentials to identify users. Learn more about identifying users by reading our [integration developer documentation](#).
- Public link**
  - <https://github.com/apps/sg-bookclub>
  - [Copy](#)
- Client secrets**
  - [Generate a new client secret](#)
  - Make sure to copy your new client secret now. You won't be able to see it again.
  - Client secret** (key icon)
    - Added now by sagara-gunathunga
    - Never used
    - You cannot delete the only client secret. Generate a new client secret first.
    - [Delete](#)

*Figure 10.15: GitHub application metadata*

Now, we are ready to modify our application. You can either edit the **bookclub-security** sample we developed in *Chapter 4, Building Spring MVC Web Applications*, or use the **bookclub-initial** project provided in the **ch10** directory of the companion GitHub repository for this book.

First, replace the **spring-boot-starter-security** dependency with the **spring-boot-starter-oauth2-client** dependency. The **spring-boot-starter-oauth2-client** is a Spring Boot starter that includes Spring Security's OAuth 2.0 Client support and provides auto-configuration to set up OAuth2/OpenID Connect clients. This starter allows you to easily integrate OAuth 2.0 client functionality into your Spring Boot application. Additionally, it includes the

necessary Spring Security dependencies, so you will not need to add Spring Security separately.

1. <dependency>
2. <groupId>org.springframework.boot</groupId>
3. <artifactId>spring-boot-starter-oauth2-client</artifactId>
4. </dependency>

If you are using your own project, make sure to remove the **SecurityConfiguration.java** file, as we do not need it.

Next, add the following properties to the **application.properties** file; you need to replace the placeholders with the values you copied previously when creating the GitHub application:

1. *#OAuth Application Properties*
2. `spring.security.oauth2.client.registration.github.`  
`client-id=[Client-ID]`
3. `spring.security.oauth2.client.registration.`  
`github.client-secret=[Client-Secret]`

We can also add the following properties to the **application.properties** files so that we can clearly see the content of the token and OIDC requests:

1. `logging.level.org.springframework.security.web=DEBUG`
2. `logging.level.org.springframework.security.oauth2.client.web=DEBUG`
3. `logging.level.org.springframework.security.web.context=INFO`

Finally, we need to modify the **CommonAttributesInterceptor** class because now we have to get user information from the token. The updated code is as follows:

1. `package com.bpb.hssb.ch4.bookclub.config;`
- 2.
3. `import org.springframework.security.core.Authentication;`
4. `import org.springframework.security.core.context.`  
`SecurityContextHolder;`
5. `import org.springframework.security.oauth2.client.`  
`authentication.OAuth2AuthenticationToken;`
6. `import org.springframework.web.servlet.HandlerInterceptor;`
7. `import org.springframework.web.servlet.ModelAndView;`
- 8.
9. `import jakarta.servlet.http.HttpServletRequest;`
10. `import jakarta.servlet.http.HttpServletResponse;`
- 11.
12. `public class CommonAttributesInterceptor`  
`implements HandlerInterceptor {`

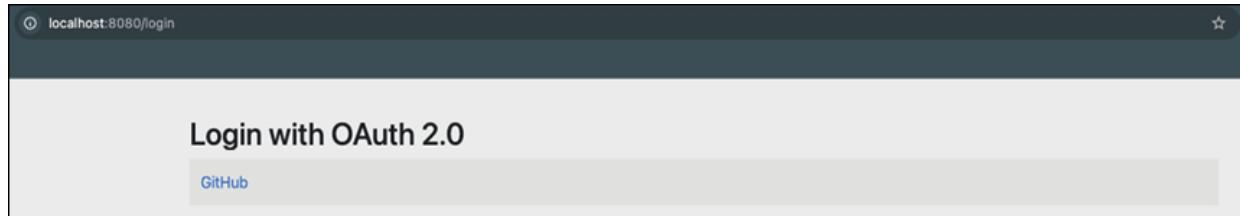
```

13.
14. @Override
15. public void postHandle(HttpServletRequest request,
   HttpServletResponse response, Object handler,
16.           ModelAndView modelAndView) throws Exception {
17.     if (modelAndView != null) {
18.         setUserAtributes(modelAndView);
19.     }
20. }
21.
22. private void setUserAtributes(ModelAndView modelAndView) {
23.     Authentication authentication =
   SecurityContextHolder.getContext().getAuthentication();
24.     if (authentication != null &&
   authentication.isAuthenticated()) {
25.         if (authentication instanceof
   OAuth2AuthenticationToken) {
26.             OAuth2AuthenticationToken token =
   (OAuth2AuthenticationToken) authentication;
27.             modelAndView.addObject("user",
   token.getPrincipal().getAttribute("name"));
28.         }
29.     }
30. }
31. }

```

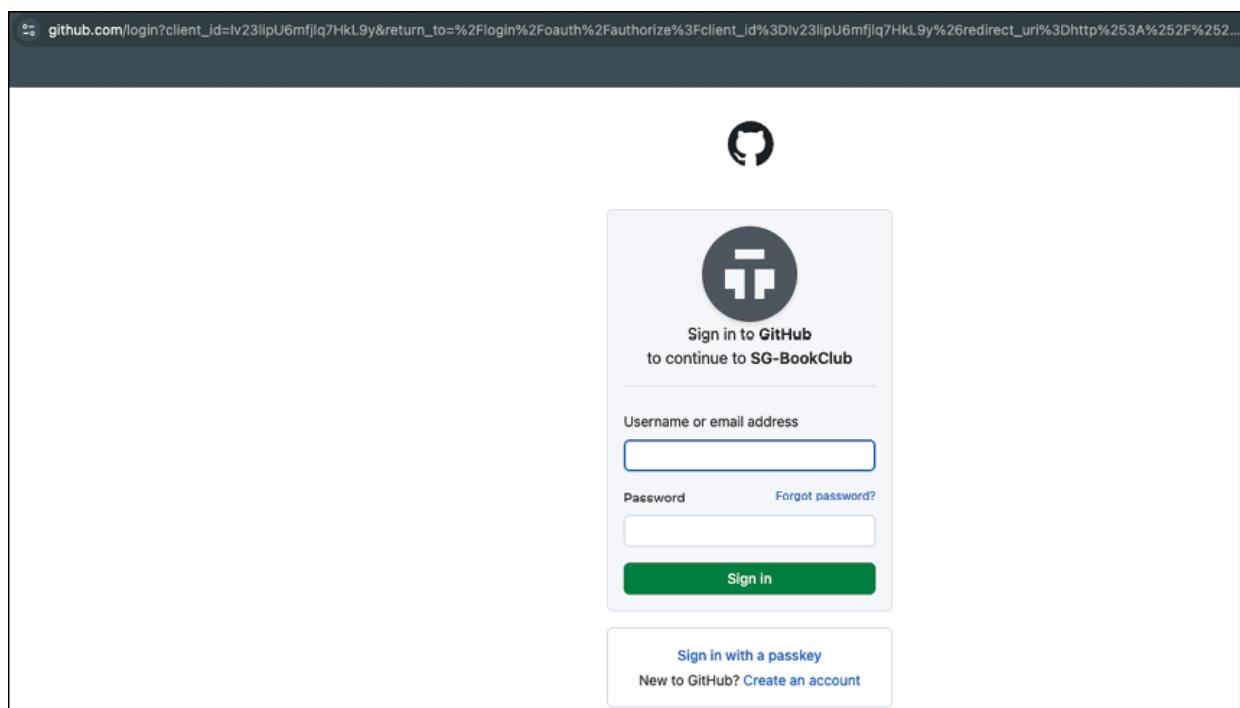
In the above code, the current **Authentication** object is retrieved from the **SecurityContextHolder**. If an authentication exists and is authenticated, the code checks if it is an instance of **OAuth2AuthenticationToken**. If true, the authentication is cast to **OAuth2AuthenticationToke**, and the user's name is extracted from the principal's attributes. This name is then added to the **modelAndView** object to display in a view.

Now, you can run the application and visit the login page **at <http://localhost:8080/login>**. Upon accessing this URL, you should see the login screen generated by Spring Boot, where GitHub is listed as the identity provider:



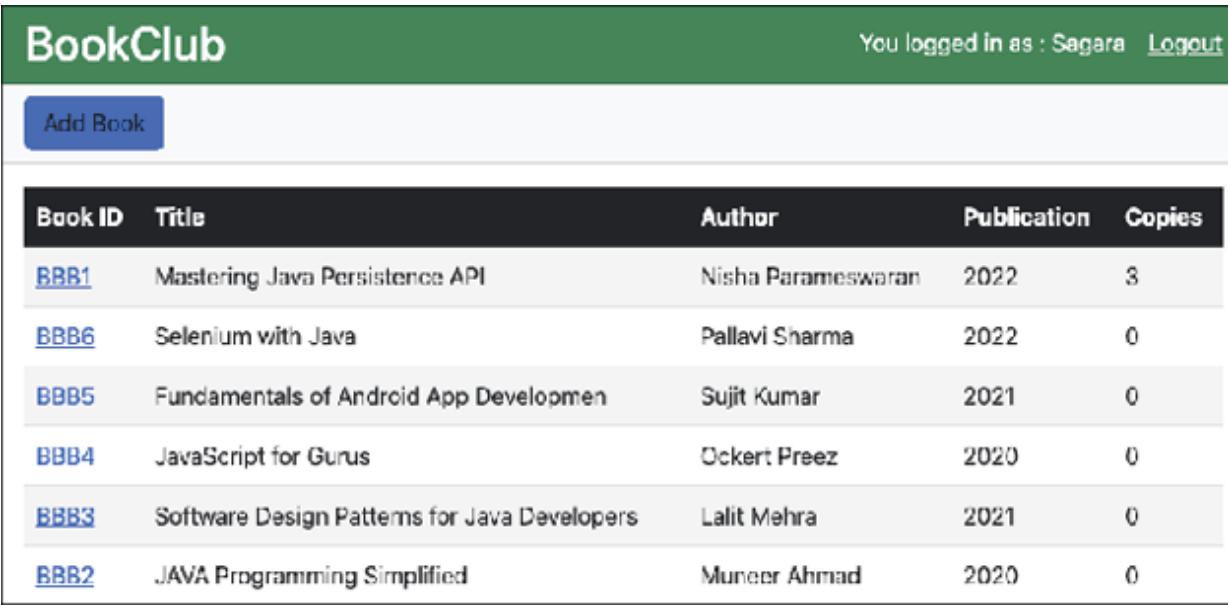
*Figure 10.16: Default login option page generated by Spring Security*

Once you click the GitHub link, you will be redirected to the GitHub login page, as shown in the following figure:



*Figure 10.17: Login screen from GitHub*

Once you login to GitHub, you will be redirected to the **BookClub** application, as shown in the following figure:



| Book ID              | Title                                        | Author             | Publication | Copies |
|----------------------|----------------------------------------------|--------------------|-------------|--------|
| <a href="#">BBB1</a> | Mastering Java Persistence API               | Nisha Parameswaran | 2022        | 3      |
| <a href="#">BBB6</a> | Selenium with Java                           | Pallavi Sharma     | 2022        | 0      |
| <a href="#">BBB5</a> | Fundamentals of Android App Development      | Sujit Kumar        | 2021        | 0      |
| <a href="#">BBB4</a> | JavaScript for Gurus                         | Ockert Preez       | 2020        | 0      |
| <a href="#">BBB3</a> | Software Design Patterns for Java Developers | Lalit Mehra        | 2021        | 0      |
| <a href="#">BBB2</a> | JAVA Programming Simplified                  | Muneer Ahmad       | 2020        | 0      |

*Figure 10.18: BookClub homepage after login*

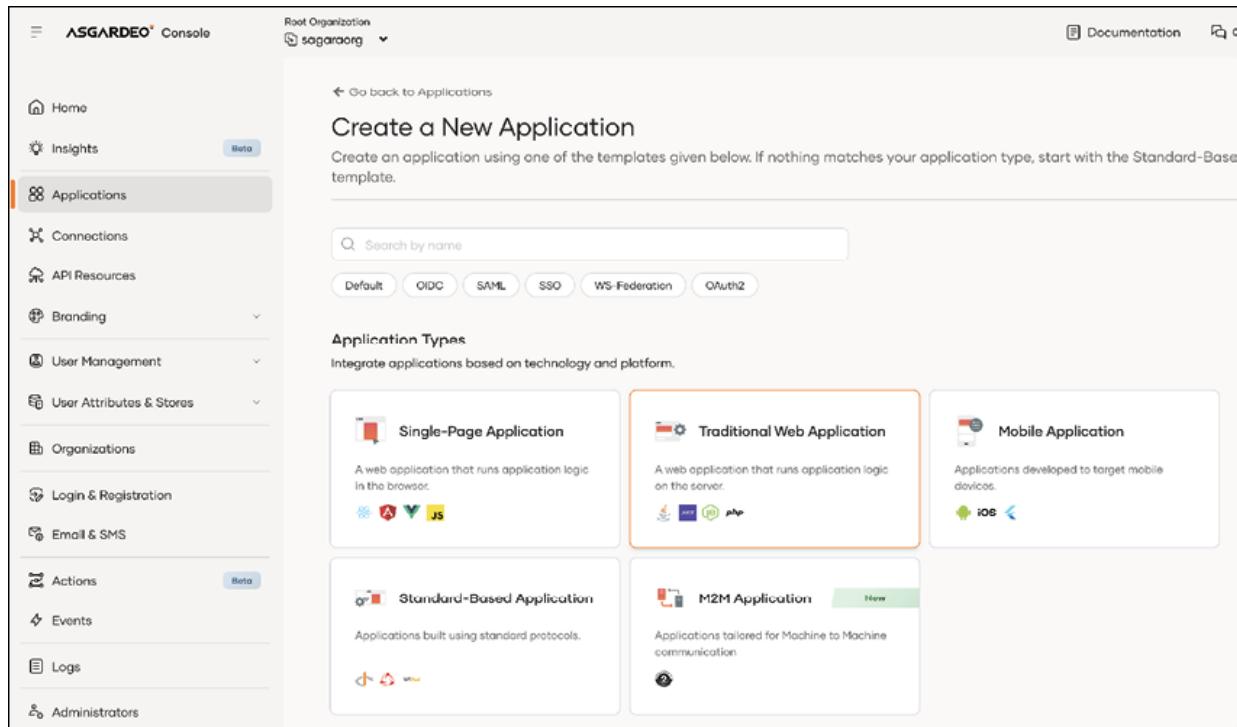
In this exercise, we used GitHub, a pre-configured provider in Spring Security. Next, let us configure an OIDC provider that is not pre-configured in Spring Security. This will help you understand the complete process of setting up an OIDC provider.

Since we have already used a social provider, let us now try a full-fledged identity provider. For this example, we will use Asgardeo Cloud IDaaS because it is straightforward to configure and offers a free tier suitable for our application.

First, sign up for a free Asgardeo account by visiting the following URL:

<https://asgardeo.io/>

Once you sign up with Asgardeo, we need to create an application. When you click the New Application button after selecting the Application from the left side menu, you should be able to see the **Create a New Application** wizard as shown in the following figure:



*Figure 10.19: Asgardeo application registration page*

As shown in the above screenshot, since we are developing a server-side application, you need to pick the **Traditional Web Application** option among the options. Then, enter the following values:

**Name:** Any available name

**Redirect URL:** <http://localhost:8080/login/oauth2/code/asgardeo>

**Traditional Web Application**  
A web application that runs application logic on the server. [Learn More](#)

**Name** \*

**Protocol**

 OpenID Connect
 SAML

**Authorized redirect URLs** \*  
 +

i Don't have an app? Try out a sample app using <http://localhost:8080/oidc-sample-app/oauth2client> as the authorized redirect URL. (You can download and run a sample at a later step.)

[Add Now](#)

**Help** <

**Name**  
A unique name to identify your application.  
E.g., My App

**Protocol**  
The access configuration protocol which will be used to log in to the application using SSO.

**Authorized redirect URLs**  
The URL to which the authorization code is sent to upon authentication and where the user is redirected to upon logout.  
E.g., <https://myapp.io/login>

Cancel
Create

**Figure 10.20:** Required metadata for Asgardeo application registration

After you create the application, you need to copy some configuration properties as we did in the previous example. First, move to the **Protocol** tab of the BookClub application as shown here, and copy the following values as we need them to configure our sample application:

- Client id
- Client secret

The screenshot shows the Asgardeo Console interface. The left sidebar has a 'Protocol' section with 'OpenID Connect' selected. The main content area is titled 'OpenID Connect' and displays configuration settings for an application. The 'Client ID' is set to '6gnngmtQcfRLsqCApg6lq18QXRga'. The 'Client secret' field is present but empty. Under 'Allowed grant types', 'Code' is checked, while 'Client Credential', 'Refresh Token', 'Implicit', and 'Token Exchange' are unchecked. A note below states: 'This will determine how the application communicates with the token service.' The 'Authorized redirect URLs' field contains 'https://myapp.io/login' and 'http://localhost:8080/login/oauth2/code/asgardeo'. The 'Protocol' tab is highlighted in orange, and the 'Documentation' link is visible in the top right.

*Figure 10.21: Metadata from Asgardeo application registration*

Next, move to the **Info** tab, as shown in the following figure, and copy the following values:

- Issuer

*Figure 10.22: Server endpoints from Asgardeo application registration*

Now, we are ready to configure our application, first update the **application.properties** file with the values that we copied in the previous step:

1. *#OAuth Application Properties*
2. `spring.security.oauth2.client.registration.asgardeo.client-name=Asgardeo`
3. `spring.security.oauth2.client.registration.asgardeo.client-id=[Client-ID]`
4. `spring.security.oauth2.client.registration.asgardeo.client-secret=[Client-Secret]`
5. `spring.security.oauth2.client.registration.asgardeo.`  
`redirect-uri=http://localhost:8080/login/oauth2/code/asgardeo`
6. `spring.security.oauth2.client.registration.asgardeo.`  
`authorization-grant-type=authorization_code`
7. `spring.security.oauth2.client.registration.asgardeo.scope=openid,profile`
- 8.
9. *#Identity Server Properties*
10. `spring.security.oauth2.client.provider.asgardeo.issuer-uri=[Issuer]`

Next, we need to update the **setUserAttributes** method of the

**CommonAttributes Interceptor** class with the following code:

```
1. private void setUserAttributes(ModelAndView modelAndView) {  
2.     Authentication authentication =  
    SecurityContextHolder.getContext().getAuthentication();  
3.     if (authentication != null && authentication.isAuthenticated()) {  
4.         if (authentication instanceof OAuth2AuthenticationToken) {  
5.             OAuth2AuthenticationToken token =  
    (OAuth2AuthenticationToken) authentication;  
6.             modelAndView.addObject("user",  
    token.getPrincipal().getAttribute("username"));  
7.         }  
8.     }  
9. }
```

Now, we can run our application.

Before we log in to our sample application, we need to create a test user in Asgardeo. You can do this by selecting **User Management** and then the **Users** option from the left menu. As shown in the following figure, create a test user with a password:

**Create User**  
Follow the steps to create a new user.

Basic Details    User Groups

Select user store  
DEFAULT

Username (Email) \*  
sam@bookclub.com

First Name \*  
Sam

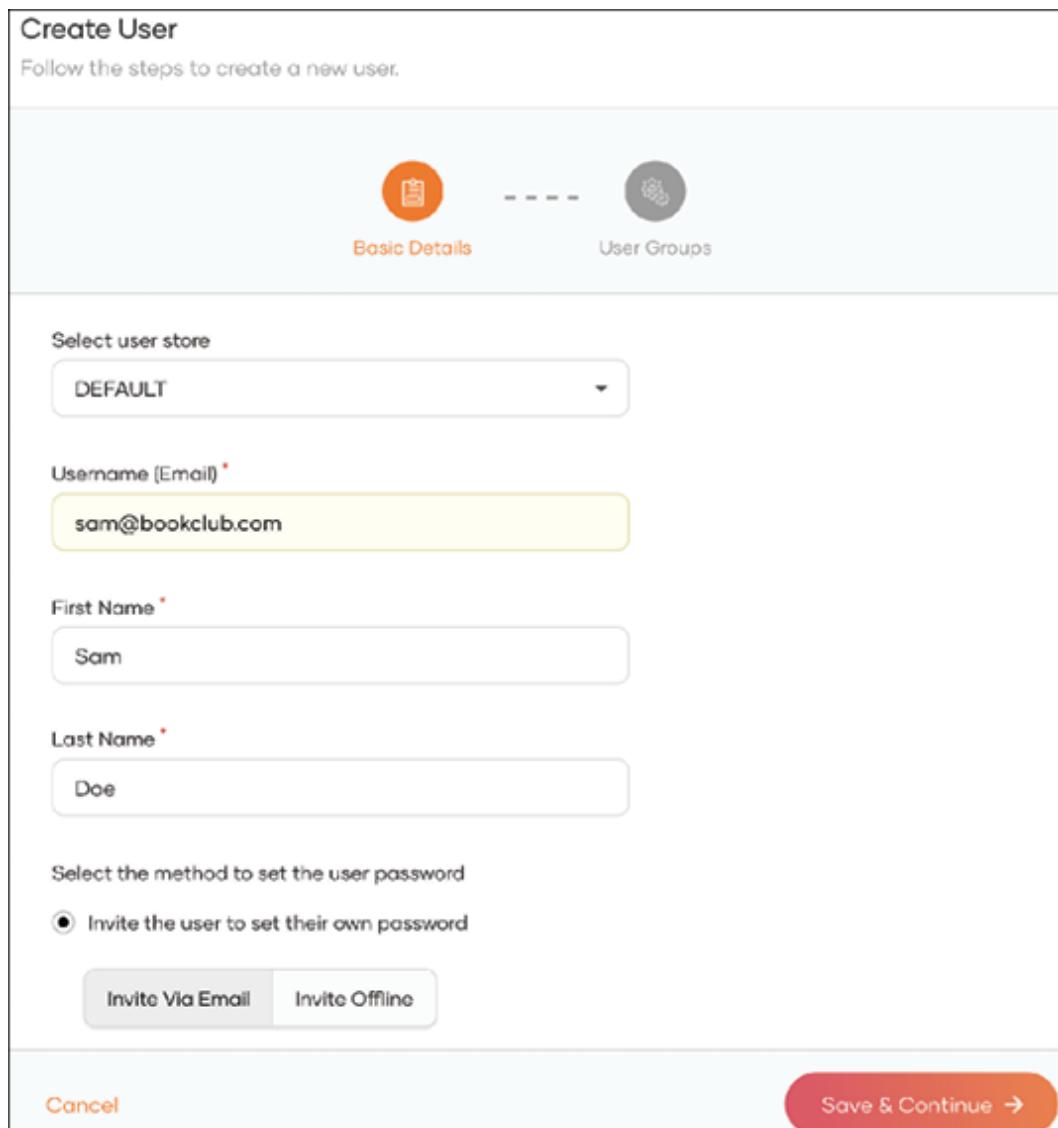
Last Name \*  
Doe

Select the method to set the user password

Invite the user to set their own password

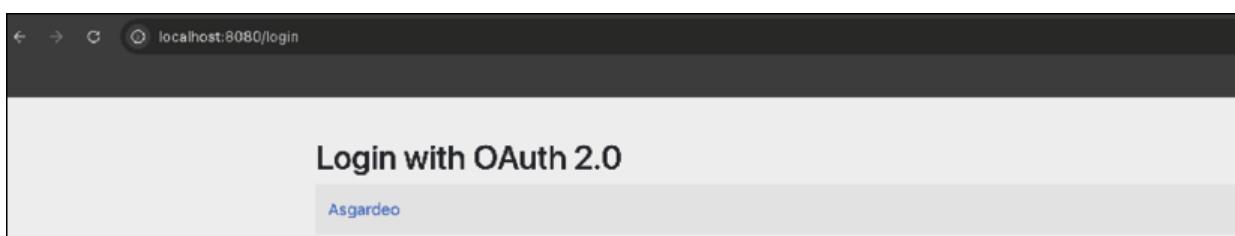
Invite Via Email    Invite Offline

Cancel    Save & Continue →



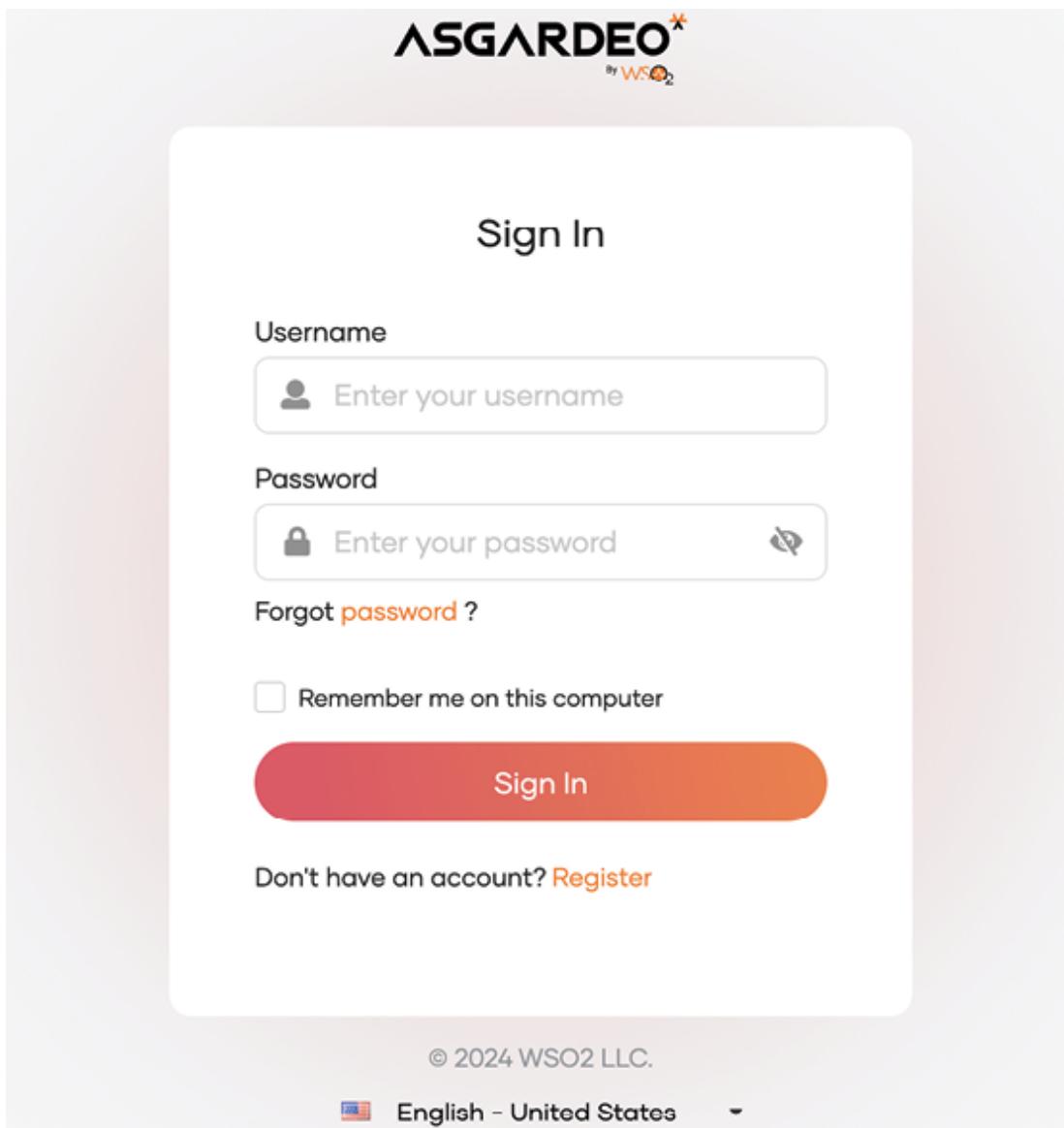
**Figure 10.23:** New user create wizard of Asgardeo

Now, we can visit our application using <HTTP://localhost:8080/login> URL. You should be able to see the default login screen generated by Spring Security with Asgardeo as an IdP:



**Figure 10.24:** Default login option page generated by Spring Security listing Asgardeo as an option

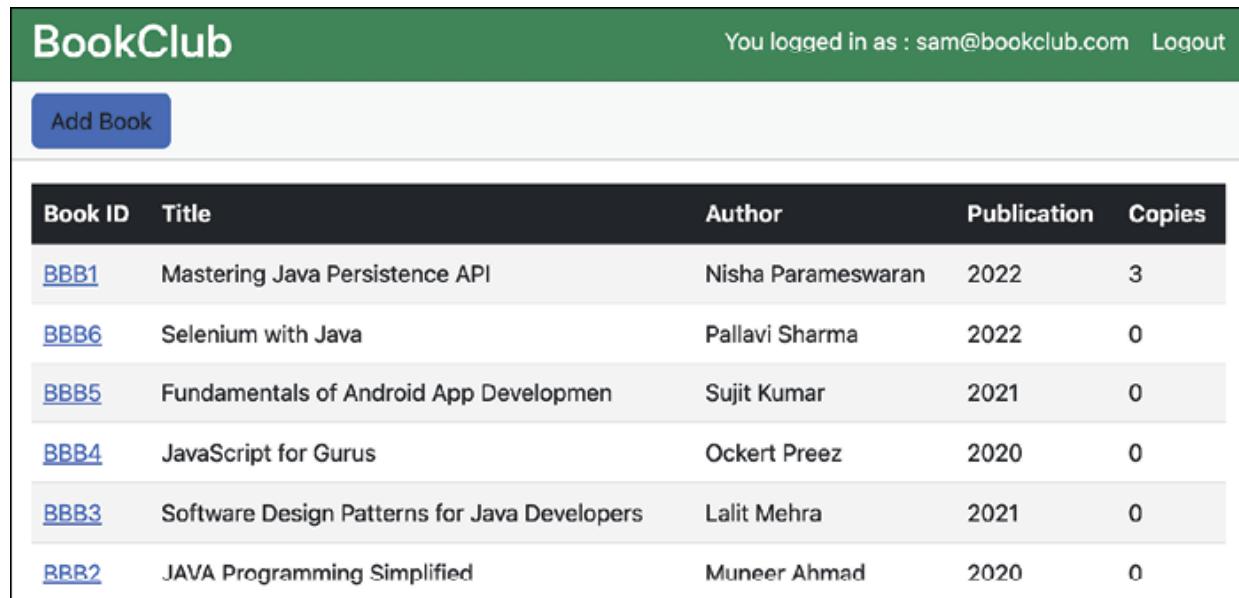
When you click the Asgardeo link, you will be redirected to the default login screen of Asgardeo, as shown in the following figure. You can fully rebrand the look, feel, and layout of this login screen to match the branding of your organization.



*Figure 10.25: Asgardeo default login screen*

Once authentication is completed, Asgardeo redirects you back to the application with a code grant. Spring Security then exchanges the code grant for an ID Token by making a token request to Asgardeo. This step occurs behind the scenes, so it is not visible to you. What you will see is the application's home page, displaying your logged-in username, as shown in

the following figure:



The screenshot shows the BookClub application's home page. At the top, there is a green header bar with the text "BookClub" on the left and "You logged in as : sam@bookclub.com Logout" on the right. Below the header is a light gray navigation bar with a blue "Add Book" button. The main content area is a table with a dark header row and white data rows. The columns are "Book ID", "Title", "Author", "Publication", and "Copies". The data rows are as follows:

| Book ID | Title                                        | Author             | Publication | Copies |
|---------|----------------------------------------------|--------------------|-------------|--------|
| BBB1    | Mastering Java Persistence API               | Nisha Parameswaran | 2022        | 3      |
| BBB6    | Selenium with Java                           | Pallavi Sharma     | 2022        | 0      |
| BBB5    | Fundamentals of Android App Developmen       | Sujit Kumar        | 2021        | 0      |
| BBB4    | JavaScript for Gurus                         | Ockert Preez       | 2020        | 0      |
| BBB3    | Software Design Patterns for Java Developers | Lalit Mehra        | 2021        | 0      |
| BBB2    | JAVA Programming Simplified                  | Muneer Ahmad       | 2020        | 0      |

*Figure 10.26: BookClub home page after login with Asgardeo*

Although we have achieved our objective, it would be better to display the first name of the logged-in user instead of the username. Since we are using an identity solution, we have the flexibility to define the user attributes and other information, such as roles, that the application expects. We can then configure the identity provider to include these attributes in the ID Token.

To achieve this, visit the BookClub application created in Asgardeo, navigate to the Attribute tab, and enable the **given\_name** attribute under the **Profile** section. This process is illustrated in the following figure:

| User Attribute                    | Requested                           | Mandatory                |
|-----------------------------------|-------------------------------------|--------------------------|
| Gender (gender)                   | <input type="checkbox"/>            | <input type="checkbox"/> |
| Time Zone (zoneinfo)              | <input type="checkbox"/>            | <input type="checkbox"/> |
| First Name (given_name)           | <input checked="" type="checkbox"/> | <input type="checkbox"/> |
| Photo URL (picture)               | <input type="checkbox"/>            | <input type="checkbox"/> |
| Last Name (family_name)           | <input type="checkbox"/>            | <input type="checkbox"/> |
| Display Name (preferred_username) | <input type="checkbox"/>            | <input type="checkbox"/> |
| Middle Name (middle_name)         | <input type="checkbox"/>            | <input type="checkbox"/> |
| Full Name (name)                  | <input type="checkbox"/>            | <input type="checkbox"/> |
| Locl (locale)                     | <input type="checkbox"/>            | <input type="checkbox"/> |
| Website URL (website)             | <input type="checkbox"/>            | <input type="checkbox"/> |
| Nick Name (nickname)              | <input type="checkbox"/>            | <input type="checkbox"/> |
| Username (username)               | <input checked="" type="checkbox"/> | <input type="checkbox"/> |

**Figure 10.27:** Select additional user attributes to be included in ID Token

Once we completed that, Asgardeo made sure to include the **given\_name** attribute in the ID token. Now, we need to change the **setUserAtributes** method of the **CommonAttributesInterceptor** class slightly to use the **given\_name** attribute instead of the **username** attribute. The code for the updated method is given as follows:

```

1. private void setUserAtributes(ModelAndView modelAndView) {
2.     Authentication authentication =
3.         SecurityContextHolder.getContext().getAuthentication();
4.     if (authentication != null &&
5.         authentication.isAuthenticated()) {
6.         if (authentication instanceof OAuth2AuthenticationToken) {
7.             OAuth2AuthenticationToken token =
8.                 (OAuth2AuthenticationToken) authentication;
9.             modelAndView.addObject("user",
10.                 token.getPrincipal().getAttribute("given_name"));
11.         }
12.     }
13. }
```

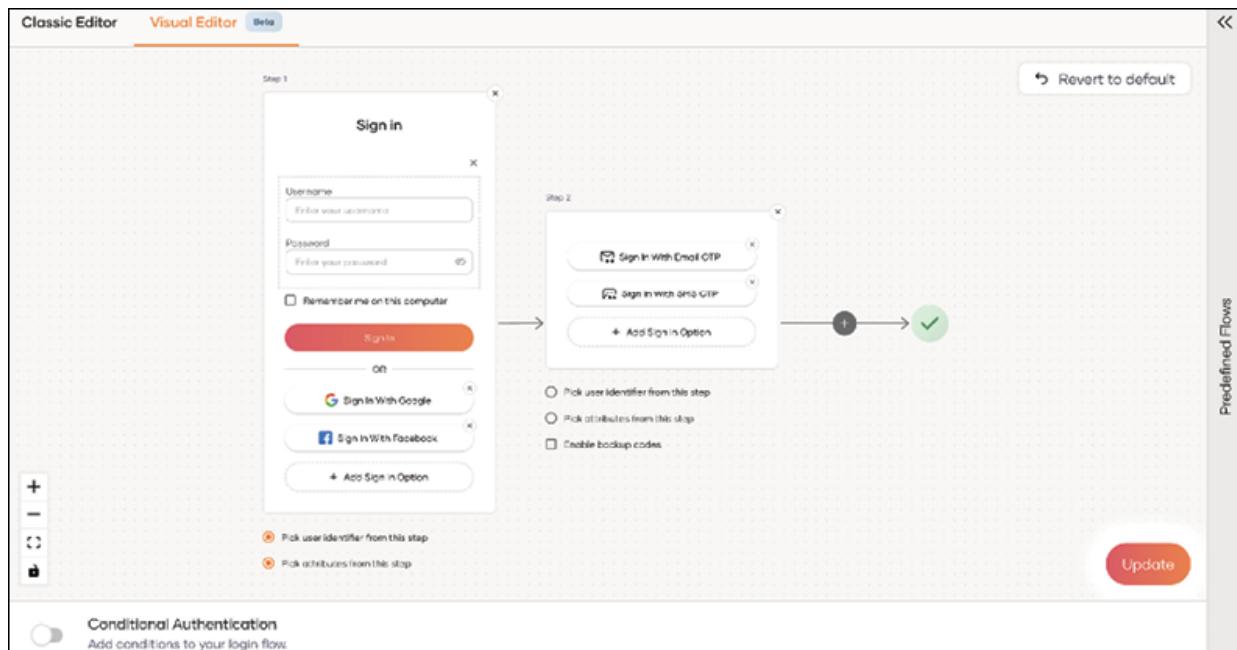
Now, if you revisit the application, you should be able to see the home page with the first name of the logged-in user, as shown in the following figure:

[Add Book](#)

| Book ID              | Title                                        | Author             | Publication | Copies |
|----------------------|----------------------------------------------|--------------------|-------------|--------|
| <a href="#">BBB1</a> | Mastering Java Persistence API               | Nisha Parameswaran | 2022        | 3      |
| <a href="#">BBB6</a> | Selenium with Java                           | Pallavi Sharma     | 2022        | 0      |
| <a href="#">BBB5</a> | Fundamentals of Android App Developmen       | Sujit Kumar        | 2021        | 0      |
| <a href="#">BBB4</a> | JavaScript for Gurus                         | Ockert Preez       | 2020        | 0      |
| <a href="#">BBB3</a> | Software Design Patterns for Java Developers | Lalit Mehra        | 2021        | 0      |
| <a href="#">BBB2</a> | JAVA Programming Simplified                  | Muneer Ahmad       | 2020        | 0      |

*Figure 10.28: BookClub home page showing the proper user name*

At the beginning of this chapter, we mentioned that using an identity provider saves a lot of time and investment by providing out-of-the-box features. To give you an idea, Asgardeo allows you to design and add MFA and social logins using a graphical login flow designer. An example design is shown in [Figure 10.29](#). This capability is not specific to Asgardeo; most identity providers available today offer similar features:

*Figure 10.29: Asgardeo visual login flow editor*

## Working with logs, metrics, and traces

In [Chapter 3](#), *Spring Essentials for Enterprise Applications*, we discussed logs, metrics, and traces as the three key pillars of observability for any application, including those built with Spring Boot. We covered how to generate logs, control log levels, and effectively use log statements in your code. Additionally, we explored how to generate metrics and traces using the Micrometer Observability features available in Spring Boot applications. In the bottom line, we focused on how to produce meaningful and useful logs, metrics, and traces from our applications. In this section, we will shift our focus to understanding how to collect, analyze, and visualize logs, metrics, and traces in production systems. Let us begin with logs.

Log statements are one of the simplest and most popular debugging techniques during development. For instance, while developing a RESTful Controller, we might add log statements to print headers and parameters from the body to verify that we are receiving the correct values from the client. Similarly, in a complex function, we might include log statements to check its behavior with different inputs. After adding these log statements, we simply check the console or log files for new entries to verify our observations. This straightforward and efficient process works well during development.

However, this simple approach does not fit production systems for several reasons:

- Production systems run 24/7, year-round, making it impractical for someone to monitor log entries in real-time for specific information.
- Production systems generate a significant amount of log data. If not managed properly, this can affect server performance and even interrupt services. Log rotation addresses this issue by periodically archiving and creating new log files to prevent them from growing too large. In Spring Boot applications, this can be configured using Logback's `RollingFileAppender` in the **application.properties** file.
- In production, logs are typically not reviewed regularly. Instead, they are analyzed to troubleshoot issues such as failed transactions or performance problems. This requires searching for specific keywords and filtering historical logs for a particular time period. In some

industries, compliance regulations mandate retaining logs for specific durations, which may range from 30 days to several years.

- Production logs can be analyzed and aggregated to generate reports and dashboards that provide valuable insights into system and business performance. For example, a dashboard showing HTTP request trends by hour or day can offer critical information about system load and business activity.

**Tip:** If you are deploying your Spring Boot application in a container management system like Kubernetes, it is important to utilize the log management capabilities provided by the underlying platform. For example, in Kubernetes, you should not write log entries to files directly. Instead, log entries should be written to the `System.out` and `System.err` streams. Kubernetes automatically captures these streams, writes them to log files, and allows you to manage them at the node level. This approach simplifies log management and aligns with Kubernetes's best practices.

By now, you have likely realized that log management in a production system is a crucial and complex task, far more critical than it might initially appear. There are three main types of log management solutions that can meet all logging requirements, including collecting, storing, analyzing, and aggregating logs into reports and dashboards:

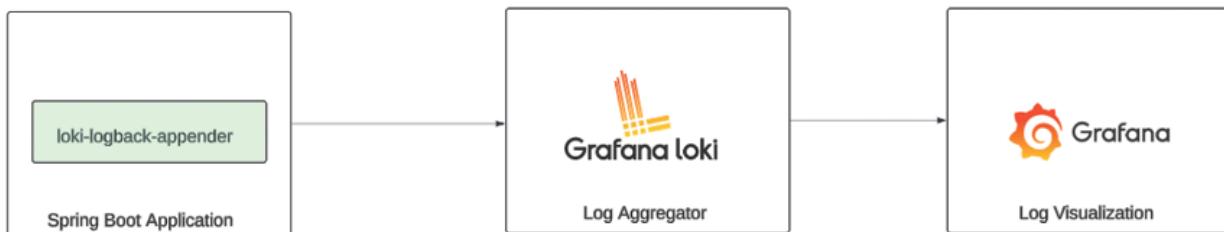
- **Cloud-based log monitoring solutions:** With cloud-based solutions, you can publish your log entries by adding an agent dependency to your application. This agent collects the required log entries and sends them to the cloud-based log management system. Once the logs are published, you can view them as raw data or define rules to analyze and aggregate them into reports and dashboards. Most cloud-based solutions offer out-of-the-box reports and dashboards for common use cases, making this the simplest and quickest approach. However, you need to consider the cost as your data grows. Popular solutions like Splunk and Datadog often provide free tiers for testing their capabilities.
- **Platform-based log monitoring solutions:** If your applications are deployed on a cloud platform such as AWS, Azure, or GCP, you can leverage the log monitoring features offered by the platform. Similar to cloud-based solutions, platform-based solutions provide pre-configured reports and dashboards for common needs. However, like cloud-based options, you must factor in long-term costs.
- **Self-managed log monitoring solutions:** Several open-source and

proprietary log monitoring tools allow you to deploy and manage the log monitoring process yourself, just like your Spring Boot application. Although it may take some time to learn these tools and set them up initially, self-managed solutions can be the most cost-effective option in the long run.

Let us look at one possible log management and visualization solution based on two components of the Grafana open-source stack, Grafana Loki and Grafana:

- Grafana Loki is a powerful and efficient log aggregation system designed to simplify the process of collecting, storing, and querying log data. Unlike traditional log aggregation systems that index the full content of logs, Loki indexes only metadata labels, resulting in reduced storage requirements and improved query performance. This new approach allows organizations to handle large volumes of log data while maintaining scalability and cost efficiency.
- Grafana serves as the visualization layer for Loki, providing a user-friendly interface for exploring and analyzing log data. Together, Loki and Grafana form a powerful combination that enables us to gain valuable insights from logs quickly and efficiently.

The following figure shows you how you can connect Loki and Grafana with your Spring Boot application. You basically need to add a Loki4j appender, such as **loki-logback-appender**, as per the underlying logging framework used in your Spring application, and then configure the Loki4j appender to push log data to the Loki:

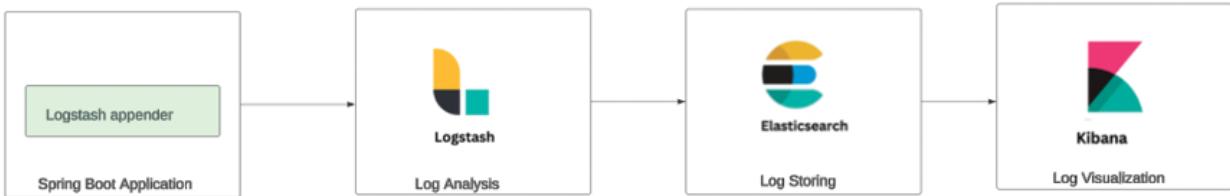


*Figure 10.30: Log processing using Grafana stack*

As shown in the above diagram, logs can be generated in JSON format and sent using HTTP POST to Loki via the Loki4j Logback appender. Loki stores and indexes the incoming log data and processes queries for log retrieval; these queries can be defined using the query language used in Loki

called LogQL. Finally, Grafana provides a visualization and analysis interface for logs where you can define dashboards and alerts based on log data.

As an alternative solution, as shown, you could use **Elasticsearch, Logstash, and Kibana (ELK)** stack instead of Grafana stack:



*Figure 10.31: Log processing using ELK stack*

When using the ELK stack for log analysis, you need to include and configure the Logstash appender to publish log data to Logstash. Logstash collects logs from multiple sources, processes and transforms them, and forwards the processed logs to Elasticsearch, acting as a part of the log processing pipeline. Elasticsearch is responsible for storing and indexing log data while providing fast, scalable, full-text search capabilities. Finally, Kibana offers a web interface for visualizing and exploring log data, allowing the creation of custom dashboards and visualizations. Although there are significant architectural differences, both Grafana and the ELK stack fundamentally offer the same set of log processing capabilities.

Now, let us shift our focus to metrics. Unlike logs, metrics often receive less attention during development. In fact, the sheer volume of metrics data generated can be overwhelming and difficult to interpret. However, in production systems, metrics are crucial. Without them, we would not know if our applications are running smoothly or if there are issues requiring intervention. Think of metrics as analogous to measuring body temperature, blood pressure, and pulse rate to assess a person's health. Metrics serve the same purpose in monitoring an application's health. They can be utilized for two main purposes:

- Metrics collected over time can be stored, processed, and aggregated to generate performance dashboards. These dashboards provide vital insights into how an application performs over time, which is essential for assessing its health and resource requirements. For instance, such dashboards can help identify peak and low load times for the

application.

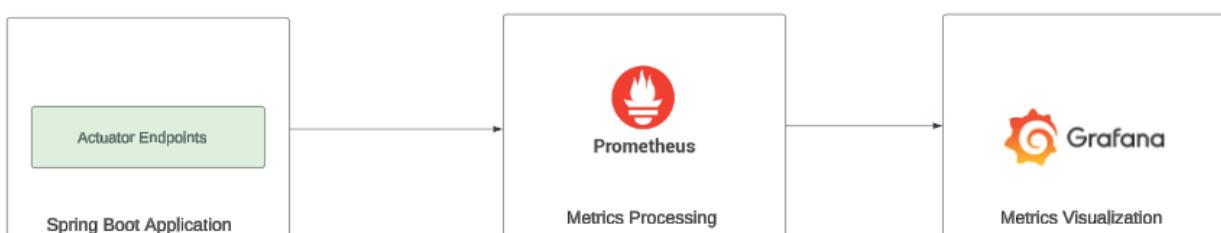
- Metrics can also be used to trigger alerts, allowing system maintainers to intervene when necessary. For example, if available free memory drops below a certain threshold, say 20%, the maintenance team should be notified. Similarly, if the number of requests processed by the application falls below a specific threshold, alerts can prompt action. Metrics solutions often come with built-in capabilities to define threshold values and specify actions, such as sending an email or SMS to the maintenance team, when these thresholds are breached.

Prometheus is an open-source monitoring and alerting solution ideal for implementing the metrics-related use cases discussed earlier. It can collect, store, and analyze metrics from various sources. Prometheus's architecture is based on a pull model, where the Prometheus server actively scrapes metrics from configured endpoints at specified intervals.

Key components of Prometheus include the Prometheus server itself, which handles data collection and storage; exporters that convert existing metrics from various systems into a format Prometheus can understand; client libraries that facilitate the instrumentation of application code for generating custom metrics; and the Alert Manager, which manages alerts based on defined thresholds or conditions.

Prometheus also offers a powerful query language, **PromQL**, for analyzing metrics and integrates seamlessly with visualization tools like Grafana. This combination provides developers and operators with deep insights into system performance and health.

The following figure illustrates how Prometheus can collect and analyze metrics from your Spring Boot applications, with Grafana used for visualization:



*Figure 10.32: Metrics processing using Prometheus*

From the Spring Boot application side, you need to add **spring-boot-starter-**

**actuator** as a dependency to enable actuator endpoints. Additionally, you need to add **micrometer-registry-prometheus** as a dependency to expose the metrics in the format expected by Prometheus.

Once these dependencies are added, Prometheus can scrape metrics from the Spring Boot application's **/actuator/prometheus** endpoint and store them as time-series data. It also provides a query interface that allows querying the data using PromQL.

Finally, Grafana connects to Prometheus and provides a web interface for visualizing the collected metrics.

As our last topic under this section, let us discuss traces in a production system. While tracing is not as common as log and metrics capabilities, it is important in certain applications, especially when the system consists of multiple processing applications, such as in a microservices architecture.

Grafana Tempo is an open-source, high-scale distributed tracing backend designed for simplicity and cost-effectiveness. It seamlessly integrates with Grafana, Prometheus, and Loki. Tempo supports the ingestion of common open-source tracing protocols, including Jaeger, Zipkin, and OpenTelemetry. Tempo's unique approach to trace storage relies solely on object storage, eliminating the need for complex databases and significantly reducing operational costs.

Key components of Tempo include its ingestion pipeline for receiving traces, a query frontend for handling trace retrieval requests, and integration with Grafana for visualization. Tempo also features **TraceQL**, a powerful query language for searching and analyzing traces. The following figure shows how you can connect your Spring Boot application with Tempo for tracing:



*Figure 10.33: Traces processing using Grafana stack*

First, you need to add **micrometer-tracing-bridge-brave** and **zipkin-reporter-brave**, along with **spring-boot-starter-actuator**, to activate the tracking generation and publishing components in your Spring Boot

application. At runtime, Spring Boot uses OpenTelemetry for instrumentation and generates traces. These traces are then published to Tempo using the OpenTelemetry Collector. Tempo stores and indexes trace data, providing a query interface for trace retrieval. Finally, Grafana connects with Tempo, offering a web interface for visualizing and analyzing traces.

We have discussed log processing, metrics, and traces as separate topics and provided potential high-level solutions. The following figure presents a consolidated solution that integrates the processing and visualization of logs, metrics, and traces, creating a comprehensive monitoring solution for your production applications.



**Figure 10.34:** Logs, metrics, and traces using Grafana stack

## Conclusion

You should now have a solid understanding of key concepts and techniques for deploying and securing Spring Boot applications. You have learned how to leverage Spring Profiles to run applications in different environments, create Docker and GraalVM native images, and customize the default web server to use alternatives like Jetty and Undertow. We also covered how to enable SSL for secure communication and integrate identity providers to

enhance your application's security. Finally, we explored how to collect, analyze, and visualize logs, metrics, and traces, equipping you with the tools necessary to monitor and maintain a healthy production system.

## **Join our book's Discord space**

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



*OceanofPDF.com*

# CHAPTER 11

## Emerging Trends in Spring Framework

### Introduction

In this final chapter, we will take a quick look at a few new Spring projects that will be important for future application development, including Spring AI, Spring Authorization Server, and Spring Vault.

Next, we will discuss the new features introduced in the latest releases of the Spring Framework and Spring Boot, along with an overview of what to expect in the upcoming major release of Spring Framework 7.

### Structure

The chapter covers the following topics:

- Introduction to Spring AI
- Introduction to Spring Authorization Server
- Introduction to Spring Vault
- Upcoming features in Spring

### Objectives

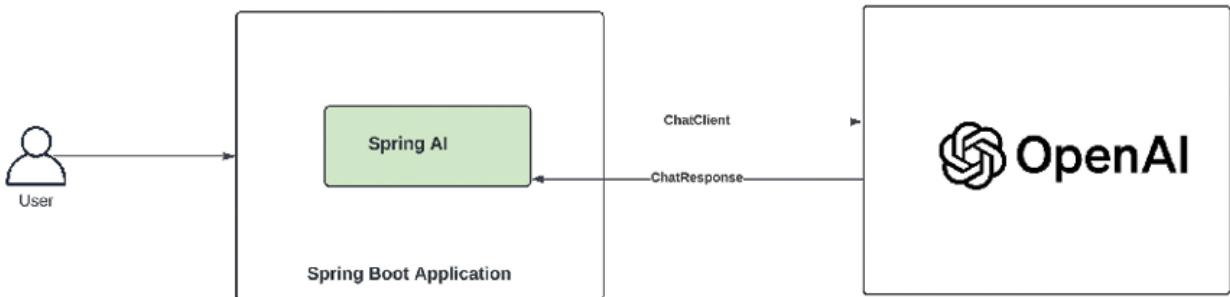
By the end of this chapter, you will have a clear understanding of the significance of Spring AI, Spring Authorization Server, and Spring Vault,

along with an overview of the key features of each project. Additionally, you will gain insight into the upcoming features in the latest releases of Spring Boot and the Spring Framework, as well as what to expect from the Spring Framework 7.0 release.

## Introduction to Spring AI

The Spring AI project is the latest addition to the Spring ecosystem, aiming to facilitate Java developers to build a generative AI application, which is currently dominated by Python. Following the path of its predecessors, instead of building its own AI models, Spring AI focuses on simplifying the integration of existing AI model providers, such as OpenAI, Anthropic, Microsoft, Amazon, and Google, into your applications. Spring AI provides portable abstraction APIs across AI providers and makes it easy to add features like AI-powered chat completion, text-to-image generation, text-to-speech conversion, and audio transcription to your applications.

As an example, the following figure illustrates how you could leverage OpenAI capabilities within your Spring Boot application using the ChatClient provided by the Spring AI project to integrate with OpenAI APIs:



*Figure 11.1: OpenAI integration with a Spring Boot application*

Here are some of the important features of the Spring AI project:

- **Portable API support across AI providers:** Support for chat, text-to-image, text classification, image generation, transcription, and embedding models in synchronous and asynchronous manner and extension to access the model-specific features.
- **Structured outputs:** Mapping of AI model output to POJOs hiding the complexities of AI models from Spring developers.

- **Vector database support:** Integration with major providers like Apache Cassandra, Azure Cosmos DB, Elasticsearch, MongoDB Atlas, Neo4j, PostgreSQL/PGVector.
- **Tools and function calling:** Allows models to request execution of client-side tools and functions for accessing real-time information.
- **Document injection ETL framework:** allows for inserting data from multiple sources into a transformation at runtime, reducing the need for repetitive ETL tasks for various input sources.
- **AI model evaluation:** Utilities to evaluate generated content and protect against hallucinated responses
- **ChatClient API:** A fluent API for communicating with AI chat models similar to similar to WebClient and RestClient APIs.
- **Advisors API:** Encapsulates recurring generative AI patterns and transforms data sent to and from **large language models (LLMs)** to enable and provide portability across various models and use cases.

In addition to that, similar to any other Spring project, Spring AI provides auto-configuration and starters and observability features.

## Introduction to Spring Authorization Server

Throughout this book, we have extensively utilized the capabilities offered by the Spring Security project to secure web applications and services. The Spring Authorization Server is a framework that implements OpenID Connect 1.0 and OAuth 2.1, along with related specifications. This allows you to implement **authorization server (AS)** capabilities according to the OAuth 2.1 specification and **OpenID Provider (OP)** capabilities according to the OpenID Connect specification. In simpler terms, this means the Authorization Server can be used to implement provider capabilities for your applications and services without relying on separate **identity providers (IdPs)**.

In the previous chapter, we had an in-depth discussion on the pros and cons of building your own identity provider versus using a third-party identity provider solution. If building your own identity provider is the right option for you, the Authorization Server makes it much easier to do so.

## Introduction to Vault

The Spring Vault project provides both low-level and high-level client-side abstraction APIs for accessing, storing, and revoking secrets used by your applications and services while integrating seamlessly with HashiCorp Vault. It also securely manages credentials for external services such as MySQL, PostgreSQL, Cassandra, and AWS.

Some of the key features of Spring Vault include:

- Spring Vault allows easy configuration of the Vault client using Java-based configuration.
- Spring Vault can initialize Spring Environment with remote property sources using the properties stored in Vault.
- Supports multiple authentication methods such as token, AppId, AppRole, Client Certificate, Cubbyhole, and AWS-EC2 authentication.
- VaultTemplate is a central class providing a rich feature set to interact with Vault, including reading, writing, and deleting data.
- @VaultPropertySource is a convenient and declarative mechanism for adding a PropertySource to Spring's Environment from Vault.

## Upcoming features in Spring

As of the time of writing this book, Spring Framework 6.2.0 and Spring Boot 3.4.0 are the latest released versions, both launched in November 2024. The Spring Framework 6.2 version introduces a slightly revised auto-wiring algorithm, improvements to **Spring Expression Language (SpEL)**, and an upgrade to HtmlUnit 3. Meanwhile, the Spring Boot 3.4.0 release includes several enhancements across various projects, including Spring AMQP, Spring Security, Spring Batch, and Spring Integration.

The Spring project team has also announced plans to release the next major version, Spring Framework 7.0.0, in late 2025. Some of the notable improvements expected in this release include:

- **Jakarta EE 11 support:** Spring Framework will upgrade its support to the latest Jakarta EE 11 specifications, aligning with Spring's strategy to stay current with Java enterprise standards. This will ensure compatibility with and enable the use of features from Tomcat 11,

Hibernate 7, and Hibernate Validator 9.

- **JDK 25 support:** Spring Framework 7.0 will support JDK 25, the next **Long-Term Support (LTS)** version of Java after JDK 21, while still retaining JDK 17 as its baseline version. This means Spring Framework 7.0 will be compatible with JDK versions from 17 through 25.
- **JSpecify adoption:** Spring Framework has traditionally used its own nullability annotations. With Spring Framework 7, it will align with the newly released JSpecify standard, a collaborative effort to define a standard set of annotations for nullability in Java.

Other than the above-mentioned features, The Spring Framework 7.0 will deliver advancements in **ahead-of-time (AOT)** compilation, aligning with GraalVM and Project Leyden. AOT compilation is an advanced optimization technique that transforms application code into native machine code before runtime execution. In Spring, particularly with Spring Boot 3 and later versions, AOT processing analyzes the application context at build time, generating optimized bytecode and resources. This process prepares Spring applications for efficient native image compilation, significantly reducing start-up times and memory usage and improving overall performance, which is especially beneficial in environments where quick scaling and resource optimization are critical.

## Conclusion

In this chapter, we have explored the significance of Spring AI, Spring Authorization Server, and Spring Vault, providing you with an overview of the key features of each project. Additionally, we have discussed the latest updates in Spring Boot and Spring Framework, offering insights into their new features. Finally, we have highlighted the upcoming advancements in Spring Framework 7.0.

## Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

[https://discord\(bpbonline.com](https://discord(bpbonline.com)



*OceanofPDF.com*

# Index

## Symbols

`<bean>` [13](#)  
`@Argument` [185](#)  
`@DataJpaTest` [148](#)  
`@GraphQlTest` [193](#)  
`@JmsListener` [237](#)  
`@Profile` [260](#)  
`@QueryMapping` [185](#)  
`@RestController` [158](#)

## A

Advanced Message Queuing Protocol (AMQP) [245](#)  
Apache Maven [20](#), [21](#)  
ApplicationContext [9](#)

## B

BookClub [88](#)  
BookClub Application, optimizing [89-96](#)  
BookController [96](#)

## C

`collectList()` [227](#)  
CrudRepository [144](#), [145](#)

## D

Data Access [130](#)  
Data Access, features  
    Consistent, programming [130](#)  
    Exception, handling [130](#)  
    JDBC, supporting [130](#)  
    ORM, integrating [130](#)  
    Transaction, managing [130](#)  
Data Access, points  
    Auditing [131](#)  
    Data Source, integrating [131](#)  
    Query, methods [131](#)  
    Repository, abstraction [131](#)

Data Access, tips  
Auto-Configuration 131  
Embedded Database, supporting 131  
Property-Based, configuring 131  
Starter, dependencies 131  
Data Validation 108-112  
Dependency Injection (DI) 4-6  
DI, approaches  
Constructor Injection 7  
Field Injection 8  
Setter Injection 7  
DispatcherServlet 83  
Docker 263  
Docker With Spring Boot, optimizing 262-266

## **E**

Enterprise Applications 44  
Error Handling 112  
Error Handling, architecture 112-116  
Error Handling, concepts 161-163

## **F**

flatMap 223  
Functional Endpoint Model 222  
Functional Endpoint Model, configuring 223-225  
Functional Endpoint Model, elements 223

## **G**

GraalVM 266  
GraalVM With Spring Boot, preventing 266, 267  
Gradle 21  
Grafana Loki 292, 293  
Grafana Loki, purpose 293  
Grafana Tempo 294  
Grafana Tempo, components 294, 295  
GraphQL 180  
GraphQL Clients 195, 196  
GraphQL Clients, configuring 196-200  
GraphQL Clients, interfaces 196  
GraphQL, drawbacks  
Complexity 181  
Curve, learning 181  
Performance, concerns 181  
GraphQL, features  
Data, aggregation 181  
Flexibility/Efficiency 181

Self-Documenting [181](#)  
Strongly Typed, schema [181](#)  
Versioning [181](#)  
GraphQL, operations  
  Mutations [181](#)  
  Queries [181](#)  
GraphQL Services [181, 182](#)  
GraphQL Services, configuring [182-189](#)  
GraphQL Services, options [182](#)  
GraphQL Services, securing [200, 201](#)  
GraphQL Services, testing [192-195](#)  
GreetingService [5](#)

## **H**

HelloController [86](#)  
Hypertext Transfer Protocol (HTTP) [156](#)

## **I**

IDE, options  
  Executable JAR [41](#)  
  Traditional WARs [42](#)  
IDE, setting up [21, 22](#)  
ID Token [276, 277](#)  
Inversion of Control (IoC) [4](#)

## **J**

Java Database Connectivity (JDBC) [131-133](#)  
Java Message Service (JMS) [233](#)  
Java Persistence API (JPA) [140](#)  
JDBC, configuring [133-140](#)  
JdbcTemplate [135, 139](#)  
Jetty [268](#)  
JmsTemplate [241](#)  
JPA, architecture [140-145](#)  
JPA, repositories [145](#)

## **L**

lambda [210](#)

## **M**

Messaging [230-232](#)  
Messaging, terms  
  ActiveMQ [233-240](#)  
  RabbitMQ [245-248](#)  
Mocking [50-54](#)

MongoDB [149](#)  
MongoDB, configuring [149-154](#)

## O

Observability [65](#)  
Observability, configuring [127](#)  
Observability, pillars  
    Logging [65, 66](#)  
    Metrics [65](#)  
    Traces [66](#)  
OIDC [275](#)  
OIDC, key terms [275](#)  
OIDC, steps [276](#)

## P

Prometheus [293](#)  
Prometheus, components [294](#)  
Prometheus, configuring [294](#)  
Publisher [208](#)

## R

RabbitTemplate [249](#)  
Reactive Clients, building [225-228](#)  
Reactive Programming [204, 205](#)  
Reactive Programming, characteristics  
    Asynchronous Data, streaming [206](#)  
    Data-Oriented/Declarative [206](#)  
    Event-Driven [206](#)  
    Non-Blocking [206](#)  
    Responsiveness [206](#)  
Reactive Services [206, 207](#)  
Reactive Services, method  
    onComplete [208](#)  
    onError [208](#)  
    onNext [208](#)  
    onSubscribe [208](#)  
Reactive Services, testing [219-222](#)  
Representational State Transfer (REST) [156](#)  
REST APIs [180](#)  
REST APIs, points  
    Inflexibility [180](#)  
    Overfetching [180](#)  
    Underfetching [180](#)  
REST Documentation [166-168](#)  
RESTful Clients [169](#)  
RESTful Clients, configuring [169-174](#)

RESTful Services [158](#)  
RESTful Services, architecture [158-160](#)  
RESTful Services, securing [174-176](#)  
RESTful Services, testing [164-166](#)  
RestTemplate [173](#)  
REST, use cases  
    HTTP Headers [157](#)  
    HTTP Methods [157](#)  
    HTTP Status Code [157](#)  
    Resource, identifying [156](#)

## S

SDKMAN [20](#)  
Securing Spring Boot [57, 58](#)  
Securing Spring Boot, concepts  
    Authentication [57](#)  
    Authorization [57](#)  
    Exploits [57](#)  
Spring AI [298](#)  
Spring AI, features  
    Advisors API [299](#)  
    AI Model, evaluating [299](#)  
    ChatClient API [299](#)  
    Document Injection [299](#)  
    Function, calling [299](#)  
    Portable API, supporting [298](#)  
    Structured Outputs [298](#)  
    Vector Database, supporting [298](#)  
Spring Authorization Server [299](#)  
Spring Bean [12](#)  
Spring Bean, scopes  
    Application [13](#)  
    Prototype [12](#)  
    Request [12](#)  
    Session [12](#)  
    Singleton [12](#)  
    WebSocket [13](#)  
Spring Boot [14](#)  
Spring Boot 3.0 [18](#)  
Spring Boot Actuators [66-71](#)  
Spring Boot Actuators, endpoints [72](#)  
Spring Boot Actuators, features  
    Application Info [17](#)  
    Audit Events [17](#)  
    Beans, inspection [17](#)  
    Dynamic, configuration [18](#)  
    Health Checks [17](#)  
    HTTP, tracing [17](#)

- Loggers 18
- Prometheus 18
- Sessions 18
- Shutdown 18
- System Metrics 17
- Threaddump 18
- Spring Boot Application 23, 24
- Spring Boot Application, Annotation
  - @ComponentScan 28
  - @EnableAutoConfiguration 27
  - @SpringBootConfiguration 27
- Spring Boot Application, configuring 25-33
- Spring Boot Application, points 24
- Spring Boot Application, terms
  - Bean, configuring 255
  - Properties, configuring 254
- Spring Boot Auto-Configuration 16
- Spring Boot Auto-Configuration, cases 16
- Spring Boot, features
  - Actuator, monitoring 14
  - Classpath-Based Auto-Configuration 14
  - Command-Line, runner 14
  - Dependency, managing 14
  - Developer Tools 15
  - Initial Configuration 14
  - Multiple Environment, supporting 14
  - Web Servers, embedding 14
- Spring Boot, integrating 272, 273
- Spring Boot Logging 77
- Spring Boot Logging, capabilities 78
- Spring Boot Metrics 74-76
- Spring Boot, protocols
  - OIDC 275
  - SAML 274
- Spring Boot, reasons 290, 291
- Spring Boot, setting up 20
- Spring Boot, solutions 291, 292
- Spring Boot Starters 16
- Spring Boot Test 47, 48
- Spring Boot Test, configuring 49, 50
- Spring Boot Test, features 49
- Spring Boot Test, points
  - Mocking 50
  - Web Applications 54-56
- Spring Boot Tracing 79, 80
- Spring Framework 2, 15
- Spring Framework 6.0 18
- Spring Framework, challenges

- Dependencies, discovering [15](#)
- Incompatible, dependencies [15](#)
- Spring Framework, points
  - Enterprise Applications [3](#)
  - Java Applications [2](#)
  - Lightweight Framework [2](#)
- Spring Framework, use cases
  - Aspect-Oriented, programming [3](#)
  - Control Container, inversion [3](#)
  - Messaging [4](#)
  - Non-Relational Data, accessing [3](#)
  - Reactive Framework [3](#)
  - Relational Data, access [3](#)
  - Security [4](#)
  - Web Framework [3](#)
- Spring Initializer [22, 23](#)
- Spring IoC Container [8](#)
- Spring IoC Container, approaches
  - Annotation-Based Configuration [11](#)
  - Java-Based Configuration [11, 12](#)
  - XML Configuration [10](#)
- Spring IoC Container, interface
  - ApplicationContext [9](#)
  - BeanFactory [8](#)
- Spring MVC [82](#)
- Spring MVC Application, preventing [84-88](#)
- Spring MVC, architecture [82](#)
- Spring MVC, pattern [82](#)
- Spring MVC, steps [83](#)
- Spring MVC, technology
  - FreeMarker [84](#)
  - Groovy Markup [84](#)
  - Java Scripting Engine (JSR) [84](#)
  - JSP/JSTL [84](#)
  - Thymeleaf [84](#)
- Spring Profiles [255](#)
- Spring Profiles, configuring [255-261](#)
- Spring Security [58](#)
- Spring Security, attacks [59](#)
- Spring Security, configuring [59-65](#)
- Spring Security, functionalities [58](#)
- Spring Security, mechanisms [58](#)
- Spring Security Starter, steps [117-126](#)
- Spring Security Starter, terms [120](#)
- Spring, versions
  - Jakarta EE 11 [300](#)
  - JDK 25 [300](#)
  - JSpecify [300](#)

Spring WebFlux 206  
Subscriber 208

## T

TDD, principles  
Fast 46  
Independent 46  
Repeatable 46  
Self-Validating 46  
Thorough 46  
TDD, tests  
Functional 46  
Integration 46  
Performance 46  
Unit 45  
Testability 44  
Testability, principles  
Automated, testing 45  
Interfaces 44  
Modularity 44  
Observability 45  
Test-Driven Development (TDD) 45  
Test-driven Development (TDD) 45  
testGreet 49

## U

Undertow 269  
Undertow, configuring 269, 270

## V

Vault 299  
Vault, features 300