



Spring Boot 2 Recipes

A Problem-Solution Approach

—

Marten Deinum

Apress®

Spring Boot 2 Recipes

A Problem-Solution Approach

Marten Deinum

Apress®

Spring Boot 2 Recipes: A Problem-Solution Approach

Marten Deinum

Meppel, Drenthe, The Netherlands

ISBN-13 (pbk): 978-1-4842-3962-9

<https://doi.org/10.1007/978-1-4842-3963-6>

ISBN-13 (electronic): 978-1-4842-3963-6

Library of Congress Control Number: 2018964913

Copyright © 2018 by Marten Deinum

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spaehr

Acquisitions Editor: Steve Anglin

Development Editor: Matthew Moodie

Coordinating Editor: Mark Powers

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a Delaware corporation.

For information on translations, please e-mail editorial@apress.com; for reprint, paperback, or audio rights, please email bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484239629. For more detailed information, please visit www.apress.com/source-code.

Printed on acid-free paper

*For my wife and daughters.
I love you.*

Table of Contents

About the Author	xiii
About the Technical Reviewer	xv
Acknowledgments	xvii
Introduction	xix
Chapter 1: Spring Boot—Introduction.....	1
1-1 Create a Spring Boot Application Using Maven	2
Problem	2
Solution	2
How It Works	2
1-2 Create a Spring Boot Application Using Gradle.....	6
Problem	6
Solution	6
How It Works	6
1-3 Create a Spring Boot Application Using Spring Initializr.....	10
Problem	10
Solution	10
How It Works	10
1-4 Summary	14
Chapter 2: Spring Boot—Basics.....	15
2-1 Configure a Bean	15
Problem	15
Solution	15
How It Works	15

TABLE OF CONTENTS

2-2 Externalize Properties	21
Problem	21
Solution	22
How It Works	23
2-3 Testing	26
Problem	26
Solution	27
How It Works	27
2-4 Configure Logging	32
Problem	32
Solution	32
How It Works	32
2-5 Reusing Existing Configuration.....	34
Problem	34
Solution	34
How It Works	34
Chapter 3: Spring MVC	37
3-1 Getting Started with Spring MVC.....	37
Problem	37
Solution	37
How It Works	37
3-2 Exposing REST Resources with Spring MVC	42
Problem	42
Solution	42
How It Works	42
3-3 Using Thymeleaf with Spring Boot	52
Problem	52
Solution	52
How It Works	52

TABLE OF CONTENTS

3-4 Handling Exceptions	59
Problem	59
Solution	59
How It Works	59
3-5 Internationalizing Your Application	65
Problem	65
Solution	65
How It Works	65
3-6 Resolving User Locales.....	68
Problem	68
Solution	68
How It Works	69
3-7 Selecting and Configuring the Embedded Server.....	73
Problem	73
Solution	73
How It Works	73
3-8 Configuring SSL for the Servlet Container.....	79
Problem	79
Solution	79
How It Works	79
Chapter 4: Spring MVC - Async	85
4-1 Asynchronous Request Handling with Controllers and TaskExecutor.....	86
Problem	86
Solution	86
How It Works	86
4-2 Response Writers.....	91
Problem	91
Solution	91
How It Works	91

TABLE OF CONTENTS

4-3 WebSockets	100
Problem	100
Solution	100
How It Works	100
4-4 WebSockets with STOMP	110
Problem	110
Solution	111
How It Works	111
Chapter 5: Spring WebFlux	121
5-1 Developing a Reactive Application with Spring WebFlux.....	121
Problem	121
Solution	121
How It Works	123
5-2 Publishing and Consuming with Reactive Rest Services	128
Problem	128
Solution	128
How it Works	128
5-3 Use Thymeleaf as a Template Engine	137
Problem	137
Solution	137
How It Works	137
5-4 WebFlux and WebSockets.....	144
Problem	144
Solution	144
How It Works	144
Chapter 6: Spring Security	155
6-1 Enable Security in Your Spring Boot Application	155
Problem	155
Solution	155
How it Works	155

TABLE OF CONTENTS

6-2 Logging into Web Applications.....	163
Problem	163
Solution	163
How It Works	164
6-3 Authenticating Users	173
Problem	173
Solution	173
How It Works	174
6-4 Making Access Control Decisions.....	180
Problem	180
Solution	180
How It Works	180
6-5 Adding Security to a WebFlux Application	185
Problem	185
Solution	185
How It Works	185
6-6 Summary	191
Chapter 7: Data Access	193
7-1 Configuring a DataSource.....	193
Problem	193
Solution	193
How It Works	194
7-2 Use JdbcTemplate	204
Problem	204
Solution	204
How It Works	204
7-3 Use JPA.....	212
Problem	212
Solution	212
How It Works	212

TABLE OF CONTENTS

7-4 Use Plain Hibernate	221
Problem	221
Solution	221
How It Works	221
7-5 Spring Data MongoDB	224
Problem	224
Solution	224
How It Works	224
Chapter 8: Java Enterprise Services	239
8-1 Spring Asynchronous Processing	239
Problem	239
Solution	239
How It Works	240
8-2 Spring Task Scheduling	243
Problem	243
How It Works	244
8-3 Sending E-mail	246
Problem	246
Solution	246
How It Works	246
8-4 Register a JMX MBean	251
Problem	251
Solution	252
How It Works	252
Chapter 9: Messaging.....	257
9-1 Configure JMS	257
Problem	257
Solution	257
How It Works	257

TABLE OF CONTENTS

9-2 Send Messages Using JMS	264
Problem	264
Solution	264
How It Works	264
9-3 Receive Messages Using JMS.....	271
Problem	271
Solution	271
How It Works	271
9-4 Configure RabbitMQ	276
Problem	276
Solution	276
How It Works	277
9-5 Send Messages Using RabbitMQ.....	278
Problem	278
Solution	278
How It Works	278
9-6 Receive Messages Using RabbitMQ	286
Problem	286
Solution	286
How It Works	286
Chapter 10: Spring Boot Actuator.....	291
10-1 Enable and Configure Spring Boot Actuator	291
Problem	291
Solution	291
How It Works	292
10-2 Create Custom Health Checks and Metrics	300
Problem	300
Solution	300
How It Works	300

TABLE OF CONTENTS

10-3 Export Metrics	303
Problem	303
Solution	303
How It Works	303
Chapter 11: Packaging	307
11-1 Create an Executable Archive.....	307
Problem	307
Solution	307
How It Works	307
Making the Archive Executable	308
11-2 Create a WAR for Deployment.....	311
Problem	311
Solution	311
How It Works	311
11-3 Reduce Archive Size Through the Thin Launcher	314
Problem	314
Solution	315
How It Works	315
11-4 Using Docker	316
Problem	316
Solution	316
How It Works	317
Index.....	321

About the Author



Marten Deinum is a submitter on the open source Spring Framework project. He is also a Java/software consultant working for Conspect. He has developed and architected software, primarily in Java, for small and large companies. He is an enthusiastic open source user and longtime fan, user, and advocate of the Spring Framework. He has held a number of positions including software engineer, development lead, coach, and Java and Spring trainer.

About the Technical Reviewer



Manuel Jordan Elera is an autodidactic developer and researcher who enjoys learning new technologies for his own experiments and creating new integrations. Manuel won the Springy Award—Community Champion and Spring Champion 2013. In his little free time, he reads the Bible and composes music on his guitar. Manuel is known as dr_pompeii. He has tech reviewed numerous books for Apress, including *Pro Spring, 4th Edition* (2014), *Practical Spring LDAP* (2013), *Pro JPA 2, Second Edition* (2013), and *Pro Spring Security* (2013). Read his 13 detailed tutorials about

many Spring technologies, contact him through his blog at www.manueljordanelera.blogspot.com, and follow him on his Twitter account, @dr_pompeii.

Acknowledgments

Although this is the fourth book I've ever (co)authored, I'm still amazed by the amount of work that goes into creating the book. Not just by me writing the content, but also by the very nice people at Apress. I realized with my first book that writing a book is hard; however, writing a book on cutting edge technology (Spring Boot 2.1) is even harder. Writing content against prerelease/beta software will result in rewriting samples and content. This, at times, drove me and I guess my Editorial Manager (Mark Powers) nuts as well. So I'm sorry I had to rewrite and reproduce so much content and that all in all it took more time than I initially thought it would. However, the result is a fresh book using Java 11 and covering Spring Boot 2.1.

I owe a big thanks to all the people at Apress for trying to keep me on schedule while still keeping the quality and content up. Thank you for giving me the chance to finish my fourth book and for the support. So a very big thanks to all of you.

A book isn't written in isolation and without the comments and suggestions from the review, Manuel Jordan, some parts of the book would have been quite different. So Manuel, thanks for the comments, suggestions, and your time for reviewing this book (and reaching out to me).

Thanks to my family and friends for the times they had to miss me and again to my dive buddies for all the dives and trips I missed.

Last but definitely not least, I thank my wife, Djoke Deinum, and daughters, Geeske and Sietske, for their endless support, love, and dedication, despite the long evenings and sacrificed weekends and holidays to finish the book. Without your support, I probably would have abandoned the endeavor long ago.

Introduction

Welcome to the first edition of *Spring Boot 2 Recipes*. It will focus on developing software using Spring Boot 2.1 and the supported projects like Spring Security, Spring AMQP etc.

Who This Book is For

This book is for the developer who wants to simplify application development and have a faster startup time for writing applications. Introducing Spring Boot to you will simplify your application configuration. Using the full power of Spring Boot simplifies your deployment and management as well.

This book assumes that you are familiar with Java, Spring, and an IDE of some sort. This book doesn't explain all the internals and in-depth workings of Spring or the related projects. For in-depth coverage, pick up a copy of Spring 5 Recipes or Pro Spring MVC.

How This Book Is Structured

Chapter 1, “Spring Boot – Introduction,” gives a quick overview of Spring Boot and how to get started.

Chapter 2, “Spring Boot – Basics,” covers the basic scenarios of how to define and configure a bean and do dependency injection with Spring Boot.

Chapter 3, “Spring MVC,” Covers web-based application development using Spring MVC.

Chapter 4, “Spring MVC - Async,” covers asynchronous web-based application development using Spring MVC.

Chapter 5, “Spring WebFlux,” covers reactive web-application development using Spring WebFlux.

Chapter 6, “Spring Security,” provides an overview on how to secure your Spring Boot application using Spring Security.

Chapter 7 “Data Access,” explains how to access data storage like a database or MongoDB.

INTRODUCTION

Chapter 8 “Java Enterprise Services,” introduces how to use enterprise services like JMX, Mail, and scheduling with Spring Boot.

Chapter 9 “Messaging,” provides an introduction on how to do messaging with JMS and RabbitMQ using Spring Boot.

Chapter 10, “Spring Boot Actuator,” explains how to use the production-ready features like health and metrics endpoints from Spring Boot Actuator.

Chapter 11, “Packaging” covers how to package and deploy your Spring Boot application by making it executable or wrapping it in a Docker container.

Conventions

Sometimes, when we want you to pay particular attention to a part within a code example, we will make the font bold. Please note that the bold doesn't necessarily reflect a code change from the previous version.

In case a code line is too long to fit the page's width, we will break it with a code continuation character. Please note that when you try to type the code, you have to concatenate the line by yourself without any spaces.

Prerequisites

Because the Java programming language is platform independent, you are free to choose any supported operating system. However, some of the examples in this book use platform-specific paths. Translate them as necessary to your operating system's format before typing the examples.

To make the most out of this book, install JDK version 11¹ or higher. You should have a Java IDE installed to make development easier. For this book, most of the sample code is Maven² based, and most IDE's have build in support for Maven to manage the classpath. The samples all make use of the Maven Wrapper³ so you don't necessarily need to install Maven to be able to build the samples from the command line.

The samples sometimes need additional libraries installed like PostgreSQL, ActiveMQ, etc.; for this, the book uses Docker.⁴ Of course you can install the libraries

¹<https://adoptopenjdk.net/>

²<https://maven.apache.org/>

³<https://github.com/takari/maven-wrapper>

⁴<https://www.docker.com>

on your machine instead of using Docker, but for ease of use (and not polluting your system) using Docker is preferred.

Downloading the Code

The source code for this book can be accessed by clicking the Download Source Code link located at www.apress.com/9781484239629. The source code is organized by chapters, each of which includes one or more independent examples.

Contacting the Author

We always welcome your questions and feedback regarding the contents of this book. You can contact Marten Deinum by e-mail at marten@deinum.biz or Twitter at @mdeinum.

CHAPTER 1

Spring Boot—Introduction

In this chapter you will get a brief introduction to Spring Boot. At the heart of Spring Boot lies the Spring Framework; Spring Boot extends this to make auto-configuration, among others, possible.

Spring Boot makes it easy to create stand-alone, production-grade, Spring-based Applications that you can “just run.” We take an opinionated view of the Spring platform and third-party libraries so you can get started with minimum fuss. Most Spring Boot applications need very little Spring configuration.

From the Spring Boot Reference Guide

Spring Boot has auto-configuration for infrastructure like JMS, JDBC, JPA, RabbitMQ, and lots more. Spring Boot also offers auto-configuration for different frameworks like Spring Integration, Spring Batch, Spring Security, and many others. When these frameworks or capabilities are detected, Spring Boot will configure them with opinionated but sensible defaults.

The source code uses Maven for its build. Maven will take care of getting the necessary dependencies, compiling the code, and creating the artifact (generally a jar-file). Furthermore, if a recipe illustrates more than one approach, the source code is classified with various examples with roman letters (e.g., Recipe_2_1_i, Recipe_2_1_ii, Recipe_2_1_iii, etc.).

Tip To build each application, go inside the Recipe directory (e.g., ch2/recipe_2_1_i/) and execute the mvnw command to compile the source code. Once the source code is compiled, a target subdirectory is created with the application executable. You can then run the application JAR from the command line (e.g., java -jar target/Recipe_2_1_i.jar).

1-1 Create a Spring Boot Application Using Maven Problem

You want to start developing an application using Spring Boot and Maven.

Solution

Create a Maven build file, the `pom.xml`, and add the needed dependencies. To launch the application, create a Java class containing a `main` method to bootstrap the application.

How It Works

Suppose you are going to create a simple application that bootstraps a `SpringApplication` (the main entry point for a Spring Boot application), gets all the beans from the `ApplicationContext`, and outputs them to the console.

Create the `pom.xml`

Before you can start coding you need to create the `pom.xml` the file used by Maven to determine what needs to be done. The easiest way to use Spring Boot is by using the `spring-boot-starter-parent` as the parent for your application.

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.1.0.RELEASE</version>
  <relativePath />
</parent>
```

Next you need to add some Spring dependencies to get started using Spring; for this, add the `spring-boot-starter` as a dependency to your `pom.xml`.

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

Notice the fact that there is no version or other information needed; all this is managed because the `spring-boot-starter-parent` is used as the parent for the application. The `spring-boot-starter` will pull in all the core dependencies needed to start a very basic Spring Boot application, things like the Spring Framework, Logback for logging, and Spring Boot itself.

The full `pom.xml` should now look like this.

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

    <modelVersion>4.0.0</modelVersion>
    <groupId>com.apress.springboottutorial</groupId>
    <artifactId>chapter_1_1</artifactId>
    <version>2.0.0</version>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.1.0.RELEASE</version>
    </parent>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter</artifactId>
        </dependency>
    </dependencies>
</project>
```

Create the Application Class

Let's create a `DemoApplication` class with a `main` method. The `main` method calls `SpringApplication.run` with the `DemoApplication.class` and arguments from the `main` method. The `run` method returns an `ApplicationContext`, which is used to retrieve the bean names from `ApplicationContext`. The names are sorted and then printed to the console.

The resulting class would look like the following:

```
package com.apress.springbootrecipes.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

import java.util.Arrays;

@Configuration
@EnableAutoConfiguration
@ComponentScan
public class DemoApplication {

    public static void main(String[] args) {
        var ctx = SpringApplication.run(DemoApplication.class, args);

        System.out.println("# Beans: " + ctx.getBeanDefinitionCount());

        var names = ctx.getBeanDefinitionNames();
        Arrays.sort(names);
        Arrays.asList(names).forEach(System.out::println);
    }
}
```

This class is a regular Java class with a `main` method. You can run this class from your IDE. When the application runs it will show output similar to Figure 1-1.

Figure 1-1. Output of running application

What did happen with the code and annotations? The `@Configuration` annotation makes this class a Spring Java configuration class. Generally when creating an application you also have other components you need to have picked up; for this add the `@ComponentScan` annotation. Finally, to let Spring Boot do its auto-configuration, add the `@EnableAutoConfiguration` annotation.

Simplify the Application Class

If you look at the class definition, there are three annotations on it.

```
@Configuration  
@EnableAutoConfiguration  
@ComponentScan  
public class DemoApplication { ... }
```

When writing a Spring Boot-based application, most of those require all these annotations. This code can be simplified by using the `@SpringBootApplication` annotation instead. The class header then becomes

```
@SpringBootApplication  
public class DemoApplication { ... }
```

The `@SpringBootApplication` annotation is a so-called composed annotation and consists of the earlier needed annotations.

```
@Target({ElementType.TYPE})  
@Retention(RetentionPolicy.RUNTIME)  
@Documented  
@Inherited  
@SpringBootConfiguration  
@EnableAutoConfiguration  
@ComponentScan  
public @interface SpringBootApplication { ... }
```

There is one difference between the `@SpringBootApplication` and the earlier mentioned annotations. Here the `@SpringBootConfiguration` annotation is used instead of the `@Configuration` annotation. The `@SpringBootConfiguration` is a specialized `@Configuration` annotation. It indicates that this is a Spring Boot-based application. When using `@SpringBootConfiguration` in your application, there can only be one class annotated with this annotation!

1-2 Create a Spring Boot Application Using Gradle Problem

You want to start developing an application using Spring Boot and Gradle.

Solution

Create a Gradle build file, the `build.gradle`, and add the needed dependencies. To launch the application, create a Java class containing a `main` method to bootstrap the application.

How It Works

Suppose you are going to create a simple application that bootstraps a `SpringApplication`, gets all the beans from the `ApplicationContext`, and outputs them to the console.

Create the build.gradle

First you need to create a `build.gradle` and use the two plug-ins needed for Gradle to properly manage the dependencies for Spring Boot. Spring Boot requires a special Gradle plug-in (the Spring Boot Gradle plug-in) as well as a plug-in to extend the default dependency management capabilities of Gradle (the dependency management plug-in). To enable and configure these plug-ins, create a `buildscript` task in your `build.gradle`.

```
buildscript {
    ext {
        springBootVersion = '2.1.0.RELEASE'
    }
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-
            plugin:${springBootVersion}")
    }
}
```

This task will now properly configure the Spring Boot plug-in to use. Next you need to specify the plug-ins you want to use; as this is a Java-based project, you at least need the Java plug-in and as this book is about Spring Boot, you will also need the `org.springframework.boot` plug-in. Finally, you need to include the `io.spring.dependency-management` plug-in, for letting the Spring Boot Starters manage the dependencies.

```
apply plugin: 'java'
apply plugin: 'org.springframework.boot'
apply plugin: 'io.spring.dependency-management'
```

Finally you will need to add the needed dependencies; just as with Recipe 1.1, add the `spring-boot-starter` dependency.

```
dependencies {
    compile 'org.springframework.boot:spring-boot-starter'
```

Notice the absence of the specific version on the dependency. Not needing to specify the version, and have it automatically managed, is due to the usage of the `io.spring.dependency-management` plug-in which, just as with Maven, allows for easier dependency management.

The full `build.gradle` should now look something like this:

```
buildscript {  
    ext {  
        springBootVersion = '2.1.0.RELEASE'  
    }  
    repositories {  
        mavenCentral()  
    }  
    dependencies {  
        classpath("org.springframework.boot:spring-boot-gradle-  
            plugin:${springBootVersion}")  
    }  
}  
  
apply plugin: 'java'  
apply plugin: 'org.springframework.boot'  
apply plugin: 'io.spring.dependency-management'  
  
dependencies {  
    compile 'org.springframework.boot:spring-boot-starter'  
}  
  
repositories {  
    mavenCentral()  
}
```

Create the Application Class

Let's create a `DemoApplication` class with a `main` method. The `main` method calls `SpringApplication.run` with the `DemoApplication.class` and arguments from the `main` method. The `run` method returns an `ApplicationContext`, which is used to retrieve the bean names from `ApplicationContext`. The names are sorted and then printed to the console.

The resulting class would look like the following:

```
package com.apress.springbootrecipes.demo;

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

import java.util.Arrays;

@SpringBootApplication
public class DemoApplication {

    public static void main(String[] args) {
        var ctx = SpringApplication.run(DemoApplication.class, args);

        System.out.println("# Beans: " + ctx.getBeanDefinitionCount());

        var names = ctx.getBeanDefinitionNames();
        Arrays.sort(names);
        Arrays.asList(names).forEach(System.out::println);
    }
}
```

This class is a regular Java class with a `main` method. You can run this class from your IDE. When the application runs, it will show output similar to Figure 1-2.



Figure 1-2 shows the terminal output of a Spring Boot application. It starts with a decorative logo made of ASCII characters. Below the logo, the text ":: Spring Boot :: (v2.1.0.BUILD-SNAPSHOT)" is displayed. The log output includes several INFO messages from the DemoApplication class, showing the loading of various components like ThreadPoolTaskExecutor and AutoConfigurationPackages. At the bottom, it lists the number of beans found and some specific bean names.

```

:: Spring Boot :: (v2.1.0.BUILD-SNAPSHOT)

2018-08-31 09:02:38.014  INFO 39174 --- [           main] c.a.s.demo.DemoApplication
2018-08-31 09:02:38.017  INFO 39174 --- [           main] c.a.s.demo.DemoApplication
2018-08-31 09:02:38.568  INFO 39174 --- [           main] o.s.s.concurrent.ThreadPoolTaskExecutor
2018-08-31 09:02:38.596  INFO 39174 --- [           main] c.a.s.demo.DemoApplication
# Beans: 33
applicationTaskExecutor
demoApplication
mbeanExporter
mbeanServer
objectNamingStrategy
org.springframework.boot.autoconfigure.AutoConfigurationPackages

```

Figure 1-2. Output of running application

1-3 Create a Spring Boot Application Using Spring Initializr

Problem

You want to start a Spring Boot application using Spring Initializr.

Solution

Go to <http://start.spring.io>, select the Spring Boot version and the different dependencies you think you need, and download the project.

How It Works

First, go to <http://start.spring.io>, which will open the Spring Initializr (Figure 1-3)

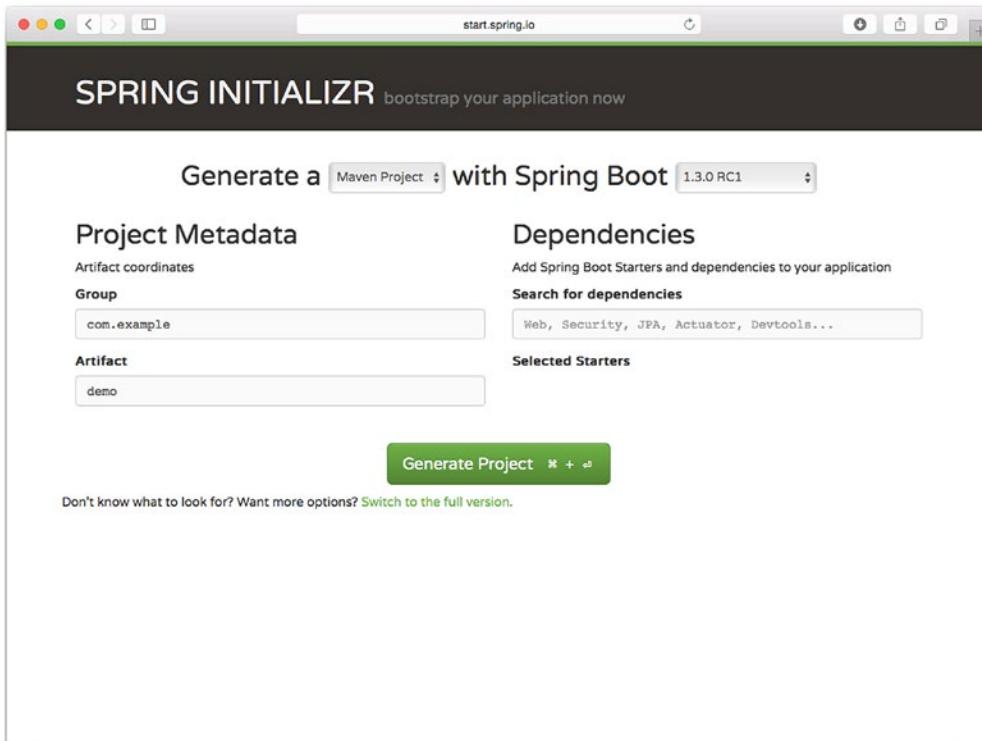


Figure 1-3. Spring Initializr

Now select what you want to generate (Maven or Gradle). Select the Spring Boot version you want to use; select the most recent one. Next, for the group enter `com.apress.springbootrecipes` and as artifact leave the default `demo` value (Figure 1-4).

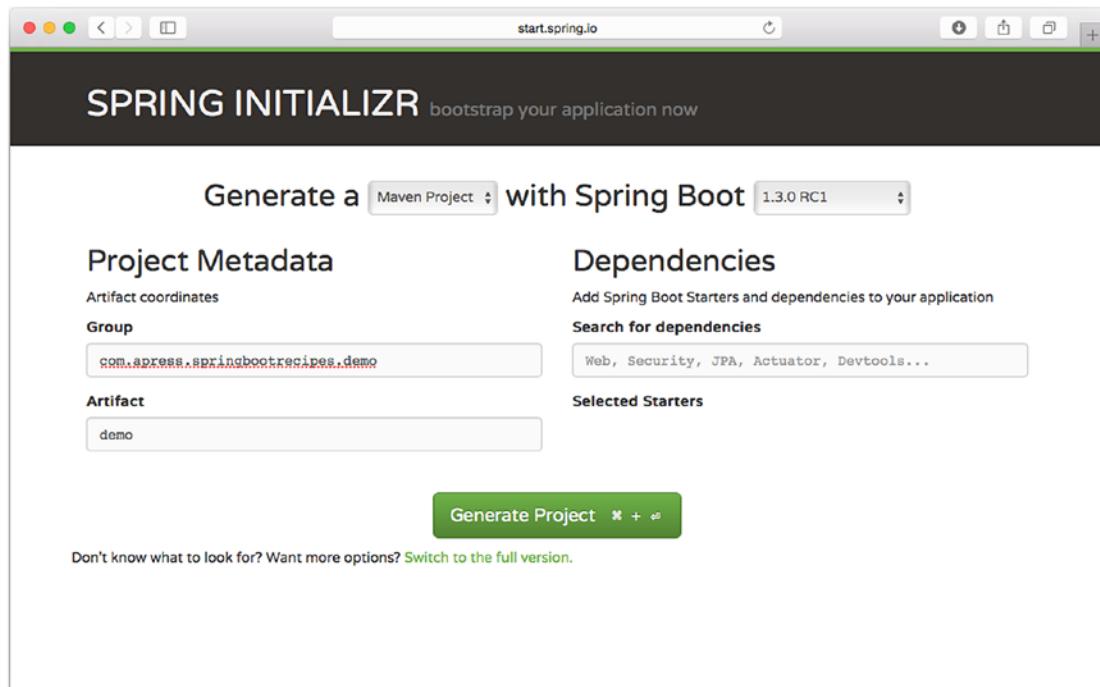


Figure 1-4. Spring Initializr with values

Finally click the Generate Project button; this will trigger a download of a `demo.zip`. Extract this zip file and import the project into your IDE. After importing, you should have a structure similar to that in Figure 1-5.

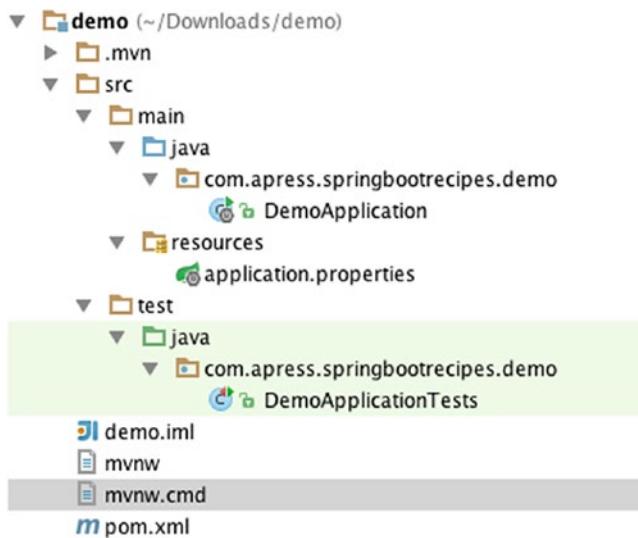


Figure 1-5. Imported project

Open the `pom.xml` and compare it with the one from Recipe 1.1 (or `build.gradle` from Recipe 1.2). It is quite similar; however, there are two differences to note. First there is an additional dependency, `spring-boot-starter-test`. This pulls in the needed test dependencies like Spring Test, Mockito, Junit 4, and AssertJ. With this single dependency you are ready to start testing.

The second difference is the fact that there is now a build section with the `spring-boot-maven-plugin` configured.

```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```

This plug-in takes care of creating a fat JAR. It takes the original JAR and repackages it with all the dependencies inside it. That way you can just hand over the JAR file to the operations team. The operations team needs to do `java -jar <your-application>.jar` to launch the application. No need to deploy it to a Servlet container or JEE container.

Implement a Simple Application

Open the DemoApplication and update the content to count and obtain the beans from the ApplicationContext.

```
package com.apress.springbootrecipes.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import java.util.Arrays;

@SpringBootApplication
public class DemoApplication {

    public static void main(String[] args) {
        var ctx = SpringApplication.run(DemoApplication.class, args);

        System.out.println("# Beans: " + ctx.getBeanDefinitionCount());

        var names = ctx.getBeanDefinitionNames();
        Arrays.sort(names);
        Arrays.asList(names).forEach(System.out::println);
    }
}
```

Building the JAR

When using the Spring Initializr, all projects come with the Maven Wrapper (or Gradle Wrapper when using Gradle) to make it easier to build the application. To use the wrapper script, open up a command line. Navigate to the directory the project is in. Finally, execute `./mvnw package` or `./gradlew build`. This should create the executable artifact in the target (or build/libs) directory.

Now that the JAR has been built, let's execute it and see what happens. Type `java -jar target/demo-0.0.1-SNAPSHOT.jar` (or `java -jar build/libs/demo-0.0.1-SNAPSHOT.jar`) and watch the application start and list the beans from the context (see Figures 1-1 and 1-2).

1-4 Summary

In this chapter you looked at how to bootstrap your development using Spring Boot. We looked at how to get started using Maven as well as Gradle, and finally we looked at how to get started using the Spring Initializer.

In the next chapter we will take a look at basic configuration of a Spring Boot application, how to define a bean, how to use property files, and how to override properties.

CHAPTER 2

Spring Boot—Basics

In this chapter we will take a look at the basic features of Spring Boot.

Note To get a starting point, use the Spring Initializr to create a project. No additional dependencies are required, just a Spring Boot project.

2-1 Configure a Bean

Problem

You want Spring Boot to use your class as a bean.

Solution

Depending on your needs, you can either leverage `@ComponentScan` to automatically detect your class and have an instance created; use it together with `@Autowired` and `@Value` to get dependencies or properties injected; or you can use a method annotated with `@Bean` to have more control over the construction of the bean being created.

How It Works

Recipe 1.1 explained that `@SpringBootApplication` includes both `@ComponentScan` and `@Configuration`. This means that any `@Component` annotated class will be automatically detected and instantiated by Spring Boot; it also allows for `@Bean` methods to be defined to declare beans.

Using @Component

First, create a class to bootstrap the application. Create a `HelloWorldApplication` that is annotated with `@SpringBootApplication`.

```
@SpringBootApplication
public class HelloWorldApplication {

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

Tip Place the `@SpringBootApplication` annotated class in a top-level package; this way it will automatically detect all your annotated components, configuration classes, etc. defined in this package and all subpackages.¹

This will bootstrap the application, detect all `@Component` annotated classes, and detect which libraries are on the classpath (see also Chapter 1). When running this `HelloWorldApplication` it won't do much, as there is nothing to detect or to run. Let's create a simple class that will be automatically detected by Spring Boot.

```
@Component
public class HelloWorld {

    @PostConstruct
    public void sayHello() {
        System.out.println("Hello World, from Spring Boot 2!");
    }
}
```

Spring Boot will detect this class and create a bean instance from it. The `@PostConstruct` annotated method is invoked after construction and injection of all dependencies. Simply put, at startup the `sayHello` method will run and the console will print the line `Hello World, from Spring Boot 2!`.

¹<https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-locating-the-main-class>

Important When you need to scan packages not covered by the default component scanning, you need to add a `@ComponentScan` annotation to your `@SpringBootApplication` annotated class so that those packages will be scanned as well. These packages are scanned in addition to the default scanning applied through the `@SpringBootApplication` annotation.

Using `@Bean` Method

Instead of automatically detecting components, you can also use a factory method to create beans. This is useful if you want or need more control over the construction of your bean. A factory method is a method annotated with `@Bean`² and it will be used to register a bean in the application context. The name of the bean is the same as the name of the method. The method can have arguments and those will be resolved to other beans in the application context.

Let's create an application that can do some basic calculations for integers. First let's write the `Calculator`; it will get a collection of `Operation` beans in the constructor. `Operation` is an interface, and the different implementations will do the actual calculation.

```
package com.apress.springbootrecipes.calculator;

import java.util.Collection;

public class Calculator {

    private final Collection<Operation> operations;

    public Calculator(Collection<Operation> operations) {
        this.operations = operations;
    }

    public void calculate(int lhs, int rhs, char op) {
        for (var operation : operations) {
```

²<https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-java-bean-annotation>

```

if (operation.handles(op)) {
    var result = operation.apply(lhs, rhs);
    System.out.printf("%d %s %d = %s%n", lhs, op, rhs, result);
    return;
}
}
throw new IllegalArgumentException("Unknown operation " + op);
}
}

```

In the calculate method, the correct Operation is detected using the Operation.handles method; when the correct one is found, the Operation.apply method is called to do the actual calculation. If we pass in an operation that the calculator cannot handle, an exception is thrown.

The Operation interface is a simple interface with the earlier mentioned two methods.

```

package com.apress.springbootrecipes.calculator;

public interface Operation {
    int apply(int lhs, int rhs);
    boolean handles(char op);
}

```

Now let's add two operations: one for adding values and one for multiplying values.

```

package com.apress.springbootrecipes.calculator.operation;

import com.apress.springbootrecipes.calculator.Operation;
import org.springframework.stereotype.Component;

@Component
class Addition implements Operation {
    @Override
    public int apply(int lhs, int rhs) {
        return lhs + rhs;
    }
}

```

```

@Override
public boolean handles(char op) {
    return '+' == op;
}
}

package com.apress.springbootrecipes.calculator.operation;

import com.apress.springbootrecipes.calculator.Operation;
import org.springframework.stereotype.Component;

@Component
class Multiplication implements Operation {

    @Override
    public int apply(int lhs, int rhs) {
        return lhs * rhs;
    }

    @Override
    public boolean handles(char op) {
        return '*' == op;
    }
}

```

These are all the components we need to make a calculator that can do additions and multiplications and still have an extensible mechanism.

Finally let's make an application that configures and uses the Calculator. To make an instance of the Calculator, a @Bean method is needed. This method can be added to the class annotated with @SpringBootApplication or a regular @Configuration.

```

package com.apress.springbootrecipes.calculator;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

import java.util.Collection;

@SpringBootApplication
public class CalculatorApplication {

```

```

public static void main(String[] args) {

    var ctx = SpringApplication.run(CalculatorApplication.class, args);

    var calculator = ctx.getBean(Calculator.class);
    calculator.calculate(137, 21, '+');
    calculator.calculate(137, 21, '*');
    calculator.calculate(137, 21, '-');

}

@Bean
public Calculator calculator(Collection<Operation> operations) {
    return new Calculator(operations);
}
}

```

The `calculator` factory method takes a `List<Operation>` and we use that to construct the `Calculator`. When using parameters in a `@Bean` annotated method, those will automatically be resolved; and when injecting a collection, Spring will automatically detect all instances of the beans required and use that to invoke the `calculator` factory method.

In the `main` method we retrieve the `Calculator` and call its `calculate` method with different numbers and operations. The first two will nicely print some output to the console; the last one will throw an exception, as there is no suitable operation to do subtractions.

Although you created a factory method for the `Calculator`, this isn't actually needed. When annotating the `Calculator` with `@Component`, Spring will detect it. The single constructor in the class will be used to construct it. When there are multiple constructors, annotate the one to use with `@Autowired`.

Another thing that isn't really nice about this code is that beans are retrieved manually and that could be considered as a bad practice; generally you want to use dependency injection. Spring Boot has an interface `ApplicationRunner`, which can be used to run some code after startup of the application. When Spring Boot detects a bean of type `ApplicationRunner`, it will invoke its `run` method as soon as the application has started.

```
package com.apress.springbootrecipes.calculator;

import org.springframework.boot.ApplicationRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
public class CalculatorApplication {

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

    @Bean
    public ApplicationRunner calculationRunner(Calculator calculator) {
        return args -> {
            calculator.calculate(137, 21, '+');
            calculator.calculate(137, 21, '*');
            calculator.calculate(137, 21, '-');
        };
    }
}
```

The method to create a `Calculator` has been replaced by an `ApplicationRunner`. This `ApplicationRunner` receives the automatically configured `Calculator` and runs some operations on it. When running this class, the output should still be the same as before. The major difference and advantage is that it is no longer needed to manually get beans from the `ApplicationContext`, as Spring will take care of getting the correct beans.

2-2 Externalize Properties

Problem

You want to use properties to configure your application for different environments or executions.

Solution

By default, Spring Boot supports getting properties from numerous locations. By default, it will load a file named `application.properties`, and use the environment variables and Java System properties. When running from the command line, it will also take command line arguments into consideration. There are more locations that are taken into account depending on the type of application and availability of capabilities (like JNDI, for instance).³ For our application the following resources are taken into considering in given order.

1. Command line arguments
2. `application.properties` outside of the packaged application
3. `application.properties` packaged inside the application

Next to that, for options 2 and 3 you can also load a profile-specific one based on the active profiles. The profiles to activate can be passed through the `spring.profiles.active` property. The profile-specific `application-{profile}.properties` takes precedence over the non-profile-specific one. Each will get loaded and with that you can override properties, which makes the list a bit longer.

1. Command line arguments
2. `application-{profile}.properties` outside the packaged application
3. `application.properties` outside the packaged application
4. `application-{profile}.properties` packaged inside the application
5. `application.properties` packaged inside the application

³<https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-external-config.html#boot-features-external-config>

How It Works

The Calculator as created for Recipe 2.1 itself is pretty flexible; however, the `CalculatorApplication` has hardcoded values when it comes to the calculations it does. Now when we want to calculate something different, we would need to modify the code, recompile, and run the newly compiled code. We want to use properties for this so that we are able to change them when needed.

First modify the application to use the value from properties instead of hardcoded values. For this, change the `@Bean` method for the `ApplicationRunner` to accept three additional parameters, and those parameters are going to be annotated with `@Value`.

```
package com.apress.springbootrecipes.calculator;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.ApplicationRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
public class CalculatorApplication {

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

    @Bean
    public ApplicationRunner calculationRunner(Calculator calculator,
                                              @Value("${lhs}") int lhs,
                                              @Value("${rhs}") int rhs,
                                              @Value("${op}") char op) {
        return args -> calculator.calculate(lhs, rhs, op);
    }
}
```

The `@Value` will instruct Spring to look up the property and use the value of that property. For instance if we would use a `@Value("${lhs}")`, Spring would try to detect a property named `lhs` and use the value. You could also specify a default value by adding a semicolon. With `@Value("${lhs:12}")`, if no value can be found it will use `12`. If there is no default value specified, an `IllegalArgumentException` will be thrown for missing properties. If we would now start the application, an exception would be thrown explaining that no property `lhs` could be found.

Add an `application.properties` in `src/main/resources` and put values in there for `lhs`, `rhs`, and `op`.

```
lhs=7
rhs=6
op=*
```

Spring Boot will load the `application.properties` at startup, and with that the properties are available. Now when running the application, it should again create output honoring the values given in the `application.properties`.

Although you now have externalized the properties into the application. properties, those are still packaged up inside the application, which would mean you still need to change them to do a different calculation. This would mean that for each combination of properties you would need a fresh build. Imagine doing this for a real production system and creating new artifacts just because your configuration needs to change. There are different ways in which Spring Boot can help you with that.

Override Using an External `application.properties`

First build the artifact and launch the application with `java -jar recipe_2_2--1.0.0.jar`. It will still run and use the supplied and packaged `application.properties`. Now, in the same location as the artifact add an `application.properties` and put values in there for the different properties.

```
lhs=26
rhs=952
op=*
```

When launching the application again, it will now use the values from this `application.properties`.

Override Properties Using Profiles

Spring Boot can use the active profiles to load additional configuration files, which can totally replace or override parts of the general configuration. Let's add an application-add.properties in the src/main/resources, which contains a different value for op.

```
op=+
```

Now build the artifact (a JAR file) and launch it from the command line with `java -jar recipe_2_2-1.0.0.jar --spring.profiles.active=add` and it will start and use the properties from both application.properties and application-add.properties to configure the application. Notice how an addition instead of multiplication is done, which indicates that the application-add.properties take precedence over the general application.properties.

Tip This also works when working with an external application.properties and application-{profile}.properties.

Override Properties Using Command Line Arguments

The last option is to use command line arguments to override properties; in the previous section you already used command line argument `--spring.profile.active=add` to specify the active profile. You can also specify the lhs and other arguments that way. Use `java -jar recipe_2_2-1.0.0.jar --lhs=12 --rhs=15 --op=+` to run the application, you will see that it does the calculation based on the arguments passed in through the command line. Arguments from the command line always override all other configuration.

Load Properties from Different Configuration File

If you are using another file than application.properties or you have some component that comes with an embedded file you want to load, you can always use an additional `@PropertySource` annotation on your `@SpringBootApplication` annotated class to load that additional file.

```
@PropertySource("classpath:your-external.properties")
@SpringBootApplication
public class MyApplication { ... }
```

The `@PropertySource` annotation allows you to add additional property files to be loaded during startup. Instead of using a `@PropertySource`, you can also instruct Spring Boot to load additional property files using the command line parameters from Table 2-1.

Table 2-1. Configuration Parameters

Parameter	Description
<code>spring.config.name</code>	Comma separated string of file names to load, default application
<code>spring.config.location</code>	Comma separated string of resource locations (or files) to consider for loading property files from, default <code>classpath:/,classpath:/config/,file:./,file:./config/</code>
<code>spring.config.additional-location</code>	Comma separated string of additional resource locations (or files) to consider for loading property files from, default empty

Warning When using `spring.config.location` or `spring.config.additional-location` with a file, this will be used as is and a profile specific one won't be loaded. When using a directory, then profile-specific files will be loaded.

To load the `your-external.properties` using `--spring.config.name=your-external` would be sufficient; however, this would break loading the `application.properties`. It is better to use `--spring.config.name=application,your-external`; now all the location will be searched for both `application.properties` and `your-external.properties` and the profile-specific versions will be taken into consideration.

2-3 Testing

Problem

You want to write a test for a component or part of your Spring Boot application.

Solution

Spring Boot extended the range of features of the Spring Test framework. It added support for mocking and spying on beans as well as provided auto-configuration for web tests. However, it also introduced easy ways of testing slices of your application by only bootstrapping that which is needed (through the use of @WebMvCTest or @JdbcTest for instance).

How It Works

Spring Boot extended the auto-configuration to parts of the test framework as well. It also integrates with Mockito⁴ for easy mocking (or spying) on beans. It also provides auto-configuration for web-based tests using either the Spring MockMvc testing framework or WebDriver based testing.

Writing a Unit Test

First, let's write a simple unit test for one of the components of the calculator, the `MultiplicationTest`, which will test the `Multiplication` class.

```
public class MultiplicationTest {

    private final Multiplication addition = new Multiplication();

    @Test
    public void shouldMatchOperation() {
        assertThat(addition.handles('*')).isTrue();
        assertThat(addition.handles('/')).isFalse();
    }

    @Test
    public void shouldCorrectlyApplyFormula() {
        assertThat(addition.apply(2, 2)).isEqualTo(4);
        assertThat(addition.apply(12, 10)).isEqualTo(120);
    }
}
```

⁴<https://site.mockito.org>

This is a basic unit test. The `Multiplication` is instantiated and we call methods on it and validate the outcome. The first test will test if it really reacts to the `*` operator and not something else. The second test will test the actual multiplication logic. To make a method a test method (for JUnit 4) you will have to annotate it with `@Test`.

Mocking Dependencies in a Unit Test

Sometimes a class needs dependencies; however, you want your test to test only a single component (when writing a unit test). Spring Boot automatically brings in the Mockito framework, which is very nice to mock classes and record behavior on it. Writing a test for the `Calculator` requires additional components, as it delegates the actual calculation to the available `Operation` classes. To test the correct behavior of the `Calculator` we would need to create a mock of the `Operation` and inject that into the `Calculator`.

```
public class CalculatorTest {

    private Calculator calculator;
    private Operation mockOperation;

    @Before
    public void setup() {
        mockOperation = Mockito.mock(Operation.class);
        calculator = new Calculator(Collections.
            singletonList(mockOperation));
    }

    @Test(expected = IllegalArgumentException.class)
    public void throwExceptionWhenNoSuitableOperationFound() {

        when(mockOperation.handles(anyChar())).thenReturn(false);
        calculator.calculate(2, 2, '*');
    }

    @Test
    public void shouldCallApplyMethodWhenSuitableOperationFound() {

        when(mockOperation.handles(anyChar())).thenReturn(true);
        when(mockOperation.apply(2, 2)).thenReturn(4);
    }
}
```

```

        calculator.calculate(2, 2, '*');

        verify(mockOperation, times(1)).apply(2,2);
    }
}

```

In the @Before class we mock the Operation by calling Mockito.mock and we construct the Calculator using the mocked operation. The mock is used in the test methods to have a certain behavior. In the first test method we want to test the situation where no suitable operation can be found, hence we instruct the mock to return false when the handles method is being called. The test will expect an exception; if the exception occurs the test will succeed. The second test is to check if the correct flow is followed; we want to test the correct behavior. The mock is instructed to return true for the handles method and a return value for the apply method.

Integration Testing with Spring Boot

Spring Boot provides several annotations to aid in testing. The first is @SpringBootTest, which will make the test a Spring Boot-driven test. This means that the test context framework will be searching for the class annotated with @SpringBootApplication (if no specific configuration is passed) and will use that to actually start the application.

```

@RunWith(SpringRunner.class)
@SpringBootTest(classes = CalculatorApplication.class)
public class CalculatorApplicationTests {

    @Autowired
    private Calculator calculator;

    @Test(expected = IllegalArgumentException.class)
    public void doingDivisionShouldFail() {
        calculator.calculate(12,13, '/');
    }
}

```

The preceding test will start the CalculatorApplication and inject the fully configured Calculator. We can then write a test, in this case a situation that the calculator cannot handle, and write expectations for it.

When doing a calculation, the output is printed to the console; using an `OutputCapture` JUnit rule, we could also write a success test and test the written output.

```
@RunWith(SpringRunner.class)
@SpringBootTest(classes = CalculatorApplication.class)
public class CalculatorApplicationTests {

    @Rule
    public OutputCapture capture = new OutputCapture();

    @Autowired
    private Calculator calculator;

    @Test
    public void doingMultiplicationShouldSucceed() {
        calculator.calculate(12,13, '*');
        capture.expect.Matchers.containsString("12 * 13 = 156"));
    }

    @Test(expected = IllegalArgumentException.class)
    public void doingDivisionShouldFail() {
        calculator.calculate(12,13, '/');
    }
}
```

The `@Rule` will configure the given JUnit rule, in this case the `OutputCapture` rule, which comes with Spring Boot and intercepts the `System.out` and `System.err` so that assertions can be written on the output generated on those streams. In this case we do a multiplication and the output should reflect that.

Integration Testing with Spring Boot and Mocks

Spring Boot makes it easy to replace a bean with a mock in the application context. For this, Spring Boot introduced the `@MockBean` annotation.

```
@MockBean
private Calculator calculator
```

This would replace the whole calculator with a mocked instance; to be able to use it you would need to define the behavior using the regular Mockito way. When there are multiple beans of a certain type, you need to specify the name of the bean you wish to replace.

```
@MockBean(name = "addition")
private Operation mockOperation;
```

This would replace the regular Addition bean with a mocked instance, which we can then use to register mock behavior on. When a bean with that name cannot be found, the mocked bean will be registered as a new instance of that bean.

```
@MockBean(name = "division")
private Operation mockOperation;
```

Would not replace an existing bean but add a bean to the application context and as a result the Calculator would have an additional operation added to the operations it can handle. You could test that using the ReflectionTestUtils helper class from the Spring Test framework.

```
@Test
public void calculatorShouldHave3Operations() {
    Object operations =
        ReflectionTestUtils.getField(calculator, "operations");
    assertThat((Collection) operations).hasSize(3);
}
```

This will obtain the operations field through reflection and assert that the collection size is 3. When the @MockBean annotation is removed (or the name changed to addition), this test will fail because there are now only two operations registered.

Using the mock is straightforward

```
@Test
public void mockDivision() {
    when(mockOperation.handles('/')).thenReturn(true);
    when(mockOperation.apply(14, 7)).thenReturn(2);

    calculator.calculate(14, 7, '/');
    capture.expect.Matchers.containsString("14 / 7 = 2"));
}
```

Here we instruct Mockito to return true when a / is being tested and to return a value when the apply method is called. Next, the method is called and we assert that the output is what is expected from this test.

2-4 Configure Logging Problem

You want to configure log levels for certain loggers.

Solution

With Spring Boot you can configure the logging framework and configuration.

How It Works

Spring Boot ships with a default configuration for the supported log providers (Logback,⁵ Log4j 2,⁶ and Java Util Logging). Next to the default configuration, it also adds support for configuring the logging levels through the regular application.properties as well as specifying patterns and where to, optionally, write log files to.

Spring Boot uses SLF4J as the logging API, and when writing components you should use those interfaces to write your logging. That way you have a choice of which logging framework to use.

Configure Logging

One of the general things to do with a logging framework is to enable or disable logging for parts of the framework. With Spring Boot you can do this by adding some lines to your application.properties. The lines need to be prefixed with logging.level followed by the name of the logger and finally the level you want it to be.

```
logging.level.org.springframework.web=DEBUG
```

⁵<https://logback.qos.ch>

⁶<https://logging.apache.org/log4j/2.x/>

The preceding line will enable DEBUG logging for the `org.springframework.web` logger (generally all classes in that package and subpackages). To set the level of the root logger, use `logging.level.root=<level>`. This will set the default level of logging.

Logging to File

By default, Spring Boot will only log to the console. If you want to write to a file as well, you need to specify either `logging.file` or `logging.path`. The first takes the name of the file; the second, the path. The default filename used is `spring.log` and the default directory used is the Java temp directory.

```
logging.file=application.log
logging.path=/var/log
```

With this configuration, a logfile named `application.log` will be written to the `/var/log` directory.

When writing logs to a file, you might want to prevent the logfiles from flooding your system. You can specify how many files to retain with the `logging.file.max-history` (default is 0 meaning unlimited) and `logging.file.max-size` to specify the file size (default is 10MB).

Using Your Preferred Logging Provider

Spring Boot by default uses Logback as the provider for the logging. It does, however, support Java Util Logging as well as Log4j 2. To use another logging framework, you will have to first exclude the default framework and include your own. Spring Boot has a `spring-boot-starter-log4j2` to include all necessary dependencies for Log4j 2. To exclude the default Logback logging, you need to add an exclusion rule to the `spring-boot-starter` dependency; this is the main dependency that brings in the logging.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter</artifactId>
  <exclusions>
    <exclusion>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-logging</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

```
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-log4j2</artifactId>
</dependency>
```

Note Spring Boot 2 doesn't support the older Log4j framework; it supports its successor, Log4j 2!

2-5 Reusing Existing Configuration Problem

You have an existing, non-Spring Boot, application or module and you want to reuse the configuration with Spring Boot.

Solution

To import an existing configuration, add the `@Import` or `@ImportResource` annotation to your `@Configuration` or `@SpringBootApplication` annotated class to import the configuration.

How It Works

On your main application class, the one with `@SpringBootApplication`, place the `@Import` or `@ImportResource` annotations to let Spring load the additional files.

Reusing Existing XML Configuration

Find the class with the `@SpringBootApplication` annotation and add the `@ImportResource` annotation to it.

```
@SpringBootApplication
@ImportResource("classpath:application-context.xml")
public class Application { ... }
```

This configuration will load the `application-context.xml` file from the classpath due to the `classpath:` prefix. If the file is somewhere in the file system, you can use the `file:` prefix, that is, `file:/var/conf/application-context.xml`.

When bootstrapping the application, Spring Boot will also load the additional configuration from the mentioned XML file.

Reusing Existing Java Configuration

Find the class with the `@SpringBootApplication` annotation and add the `@Import` annotation to it.

```
@SpringBootApplication  
@Import(ExistingConfiguration.class)  
public class Application { ... }
```

The `@Import` will take care of adding the mentioned class to the configuration. This can be needed if you want to include things not covered in the component scan or if you have disabled auto detection of `@Configuration` classes.

CHAPTER 3

Spring MVC

Spring Boot will automatically configure a web application when it finds the classes on the classpath. It will also start an embedded server (by default it will launch an embedded Tomcat)¹

3-1 Getting Started with Spring MVC Problem

You want to use Spring Boot to power a Spring MVC application.

Solution

Spring Boot will do auto-configuration for the components needed for Spring MVC. To enable this, Spring Boot needs to be able to detect the Spring MVC classes on its classpath. For this you will need to add the `spring-boot-starter-web` as a dependency.

How It Works

In your project, add the dependency for `spring-boot-starter-web`.

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

¹<https://tomcat.apache.org>

What this does is add the needed dependencies for Spring MVC. Now that Spring Boot can detect these classes, it will do additional configuration to set up the DispatcherServlet. It will also add all the JAR files needed to be able to start an embedded Tomcat server.

```
package com.apress.springbootrecipes.hello;

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

@SpringBootApplication
public class HelloWorldApplication {

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

These few lines of code are enough to start the embedded Tomcat server and have a preconfigured Spring MVC setup. When you start the application, you will see output similar to that in Figure 3-1.

```
:[ Spring Boot :: (v2.1.0.BUILD-SNAPSHOT)

2018-08-31 14:56:13.153 INFO [2270] --- [           main] c.a.s.library.HelloWorldApplication      : Starting HelloWorldApplication v2.0.0-SNAPSHOT on imac-van-marten.home
/target/spring-mvc-getting-started-2.0.0-SNAPSHOT.jar started by marten in /Users/marten/Repositories/spring-boot-recipes/code/ch03/recipe_3_1
2018-08-31 14:56:13.157 INFO [2270] --- [           main] c.a.s.library.HelloWorldApplication      : No active profile set, falling back to default profiles: default
2018-08-31 14:56:14.555 INFO [2270] --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
2018-08-31 14:56:14.578 INFO [2270] --- [           main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2018-08-31 14:56:14.578 INFO [2270] --- [           main] org.apache.catalina.core.StandardEngine : Starting Servlet Engine: Apache Tomcat/9.0.10
2018-08-31 14:56:14.592 INFO [2270] --- [           main] o.a.catalina.core.AprLifecycleListener : The APR based Apache Tomcat Native library which allows optimal performance in Java/Extensions:/Library/Java/Extensions:/System/Library/Java/Extensions:/usr/lib/java:.
2018-08-31 14:56:14.605 INFO [2270] --- [           main] o.a.c.c.c.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2018-08-31 14:56:14.605 INFO [2270] --- [           main] o.s.web.context.ContextLoader            : Root WebApplicationContext: initialization completed in 1439 ms
2018-08-31 14:56:14.724 INFO [2270] --- [           main] o.s.b.w.servlet.ServletRegistrationBean  : Servlet dispatcherServlet mapped to [/]
2018-08-31 14:56:14.729 INFO [2270] --- [           main] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'characterEncodingFilter' to: [//*]
2018-08-31 14:56:14.729 INFO [2270] --- [           main] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'hiddenHttpMethodFilter' to: [//*]
2018-08-31 14:56:14.729 INFO [2270] --- [           main] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'formContentFilter' to: [//*]
2018-08-31 14:56:14.729 INFO [2270] --- [           main] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'requestContextFilter' to: [//*]
2018-08-31 14:56:14.786 INFO [2270] --- [           main] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
2018-08-31 14:56:15.165 INFO [2270] --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''
2018-08-31 14:56:15.171 INFO [2270] --- [           main] c.a.s.library.HelloWorldApplication      : Started HelloWorldApplication in 2.508 seconds (JVM running for 3.054)
```

Figure 3-1. Startup output logging

The following things happen when you start the HelloWorldApplication:

1. Start an embedded Tomcat server on port 8080 (by default)
2. Register and enable a couple of default Servlet Filters (Table 3-1)

3. Set up static resource handling for things like .css, .js and favicon.ico
4. Enable integration with WebJars²
5. Setup basic error handling features
6. Preconfigure the DispatcherServlet with the needed components (i.e., ViewResolvers, I18N, etc.)

Table 3-1. Automatically Registered Servlet Filters

Filter	Description
CharacterEncodingFilter	Will force the encoding to be UTF-8 by default, can be configured by setting the <code>spring.http.encoding.charset</code> property. Can be disabled through setting the <code>spring.http.encoding.enabled</code> to false
HiddenHttpMethodFilter	Enables the use of hidden form field named <code>_method</code> to specify the actual HTTP method. Can be disabled by setting the <code>spring.mvc.hiddenmethod.filter.enabled</code> to false
FormContentFilter	Will wrap the request for PUT, PATCH, and DELETE requests so that those can also benefit from binding. Can be disabled by setting the <code>spring.mvc.formcontent.filter.enabled</code> to false
RequestContextFilter	Exposes the current request to the current thread so that you can use the <code>RequestContextHolder</code> and <code>LocaleContextHolder</code> even in a non-Spring MVC application like Jersey

In the current state, the `HelloWorldApplication` doesn't do anything but start the server. Let's add a controller to return some information.

```
package com.apress.springbootrecipes.hello;

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

²<https://www.webjars.org>

```

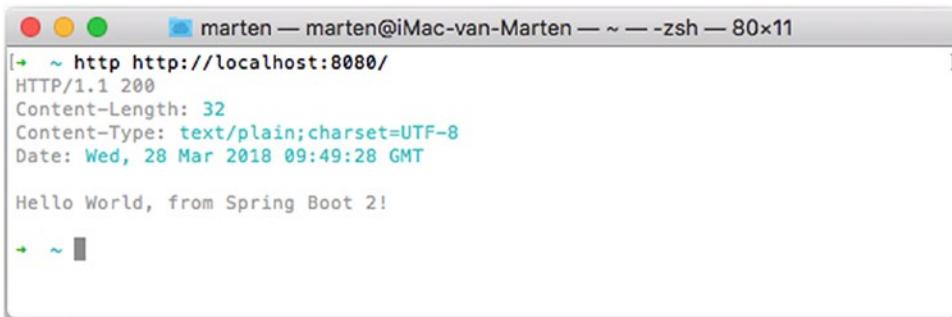
@RestController
public class HelloWorldController {

    @GetMapping("/")
    public String hello() {
        return "Hello World, from Spring Boot 2!";
    }
}

```

This `HelloWorldController` will be registered at the `/` URL and when called will return the phrase `Hello World from Spring Boot 2!`. The `@RestController` indicates that this is a `@Controller` and as such will be detected by Spring Boot. Additionally, it adds the `@ResponseBody` annotation to all request handling methods, indicating it should send the result to the client. The `@GetMapping` maps the `hello` method to every GET request that arrives at `/`. We could also have written `@RequestMapping(value="/", method=RequestMethod.GET)`.

When restarting the `HelloWorldApplication`, the `HelloWorldController` will be detected and processed. Now when using something like `curl` or `http` to access `http://localhost:8080/` the result should be like that in Figure 3-2.



```

marten — marten@iMac-van-Marten — ~ — zsh — 80x11
[+ ~ http http://localhost:8080/
HTTP/1.1 200
Content-Length: 32
Content-Type: text/plain; charset=UTF-8
Date: Wed, 28 Mar 2018 09:49:28 GMT

Hello World, from Spring Boot 2!

```

Figure 3-2. Output of controller

Testing

Now that the application is running and returning results, it's time to add a test for the controller (ideally you would write the test first!). Spring already has some impressive testing features and Spring Boot adds more of those. Testing a controller has become pretty easy with Spring Boot.

```
package com.apress.springbootrecipes.hello;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.http.MediaType;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.request.MockMvcRequestBuilders;

import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.content;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

@RunWith(SpringRunner.class)
@WebMvcTest(HelloWorldController.class)
public class HelloWorldControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @Test
    public void testHelloWorldController() throws Exception {
        mockMvc.perform(MockMvcRequestBuilders.get("/"))
            .andExpect(status().isOk())
            .andExpect(content().string("Hello World, from Spring Boot 2!"))
            .andExpect(content().contentTypeCompatibleWith(MediaType.TEXT_PLAIN));
    }
}
```

The `@RunWith(SpringRunner.class)` is needed to instruct JUnit to use this specific runner. This special runner is what integrates the Spring Test framework with JUnit. The `@WebMvcTest` instructs the Spring Test framework to set up an application context for testing this specific controller. It will start a minimal Spring Boot application with only the web-related beans like `@Controller`, `@ControllerAdvice`, etc. In addition it will preconfigure the Spring Test Mock MVC support, which can then be autowired.

Spring Test Mock MVC can be used to simulate making an HTTP request to the controller and do some expectations on the result. Here we call the / with GET and expect an HTTP 200 (thus OK) response with the plain text message of Hello World, from Spring Boot 2!.

3-2 Exposing REST Resources with Spring MVC Problem

You want to use Spring MVC to expose REST based resources @WebMvcTest.

Solution

You will need a JSON library to do the JSON marshalling (although you could use XML and other formats as well, as content negotiation³ is part of REST). In this recipe we will use the Jackson⁴ library to take care of the JSON conversion.

How It Works

Imagine you are working for a library and you need to develop a REST API to make it possible to list and search books.

The spring-boot-starter-web dependency (see also Recipe 3.1) already includes the needed Jackson libraries by default.

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

Note You can also use the Google GSON library; just use the appropriate GSON dependency instead.

³https://www.ics.uci.edu/~fielding/pubs/dissertation/evaluation.htm#sec_6_3_2_7

⁴<https://github.com/FasterXML/jackson>

As you are making an application for a library it will probably include books, so let's create a Book class.

```
package com.apress.springbootrecipes.library;

import java.util.*;

public class Book {

    private String isbn;
    private String title;
    private List<String> authors = new ArrayList<>();

    public Book() {}

    public Book(String isbn, String title, String... authors) {
        this.isbn = isbn;
        this.title = title;
        this.authors.addAll(Arrays.asList(authors));
    }

    public String getIsbn() {
        return isbn;
    }

    public void setIsbn(String isbn) {
        this.isbn = isbn;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public void setAuthors(List<String> authors) {
        this.authors = authors;
    }
}
```

```

public List<String> getAuthors() {
    return Collections.unmodifiableList(authors);
}

@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Book book = (Book) o;
    return Objects.equals(isbn, book.isbn);
}

@Override
public int hashCode() {
    return Objects.hash(isbn);
}

@Override
public String toString() {
    return String.format("Book [isbn=%s, title=%s, authors=%s]",
        this.isbn, this.title, this.authors);
}
}

```

A book is defined by its ISBN number; it has a title and 1 or more authors.

You would also need a service to work with the books in the library. Let's define an interface and implementation for the BookService.

```

package com.apress.springrecipes.library;

import java.util.Optional;

public interface BookService {

    Iterable<Book> findAll();
    Book create(Book book);
    Optional<Book> find(String isbn);
}

```

The implementation, for now, is a simple in-memory implementation.

```
package com.apress.springbootrecipes.library;

import org.springframework.stereotype.Service;

import java.util.Map;
import java.util.Optional;
import java.util.concurrent.ConcurrentHashMap;

@Service
class InMemoryBookService implements BookService {

    private final Map<String, Book> books = new ConcurrentHashMap<>();

    @Override
    public Iterable<Book> findAll() {
        return books.values();
    }

    @Override
    public Book create(Book book) {
        books.put(book.getIsbn(), book);
        return book;
    }

    @Override
    public Optional<Book> find(String isbn) {
        return Optional.ofNullable(books.get(isbn));
    }
}
```

The service has been annotated with `@Service` so that Spring Boot will detect it and create an instance of it.

```
package com.apress.springbootrecipes.library;

import org.springframework.boot.ApplicationRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
```

```

@SpringBootApplication
public class LibraryApplication {

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

    @Bean
    public ApplicationRunner booksInitializer(BookService bookService) {
        return args -> {
            bookService.create(
                new Book("9780061120084", "To Kill a Mockingbird", "Harper Lee"));
            bookService.create(
                new Book("9780451524935", "1984", "George Orwell"));
            bookService.create(
                new Book("9780618260300", "The Hobbit", "J.R.R. Tolkien"));
        };
    }
}

```

The LibraryApplication will detect all the classes and start the server. Upon starting, it will preregister three books so that we have something in our library.

To expose the Book as a REST resource, create a class, BookController, and annotate it with `@RestController`. Spring Boot will detect this class and create an instance of it. Using `@RequestMapping` (and `@GetMapping` and `@PostMapping`) you can write methods to handle the incoming requests.

Note Instead of `@RestController` you could also use `@Controller` and put `@ResponseBody` on each request handling method. Using `@RestController` will implicitly add `@ResponseBody` to request handling methods.

```

package com.apress.springbootrecipes.library.rest;

import com.apress.springbootrecipes.library.Book;
import com.apress.springbootrecipes.library.BookService;
import org.springframework.http.ResponseEntity;

```

```
import org.springframework.web.bind.annotation.*;
import org.springframework.web.util.UriComponentsBuilder;

import java.net.URI;

@RestController
@RequestMapping("/books")
public class BookController {

    private final BookService bookService;

    public BookController(BookService bookService) {
        this.bookService = bookService;
    }

    @GetMapping
    public Iterable<Book> list() {
        return bookService.findAll();
    }

    @GetMapping("/{isbn}")
    public ResponseEntity<Book> get(@PathVariable("isbn") String isbn) {
        return bookService.find(isbn)
            .map(ResponseEntity::ok)
            .orElse(ResponseEntity.notFound().build());
    }

    @PostMapping
    public Book create(@RequestBody Book book,
                       UriComponentsBuilder uriBuilder) {
        Book created = bookService.create(book);
        URI newBookUri = uriBuilder.path("/books/{isbn}").build(created.getIsbn());
        return ResponseEntity.created(newBookUri).body(created);
    }
}
```

The controller will be mapped to the /books path due to the @RequestMapping (" /books ") annotation on the class. The list method will be invoked for GET requests on /books. When /books/<isbn> is called with a GET request, the get method will be invoked and return the result for a single book or, when no book can be found, a 404 response status. Finally, you can add books to the library using a POST request on /books; then the create method will be invoked and the body of the incoming request will be converted into a book.

After the application has started, you can use HTTPie⁵ or cURL⁶ to retrieve the books. When using HTTPie to access http://localhost:8080/books, you should see output similar to that of Figure 3-3.



```
[+ target git:(master) ✘ http :8080/books
HTTP/1.1 200
Content-Type: application/json;charset=UTF-8
Date: Wed, 04 Apr 2018 07:50:40 GMT
Transfer-Encoding: chunked

[{"id": 1, "title": "The Hobbit", "isbn": "9780618260300", "authors": ["J.R.R. Tolkien"]}, {"id": 2, "title": "1984", "isbn": "9780451524935", "authors": ["George Orwell"]}, {"id": 3, "title": "To Kill a Mockingbird", "isbn": "9780061120084", "authors": ["Harper Lee"]}]
```

Figure 3-3. JSON output for list of books

⁵<https://httpie.org>

⁶<https://curl.haxx.se>

A request to `http://localhost:8080/books/9780451524935` will give you the result of a single book, in this case for **1984** by George Orwell. Using an unknown ISBN will result in a 404.

When issuing a POST request, we could add a new book to the list.

```
http POST :8080/books \
  title="The Lord of the Rings" \
  isbn="9780618640157" \
  authors:='["J.R.R. Tolkien"]'
```

The result of this call, when done correctly, is the freshly added book and a location header. Now when you get the list of books it should contain four books instead of the three books you started with.

What happens is that HTTPie translates the parameters into a JSON request body, which in turn is read by the Jackson library and turned into a Book.

```
{
  "title": "The Lord of the Rings",
  "isbn": "9780618640157",
  "authors": ["J.R.R. Tolkien"]
}
```

By default Jackson will use the getter and setter to map the JSON to an object. What happens is that a new Book instance is created using the default no-args constructor and all properties are set through the setters. For the title property, the setTitle is called, etc.

Testing a @RestController

As you want to make sure that the controller does what it is supposed to do, write a test to verify the correct behavior of the controller.

```
Package com.apress.springbootrecipes.library.rest;

import com.apress.springbootrecipes.library.Book;
import com.apress.springbootrecipes.library.BookService;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
```

CHAPTER 3 SPRING MVC

```
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.result.MockMvcResultMatchers;

import java.util.Arrays;
import java.util.Optional;

import static org.hamcrest.Matchers.*;
import static org.mockito.ArgumentMatchers.anyString;
import static org.mockito.Mockito.when;
import static org.springframework.test.web.servlet.request.
MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.
MockMvcResultMatchers.status;

@RunWith(SpringRunner.class)
@WebMvcTest(BookController.class)
public class BookControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private BookService bookService;

    @Test
    public void shouldReturnListOfBooks() throws Exception {

        when(bookService.findAll()).thenReturn(Arrays.asList(
            new Book("123", "Spring 5 Recipes", "Marten Deinum", "Josh Long"),
            new Book("321", "Pro Spring MVC", "Marten Deinum", "Colin Yates")));

        mockMvc.perform(get("/books"))
            .andExpect(status().isOk())
            .andExpect(jsonPath("$.size()", hasSize(2)))
            .andExpect(jsonPath("$[*].isbn", containsInAnyOrder("123", "321")))
            .andExpect(jsonPath("$[*].title",
                containsInAnyOrder("Spring 5 Recipes", "Pro Spring MVC")));
    }
}
```

```

@Test
public void shouldReturn404WhenBookNotFound() throws Exception {
    when(bookService.find(anyString())).thenReturn(Optional.empty());

    mockMvc.perform(get("/books/123")).andExpect(status().isNotFound());
}

@Test
public void shouldReturnBookWhenFound() throws Exception {
    when(bookService.find(anyString())).thenReturn(
        Optional.of(
            new Book("123", "Spring 5 Recipes", "Marten Deinum", "Josh Long")));

    mockMvc.perform(get("/books/123"))
        .andExpect(status().isOk())
        .andExpect(jsonPath("$.isbn", equalTo("123")))
        .andExpect(jsonPath("$.title", equalTo("Spring 5 Recipes")));
}
}

```

The test uses `@WebMvcTest` to create a `MockMvc`-based test and will create a minimal Spring Boot application to be able to run the controller. The controller needs an instance of a `BookService`, so we let the framework create a mock for this using the `@MockBean` annotation. In the different test methods, we mock the expected behavior (like returning a list of books, returning an empty `Optional`, etc.).

Note Spring Boot uses Mockito⁷ to create mocks using `@MockBean`.

Furthermore, the test uses the `JsonPath`⁸ library so that you can use expressions to test the JSON result. `JsonPath` is for JSON what `Xpath` is for XML.

⁷<https://site.mockito.org>

⁸<https://github.com/json-path/JsonPath>

3-3 Using Thymeleaf with Spring Boot Problem

You want to use Thymeleaf to render the pages of your application.

Solution

Add the dependency for Thymeleaf and create a regular @Controller to determine the view and fill the model.

How It Works

To get started you will first need to add the `spring-boot-starter-thymeleaf` as a dependency to your project to get the desired Thymeleaf⁹ dependencies.

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

With the addition of this dependency, you will get the Thymeleaf library as well as the Thymeleaf Spring Dialect so that the two integrate nicely. Due to the existence of these two libraries, Spring Boot will automatically configure the `ThymeleafViewResolver`.

The `ThymeleafViewResolver` requires a Thymeleaf `ItemplateEngine` to be able to resolve and render the views. A special `SpringTemplateEngine` will be preconfigured with the `SpringDialect` so that you can use SpEL inside Thymeleaf pages.

To configure Thymeleaf, Spring Boot exposes several properties in the `spring.thymeleaf` namespace (Table 3-2).

⁹<https://www.thymeleaf.org>

Table 3-2. Thymeleaf Properties

Property	Description
spring.thymeleaf.prefix	The prefix to use for the ViewResolver, default classpath:/templates/
Spring.thymeleaf.suffix	The suffix to use for the ViewResolver, default .html
spring.thymeleaf.encoding	The encoding of the templates, default UTF-8
spring.thymeleaf.check-template	Check if the template exists before rendering, default true.
Spring.thymeleaf.check-template-location	Check if the template location exists, the default is true.
Spring.thymeleaf.mode	Thymeleaf TemplateMode to use, default HTML
Spring.thymeleaf.cache	Should resolved templates be cached or not, default true
Spring.thymeleaf.template-resolver-order	Order of the ViewResolver default is 1.
Spring.thymeleaf.view-names	The view names (comma separated) that can be resolved with this ViewResolver
spring.thymeleaf.excluded-view-names	The view names (comma separated) that are excluded from being resolved
Spring.thymeleaf.enabled	Should Thymeleaf be enabled, default true
Spring.thymeleaf.enable-spring-el-compiler	Enable the compilation of SpEL expressions, default false
Spring.thymeleaf.servlet.content-type	Content-Type value used to write the HTTP Response, default text/html

Adding an Index Page

First add an index page to the application. Create an `index.html` inside the `src/main/resources/templates` directory (the default location).

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
```

```
<head>
    <meta charset="UTF-8">
    <title>Spring Boot Recipes - Library</title>
</head>
<body>
    <h1>Library</h1>
    <a th:href="@{/books.html}" href="#">List of books</a>
</body>
</html>
```

This is just a basic HTML5 page with some minor additions for Thymeleaf. First there is `xmlns:th="http://www.thymeleaf.org"` to enable the namespace for Thymeleaf. The namespace is used in the link through `th:href`. The `@{/books.html}` will be expanded, by Thymeleaf, to a proper link and placed in the actual `href` attribute of the link.

Now when running the application and visiting the homepage (`http://localhost:8080/`), you should be greeted by a page with a link to the books overview (Figure 3-4).

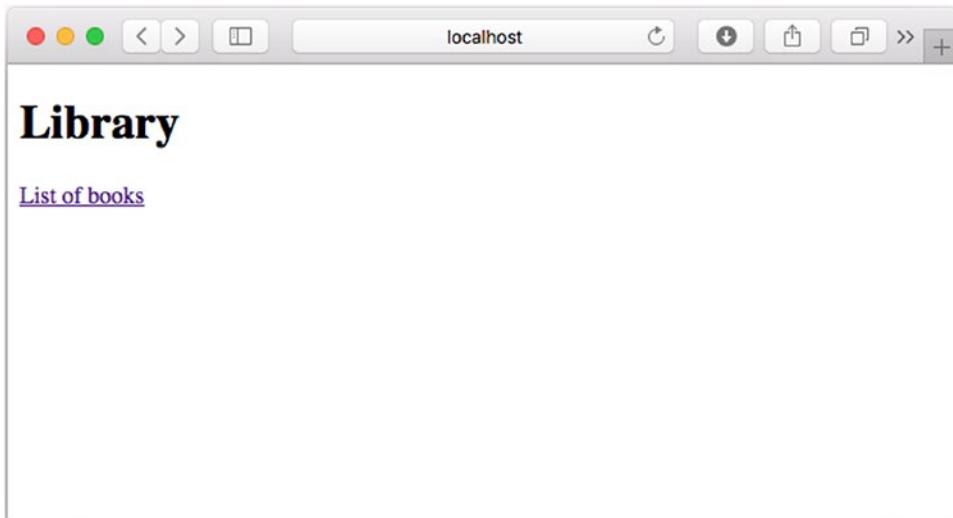


Figure 3-4. Rendered index page

Adding a Controller and View

When clicking the link provided in the index page, we want to be shown a page that shows a list of available books in the library (Figure 3-5). For this, two things need to be added: first a controller that can handle the request and prepare the model, and second a view to render the list of books.

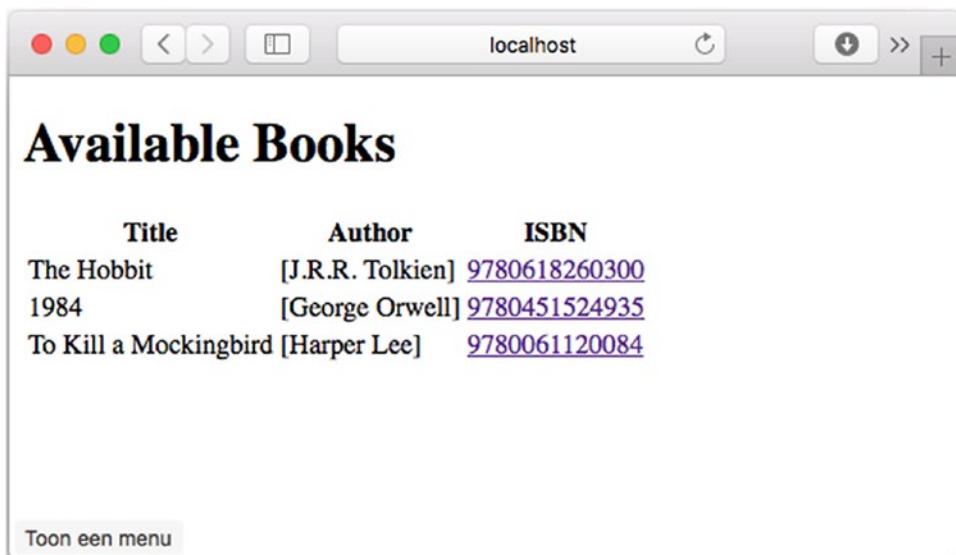


Figure 3-5. Books list page

Let's add a controller that will fill the model with a list of books and select the name of the view to render. A controller is a class annotated with `@Controller` and which contains request handling methods (methods annotated with `@RequestMapping` or as in this recipe `@GetMapping`, which is a specialized `@RequestMapping` annotation).

```
package com.apress.springboottrecipes.library.web;  
  
@Controller  
public class BookController {  
  
    private final BookService bookService;  
  
    public BookController(BookService bookService) {  
        this.bookService = bookService;  
    }  
}
```

```

@GetMapping("/books.html")
public String all(Model model) {
    model.addAttribute("books", bookService.findAll());
    return "books/list";
}
}

```

The BookController needs the BookService so that it can obtain a list of books to show. The all method has an `org.springframework.ui.Model` as method argument so that we can put the list of books in the model. A request handling method can have different arguments¹⁰; one of them is the `Model` class. In the all method, we use the BookService to retrieve all the books from the datastore and add it to the model using `model.addAttribute`. The list of books is now available in the model under the key books.

Finally, we return the name of the view to render `books/list`. This name is passed on to the `ThymeleafViewResolver` and will result in a path to `classpath:/templates/books/list.html`.

Now that the controller together with the request handling method has been added, we need to create the view. Create a `list.html` in the `src/main/templates/books` directory.

```

<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head lang="en">
    <meta charset="UTF-8">
    <title>Library - Available Books</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
</head>
<body>
    <h1>Available Books</h1>
    <table>
        <thead>
            <tr>
                <th>Title</th>

```

¹⁰<https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html#mvc-ann-arguments>

```

<th>Author</th>
<th>ISBN</th>
</tr>
</thead>
<tbody>
<tr th:each="book : ${books}">
<td th:text="${book.title}">Title</td>
<td th:text="${book.authors}">Authors</td>
<td>
<a th:href="@{/books.html(isbn=${book.isbn})}" href="#" th:text="${book.isbn}">1234567890123</a>
</td>
</tr>
</tbody>
</table>
</body>
</html>

```

This is again an HTML5 page using the Thymeleaf syntax. The page will render a list of books using the `th:each` expression. It will take all the books from the `books` property in the model and for each book create a row. Each column in the row will contain some text using the `th:text` expression; it will print the title, authors, and ISBN of the book. The final column in the table contains a link to the book details. It constructs a URL using the `th:href` expression. Notice the expression between `()`; this will add the `isbn` request parameter.

When launching the application and clicking the link on the index page, you should be greeted with a page showing the contents of the library as shown in Figure 3-5.

Adding a Details Page

Finally, when clicking the ISBN number in the table, you want a page with details to be shown. The link contains a request parameter named `isbn`, which we can retrieve and use in the controller to find a book. The request parameter can be retrieved through a method argument annotated with `@RequestParam`.

The following method will handle the GET request, map the request parameter to the method argument, and includes the model so that we can add the book to the model.

```

@GetMapping(value = "/books.html", params = "isbn")
public String get(@RequestParam("isbn") String isbn, Model model) {
    bookService.find(isbn)
        .ifPresent(book -> model.addAttribute("book", book));
    return "books/details";
}

```

The controller will render the books/details page. Add the details.html to the src/main/resources/templates/books directory.

```

<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head lang="en">
    <meta charset="UTF-8">
    <title>Library - Available Books</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
</head>
<body>
    <div th:if="${book != null}">
        <div>
            <div th:text="${book.title}">Title</div>
            <div th:text="${book.authors}">authors</div>
            <div th:text="${book.isbn}">ISBN</div>
        </div>
    </div>

    <div th:if="${book} == null">
        <h1 th:text="'No book found with ISBN: ' + ${param.isbn}">Not
        Found</h1>
    </div>
</body>
</html>

```

This HTML5 Thymeleaf template will render one of the two available blocks on the page. Either the book has been found and it will display the details, else it will show a not found message. This is achieved by using the `th:if` expression. The `isbn` for the not found message is retrieved from the request parameters using the `param` as a prefix. `${param.isbn}` will get the `isbn` request parameter.

3-4 Handling Exceptions

Problem

You want to customize the default white label error page shown by Spring Boot.

Solution

Add an additional `error.html` as a customized error page, or specific error pages for specific HTTP error codes (i.e., `404.html` and `500.html`).

How It Works

Spring Boot by default comes with the error handling enabled and will show a default error page. This can be disabled in full by setting the `server.error.whitelabel.enabled` property to `false`. When disabled, the exception handling will be handled by the Servlet Container instead of the general exception handling mechanism provided by Spring and Spring Boot.

There are also some other properties that can be used to configure the `whitelabel` error page, mainly on what is going to be included in the model so that it, optionally, could be used to display. See Table 3-3 for the properties.

Table 3-3. Error Handling Properties

Property	Description
<code>server.error.whitelabel.enabled</code>	Is the <code>whitelabel</code> error page enabled, default <code>true</code>
<code>server.error.path</code>	The path of the error page, default <code>/error</code>
<code>server.error.include-exception</code>	Should the name of the exception be included in the model, default <code>false</code>
<code>server.error.include-stacktrace</code>	Should the stacktrace be included in the model, default <code>never</code>

First let's add a method to the BookController that forces an exception.

```
@GetMapping("/books/500")
public void error() {
    throw new NullPointerException("Dummy NullPointerException.");
}
```

This will throw an exception and as a result the whitelabel error page will be shown when visiting <http://localhost:8080/books/500> (see Figure 3-6).

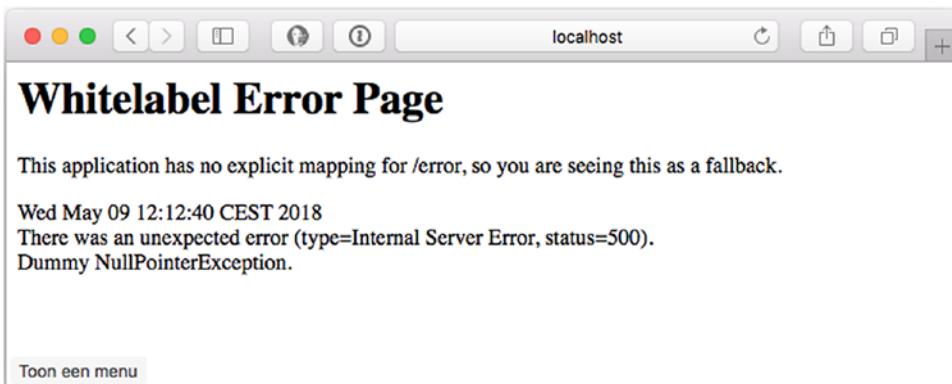


Figure 3-6. Default error page

This is shown if no error page can be found. To override this, add an `error.html` to the `src/main/resources/templates` directory.

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Spring Boot Recipes - Library</title>
</head>
<body>
    <h1>Oops something went wrong, we don't know what but we are going to work on it!</h1>
    <div>
        <div>
            <span><strong>Status</strong></span>
            <span th:text="${status}"></span>
        </div>
    </div>

```

```
</div>
<div>
    <span><strong>Error</strong></span>
    <span th:text="${error}"></span>
</div>
<div>
    <span><strong>Message</strong></span>
    <span th:text="${message}"></span>
</div>
<div th:if="${exception != null}">
    <span><strong>Exception</strong></span>
    <span th:text="${exception}"></span>
</div>
<div th:if="${trace != null}">
    <h3>Stacktrace</h3>
    <span th:text="${trace}"></span>
</div>
</div>
</body>
</html>
```

Now when the application is started and an exception occurs, this custom error page will be shown (Figure 3-7). The page will be rendered by the view technology of your choice (in this case it uses Thymeleaf).

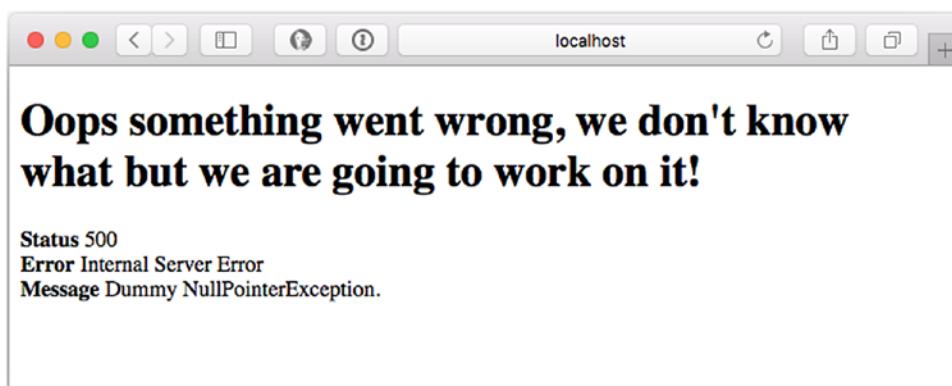


Figure 3-7. Custom error page

If you now set the `server.error.include-exception` to true and `server.error.include-stacktrace` to always, the customized error page will also include the classname of the exception and the stacktrace (Figure 3-8).

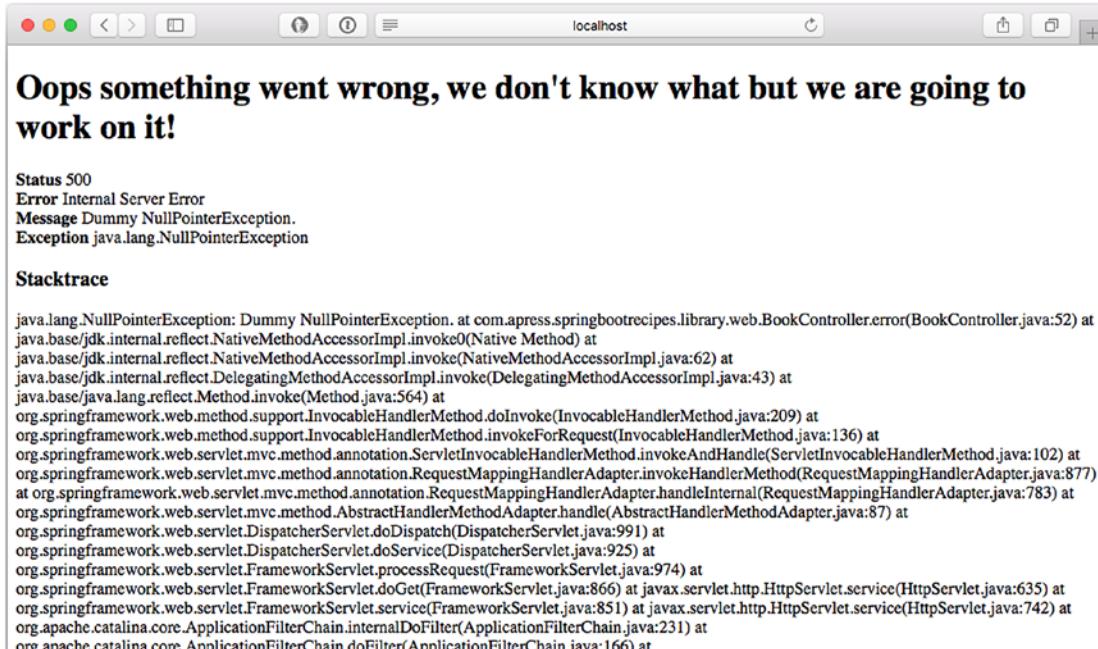


Figure 3-8. Custom error page with stacktrace

Next to providing a custom generic error page, you could also add an error page for specific HTTP status codes. This can be achieved by adding a `<http-status>.html` to the `src/main/resources/templates/error` directory. Let's add a `404.html` to be shown for unknown URLs.

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Spring Boot Recipes - Library - Resource Not Found</title>
</head>
<body>
    <h1>Oops the page couldn't be located.</h1>
</body>
</html>
```

When navigating to a URL that is unknown to the application, it will render this page, and when triggering the exception it will still show the customized error page, as shown in Figures 3-7 and 3-8.

Tip You can also add a `4xx.html` or `5xx.html` for a custom error page for all http status code in the 400 or 500 range.

Adding Attributes to the Model

By default, Spring Boot will include the attributes in the model for the error page, as listed in Table 3-4.

Table 3-4. Default Error Model Attributes

Attribute	Description
<code>timestamp</code>	The time that the errors were extracted
<code>status</code>	The status code
<code>error</code>	The error reason
<code>exception</code>	The class name of the root exception (if configured)
<code>message</code>	The exception message
<code>errors</code>	Any <code>ObjectError</code> from a <code>BindingResult</code> (when using binding and/or validation)
<code>trace</code>	The exception stack trace (if configured)
<code>path</code>	The URL path when the exception was raised

This is all done through the use of an `ErrorAttributes` component. The default used and configured is the `DefaultErrorAttributes`. You can create your own `ErrorAttributes` handler to create a custom model or extend the `DefaultErrorAttributes` to add additional attributes.

```
package com.apress.springbootrecipes.library;

import org.springframework.boot.web.servlet.error.DefaultErrorAttributes;
import org.springframework.web.context.request.WebRequest;
```

```

import java.util.Map;

public class CustomizedErrorAttributes extends DefaultErrorAttributes {

    @Override
    public Map<String, Object> getErrorAttributes(WebRequest webRequest,
        boolean includeStackTrace) {
        Map<String, Object> errorAttributes =
            super.getErrorAttributes(webRequest, includeStackTrace);
        errorAttributes.put("parameters", webRequest.getParameterMap());
        return errorAttributes;
    }
}

```

The CustomizedErrorAttributes will add the original request parameters to the model next to the default attributes. Next step is to configure this as a bean in the LibraryApplication. Spring Boot will then detect it and use it instead of configuring the default.

```

@Bean
public CustomizedErrorAttributes errorAttributes() {
    return new CustomizedErrorAttributes();
}

```

Finally, you might want to use the additional properties in your error.html.

```

<div th:if="${parameters != null}">
    <h3>Parameters</h3>
    <span th:each="parameter :${parameters}">
        <div th:text="${parameter.key} + ' : ' + ${#strings.
            arrayJoin(parameter.value, ',')}"></div>
    </span>
</div>

```

When the preceding part is included in your error.html it will print the content of the map of parameters available in the model (Figure 3-9).

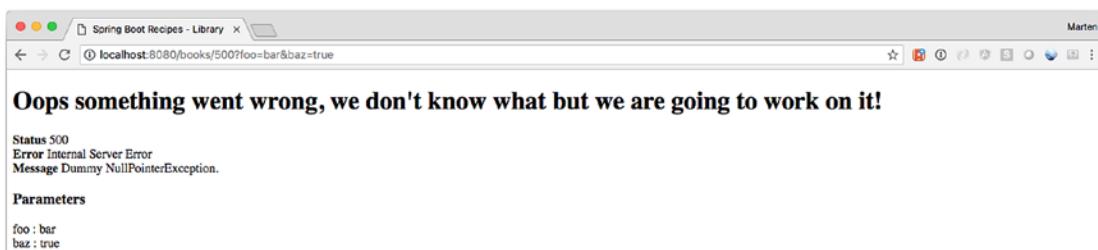


Figure 3-9. Custom error page with parameters

3-5 Internationalizing Your Application

Problem

When developing an internationalized web application, you have to display your web pages in a user's preferred locale. You don't want to create different versions of the same page for different locales.

Solution

To avoid creating different versions of a page for different locales, you should make your web page independent of the locale by externalizing locale-sensitive text messages. Spring is able to resolve text messages for you by using a message source, which has to implement the `MessageSource` interface. In your page templates you can then use either special tags or do lookups for the messages.

How It Works

Spring Boot automatically configures a `MessageSource` when it finds a `messages.properties` in `src/main/resources` (the default location). This `messages.properties` contains the default messages to be used in your application. Spring Boot will use the `Accept-Language` header from the request to determine which locale to use for the current request (see Recipe 3.6 on how to change that).

There are some properties that change the way the `MessageSource` reacts to missing translations, caching, etc. See Table 3-5 for an overview of the properties.

Table 3-5. I18N Properties

Property	Description
spring.messages.basename	Comma-separated list of basenames, default messages
spring.messages.encoding	Message bundle encoding, default UTF-8
spring.messages.always-use-message-format	Should MessageFormat be applied to all messages, default false
spring.messages.fallback-to-system-locale	Fallback to the systems locale when no resource bundle for the detected locale can be found. When disabled, will load the defaults from the default file. Default true
spring.messages.use-code-as-default-message	Use the message code as a default message when no message can be found instead of throwing a NoSuchMessageException. Default false
spring.messages.cache-duration	Cache duration, default forever

Tip It can be useful to set the `spring.messages.fallback-to-system-locale` to `false` when deploying your application to the cloud or other external hosting parties. That way you control what the default language of your application is, instead of the (out of your control) environment you are deploying on.

Add a `messages.properties` to the `src/main/resources` directory.

```
main.title=Spring Boot Recipes - Library
index.title=Library
index.books.link=List of books

books.list.title=Available Books
books.list.table.title=Title
books.list.table.author=Author
books.list.table.isbn=ISBN
```

Now change the templates to use the translations; following is the changed `index.html` file.

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title th:text="#{main.title}">Spring Boot Recipes - Library</title>
</head>
<body>
    <h1 th:text="#{index.title}">Library</h1>
    <a th:href="@{/books.html}" href="#" th:text="#{index.books.link}">List of books</a>
</body>
</html>
```

For Thymeleaf you can use a `#{...}` expression in the `th:text` attribute; this will (due to the automatic Spring integration) resolve the messages from the `MessageSource`. When restarting the application, it appears as nothing has changed in the output. However all the texts now come from the `messages.properties`.

Now let's add a `messages_nl.properties` for the Dutch translation of the website.

```
main.title=Spring Boot Recipes - Bibliotheek
index.title=Bibliotheek
index.books.link=Lijst van boeken

books.list.title=Beschikbare Boeken
books.list.table.title=Titel
books.list.table.author=Auteur
books.list.table.isbn=ISBN
```

Now when changing the accept header to Dutch, the website will translate to Dutch (Figure 3-10).

Tip Changing the language for your browser might not be that easy, for Chrome and Firefox there are plug-ins that allow you to switch the `Accept-Language` header easily.

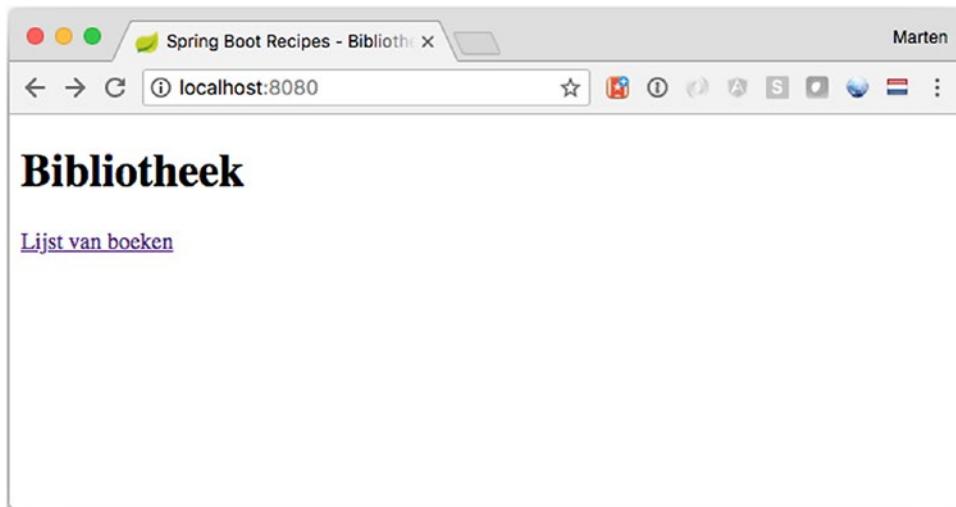


Figure 3-10. Homepage in Dutch

3-6 Resolving User Locales

Problem

In order for your web application to support internationalization, you have to identify each user's preferred locale and display contents according to this locale.

Solution

In a Spring MVC application, a user's locale is identified by a locale resolver, which has to implement the `LocaleResolver` interface. Spring MVC comes with several `LocaleResolver` implementations for you to resolve locales by different criteria. Alternatively, you may create your own custom locale resolver by implementing this interface.

Spring Boot allows you to set the `spring.mvc.locale-resolver` property. This can be set to `ACCEPT` (the default) or `FIXED`. The first will create an `AcceptHeaderLocaleResolver`; the latter, a `FixedLocaleResolver`.

You can also define a locale resolver by registering a bean of type `LocaleResolver` in the web application context. You must set the bean name of the locale resolver to `localeResolver` so it can be autodetected.

How It Works

Spring MVC ships with several default implementations of the `LocaleResolver` interface. There is a `HandlerInterceptor` provided to allow users to override the locale they want to use, the `LocaleChangeInterceptor`.

Resolving Locales by an HTTP Request Header

The default locale resolver registered by Spring Boot is the `AcceptHeaderLocaleResolver`. It resolves locales by inspecting the `Accept-Language` header of an HTTP request. This header is set by a user's web browser according to the locale setting of the underlying operating system.

Note The `AcceptHeaderLocaleResolver` cannot change a user's locale, because it is unable to modify the locale setting of the user's operating system.

```
@Bean
public LocaleResolver localeResolver () {
    return new AcceptHeaderLocaleResolver();
}
```

Resolving Locales by a Session Attribute

Another option of resolving locales is by `SessionLocaleResolver`. It resolves locales by inspecting a predefined attribute in a user's session. If the session attribute doesn't exist, this locale resolver determines the default locale from the `Accept-Language` HTTP header.

```
@Bean
public LocaleResolver localeResolver () {
    SessionLocaleResolver localeResolver = new SessionLocaleResolver();
    localeResolver.setDefaultLocale(new Locale("en"));
    return localeResolver;
}
```

You can set the `defaultLocale` property for this resolver in case the session attribute doesn't exist. Note that this locale resolver is able to change a user's locale by altering the session attribute that stores the locale.

Resolving Locales by a Cookie

You can also use `CookieLocaleResolver` to resolve locales by inspecting a cookie in a user's browser. If the cookie doesn't exist, this locale resolver determines the default locale from the `Accept-Language` HTTP header.

```
@Bean
public LocaleResolver localeResolver() {
    return new CookieLocaleResolver();
}
```

The cookie used by this locale resolver can be customized by setting the `cookieName` and `cookieMaxAge` properties. The `cookieMaxAge` property indicates how many seconds this cookie should be persisted. The value `-1` indicates that this cookie will be invalid after the browser is closed.

```
@Bean
public LocaleResolver localeResolver() {
    CookieLocaleResolver cookieLocaleResolver = new CookieLocaleResolver();
    cookieLocaleResolver.setCookieName("language");
    cookieLocaleResolver.setCookieMaxAge(3600);
    cookieLocaleResolver.setDefaultLocale(new Locale("en"));
    return cookieLocaleResolver;
}
```

You can also set the `defaultLocale` property for this resolver in case the cookie doesn't exist in a user's browser. This locale resolver is able to change a user's locale by altering the cookie that stores the locale.

Using a Fixed Locale

The `FixedLocaleResolver` always returns the same fixed locale. By default it returns the JVM default locale but it can be configured to return a different one by setting the `defaultLocale` property.

```

@Bean
public LocaleResolver localeResolver() {
    FixedLocaleResolver cookieLocaleResolver = new FixedLocaleResolver();
    cookieLocaleResolver.setDefaultLocale(new Locale("en"));
    return cookieLocaleResolver;
}

```

Note The `FixedLocaleResolver` cannot change a user's locale because, as the name implies, it is fixed.

Changing a User's Locale

In addition to changing a user's locale by calling `LocaleResolver.setLocale()` explicitly, you can also apply `LocaleChangeInterceptor` to your handler mappings. This interceptor detects if a special parameter is present in the current HTTP request. The parameter name can be customized with the `paramName` property of this interceptor (default is `locale`). If such a parameter is present in the current request, this interceptor changes the user's locale according to the parameter value.

To be able to change the locale, a `LocaleResolver` that allows change has to be used.

```

@Bean
public LocaleResolver localeResolver() {
    return new CookieLocaleResolver();
}

```

To change the locale, add the `LocaleChangeInterceptor` as a bean and register it as an interceptor; for the latter, use the `addInterceptors` method from the `WebMvcConfigurer`.

Note Instead of adding it to the `@SpringBootApplication` you could also create a specialized `@Configuration` annotated class to register the interceptors. Be aware of **not** adding `@EnableWebMvc` to that class, as that will disable the auto configuration from Spring Boot!

```

@SpringBootApplication
public class LibraryApplication implements WebMvcConfigurer {

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(localeChangeInterceptor());
    }

    @Bean
    public LocaleChangeInterceptor localeChangeInterceptor() {
        return new LocaleChangeInterceptor();
    }

    @Bean
    public LocaleResolver localeResolver() {
        return new CookieLocaleResolver();
    }
}

```

Now add the following snippet to the index.html:

```

<h3>Language</h3>
<div>
    <a href="?locale=nl" th:text="#{main.language.nl}">NL</a> |
    <a href="?locale=en" th:text="#{main.language.en}">EN</a>
</div>

```

Add the keys to the messages.properties file.

```

main.language.nl=Dutch
main.language.en=English

```

Now when selecting one of the languages, the page will re-render and be shown in the selected language; and when you continue browsing, the remainder of the pages will also be shown in the selected language.

3-7 Selecting and Configuring the Embedded Server Problem

You want to use Jetty as an embedded container instead of the default Tomcat container.

Solution

Exclude the Tomcat runtime and include the Jetty runtime. Spring Boot will automatically detect if Tomcat, Jetty, or Undertow is on the classpath and configure the container accordingly.

How It Works

Spring Boot has out-of-the-box support for Tomcat, Jetty, and Undertow as embedded servlet containers. By default, Spring Boot uses Tomcat as the container (expressed through the `spring-boot-starter-tomcat` dependency in the `spring-boot-starter-web` artifact). The container can be configured using properties for which some apply to all containers and others to a specific container. The global properties are prefixed with `server.` or `server.servlet` whereas the container ones start with `server.<container>` (where `container` is either `tomcat`, `jetty`, or `undertow`).

General Configuration Properties

There are several general server properties available, as seen in Table 3-6.

Table 3-6. General Server Properties

Property	Description
<code>server.port</code>	The HTTP server port, default 8080
<code>server.address</code>	The IP Address to bind to, default 0.0.0.0 (i.e., all adapters)
<code>server.use-forward-headers</code>	Should X-Forwarded-* headers be applied to the current request, default not set and uses the default from the selected servlet container

(continued)

Table 3-6. (continued)

Property	Description
server.server-header	Name of the header to be sent the Server name, default empty
server.max-http-header-size	Maximum size of a HTTP header, default 0 (unlimited)
server.connection-timeout	Timeout for HTTP connectors to wait for the next request before closing. Default is empty leaving it to the container; a value of -1 means infinite and never timeout.
server.http2.enabled	Enable Http2 support if the current container supports it. Default false
server.compression.enabled	Should HTTP compression be enabled, default false
server.compression.mime-types	Comma seperated list of MIME types that compression applies to
server.compression.excluded-user-agents	Comma separated list of user agents for which compression should be disabled
server.compression.min-response-size	Minimum size of the request for compression to be applied, default 2048
server.servlet.context-path	The main context path of the application, default launched as the root application
server.servlet.path	The path of the main DispatcherServlet, default /
server.servlet.application-display-name	Name used as display name in the container, default application
server.servlet.context-parameters	Servlet Container Context/init parameters

As the embedded containers all adhere to the Servlet specification, there is also support for JSP pages and that support is enabled by default. Spring Boot makes it easy to change the JSP provider or even disable the support in full. See Table 3-7 for the exposed properties.

Table 3-7. JSP Related Server Properties

Property	Description
server.servlet.jsp.registered	Should the JSP servlet be registered, default true
server.servlet.jsp.class-name	The JSP servlet classname, default org.apache.jasper.servlet.JspServlet as both Tomcat and Jetty use Jasper as the JSP implementation
server.servlet.jsp.init-parameters	Context parameters for the JSP servlet

Note The use of JSP with a Spring Boot application is discouraged and limited.¹¹

When using Spring MVC, you might want to use the HTTP Session to store attributes (generally with Spring Security to store CSRF tokens, etc.). The general servlet configuration also allows you to configure the http session and the way it will be stored (cookie, URL, etc.). See Table 3-8 for the properties.

Table 3-8. HTTP Session Related Server Properties

server.servlet.session.timeout	Session timeout, default 30 minutes
server.servlet.session.tracking-modes	Session tracking modes one or more of cookie, url and ssl. Default empty leaving it to the container
server.servlet.session.persistent	Should session data be persisted between restarts, default false
server.servlet.session.cookie.name	Name of the cookie to store the session identifier. Default empty leaving it to the container default
server.servlet.session.cookie.domain	Domain value to use for the session cookie. Default empty leaving it to the container default

(continued)

¹¹<https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-developing-web-applications.html#boot-features-jsp-limitations>

Table 3-8. (*continued*)

<code>server.servlet.session.cookie.path</code>	Path value to use for the session cookie. Default empty leaving it to the container default
<code>server.servlet.session.cookie.comment</code>	Comment to use for the session cookie. Default empty leaving it to the container default
<code>server.servlet.session.cookie.http-only</code>	Should the session cookie be http-only accessible, default empty leaving it to the container default
<code>server.servlet.session.cookie.secure</code>	Should the cookie be sent through SSL only, default empty leaving it to the container default
<code>server.servlet.session.cookie.max-age</code>	The lifetime of the session cookie. Default empty leaving it to the container default
<code>server.servlet.session.session-store-directory.directory</code>	Name of the directory to use for persistent cookies. Has to be an existing directory

Finally, Spring Boot makes it very easy to configure SSL by exposing a few properties, see Table 3-9 and Recipe 3.8 on how to configure SSL.

Table 3-9. *SSL Related Server Properties*

Property	Description
<code>server.ssl.enabled</code>	Should SSL be enabled, default true
<code>server.ssl.ciphers</code>	Supported SSL ciphers, default empty
<code>server.ssl.client-auth</code>	Should SSL client authentication be wanted (WANT) or needed (NEED). Default empty
<code>server.ssl.protocol</code>	SSL protocol to use, default TLS
<code>server.ssl.enabled-protocols</code>	Which SSL protocols are enabled, default empty
<code>server.ssl.key-alias</code>	The alias to identify the key in the keystore, default empty
<code>server.ssl.key-password</code>	The password to access the key in the keystore, default empty

(*continued*)

Table 3-9. (continued)

Property	Description
server.ssl.key-store	Location of the keystore, typically a JKS file, default empty
server.ssl.key-store-password	Password to access the keystore, default empty
server.ssl.key-store-type	Type of the keystore, default empty
server.ssl.key-store-provider	Provider of the keystore, default empty
server.ssl.trust-store	Location of the truststore
server.ssl.trust-store-password	Password to access the truststore, default empty
server.ssl.trust-store-type	Type of the truststore, default empty
server.ssl.trust-store-provider	Provider of the truststore, default empty

Note All the properties mentioned in the aforementioned tables apply **only** when using an embedded container to run your application. When deploying to an external container (i.e., deploying a WAR file), the settings do not apply!

As we are using Thymeleaf, we could disable the registration of the JSP servlet; and let's change the port and compression as well. Place the following in the application.properties:

```
server.port=8081
server.compression.enabled=true
server.servlet.jsp.registered=false
```

Now when (re)starting, your application pages (when large enough) will be compressed and the server will run on port 8081 instead of 8080.

Note There is a difference between `server.servlet.context-path` and `server.servlet.path`. When looking at a URL `http://localhost:8080/books` it consists of several parts. The first is the protocol (generally http or https); then there is the address and port of the server you want to access, followed by the context-path (default / deployed at the root), which in turn is followed by the servlet-path. The `server.context-path` is the main URL to your application; for instance if we set the `server.context-path=/library` the whole application is available on the `/library` URL. (The `DispatcherServlet` still listens to / within the context-path). Now if we set the `server.path=/dispatch` we need to use `/library/dispatch/books` to access the books.

Next, if we added a second `DispatcherServlet`, which we configure to have a path of `/services`, that would be accessible through `/library/services`. Both `DispatcherServlets` would be active within the main context-path of `/library`.

Changing the Runtime Container

When including the `spring-boot-starter-web` dependency, it will automatically include a dependency to the Tomcat container as it itself has a dependency on the `spring-boot-starter-tomcat` artifact. To enable a different servlet container, the `spring-boot-starter-tomcat` needs to be excluded and one of `spring-boot-starter-jetty` or `spring-boot-starter-undertow` needs to be included.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
  <exclusions>
    <exclusion>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-tomcat</artifactId>
    </exclusion>
  </exclusions>
```

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

In Maven you can use an `<exclusion>` element inside your `<dependency>` to exclude a dependency.

Now when the application is started it will start with Jetty instead of using Tomcat (Figure 3-11).

```
2018-09-01 09:43:46.864 INFO 44500 — [ 849348713278767.8081/] AVAILABLE}
2018-09-01 09:43:46.865 INFO 44500 — [ main] o.e.je.jetty.server.Server : Started @200ms
2018-09-01 09:43:46.905 INFO 44500 — [ main] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
2018-09-01 09:43:47.223 INFO 44500 — [ main] o.s.b.a.w.s.WelcomePageHandlerMapping : Adding welcome page template: index
2018-09-01 09:43:47.404 INFO 44500 — [ main] o.e.j.s.h.ContextHandlerApplication : Initializing Spring DispatcherServlet 'dispatcherServlet'
2018-09-01 09:43:47.404 INFO 44500 — [ main] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
2018-09-01 09:43:47.405 INFO 44500 — [ main] o.s.web.servlet.DispatcherServlet : Completed initialization in 5 ms
2018-09-01 09:43:47.444 INFO 44500 — [ main] o.s.web.servlet.AbstractConnector : Started ServerConnector@70fab835{HTTP/1.1}/{0.0.0.0:8081}
2018-09-01 09:43:47.446 INFO 44500 — [ main] o.s.b.web.embedded.jetty.JettyWebServer : Jetty started on port(s) 8081 (http/1.1) with context path ''
2018-09-01 09:43:47.451 INFO 44500 — [ main] c.a.s.library.LibraryApplication : Started LibraryApplication in 2.305 seconds (JVM running for 2.785)
```

Figure 3-11. Bootstrap logging with Jetty container

3-8 Configuring SSL for the Servlet Container Problem

You want your application to be accessible through HTTPS next (or instead of) HTTP.

Solution

Get a certificate, place it in a keystore, and use the `server.ssl` namespace to configure the keystore. Spring Boot will then automatically configure the server to be accessible through HTTPS only.

How It Works

Using the `server.ssl.key-store` (and related properties) you can configure the embedded container to only accept HTTPS connection. Before you can configure SSL you will need to have a certificate to secure your application with. Generally you will want to get a certificate from a certification authority like VeriSign or Let's Encrypt. However, for development purposes you can use a self-signed certificate (see the section on [Creating a Self-Signed Certificate](#)).

Creating a Self-Signed Certificate

Java comes packaged with a tool called the keytool, which can be used to generate certificates among other things.

```
keytool -genkey -keyalg RSA -alias sb2-recipes -keystore sb2-recipes.pfx  
-storepass password -validity 3600 -keysize 4096 -storetype pkcs12
```

The preceding command will tell keytool to generate a key using the RSA algorithm and place it in the keystore named `sb2-recipes.pfx` with the alias `sb2-recipes`, and it will be valid for 3,600 days. When running the command, it will ask a few questions; answer them accordingly (or leave empty). After that, there will be a file called `sb2-recipes.pfx` containing the certificate and protected with a password.

Place this file in the `src/main/resources` folder so that it is packaged as part of your application and Spring Boot can easily access it.

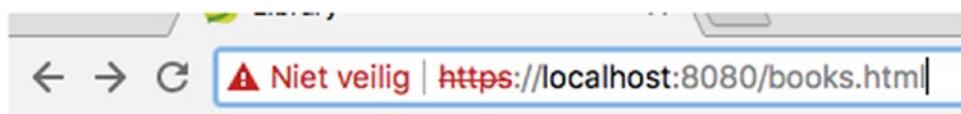
Warning Using a self-signed certificate will produce a warning in the browser that the website isn't safe and protected, because the certificate isn't issued by a trusted authority (see also Figure 3-12).

Configure Spring Boot to Use the Keystore

Spring Boot will need to know about the keystore to be able to configure the embedded container. For this, use the `server.ssl.key-store` property. You will also need to specify the type of keystore (`pkcs12`) and the password.

```
server.ssl.key-store=classpath:sb2-recipes.pfx  
server.ssl.key-store-type=pkcs12  
server.ssl.key-store-password=password  
server.ssl.key-password=password  
server.ssl.key-alias=sb2-recipes
```

Now when opening the `http://localhost:8080/books.html` page, it will be served through https (although with a warning). See Figure 3-12.



Available Books

Title	Author	ISBN
The Hobbit	[J.R.R. Tolkien]	9780618260300
1984	[George Orwell]	9780451524935
To Kill a Mockingbird	[Harper Lee]	9780061120084

Figure 3-12. HTTPS access

Support Both HTTP and HTTPS

Spring Boot by default only starts one connector: either HTTP or HTTPS but not both. If you want to support both HTTP and HTTPS, you will manually have to add an additional connector. It is easiest to create the HTTP connector yourself and let Spring Boot set up the SSL part.

First let's configure Spring Boot to start the server on port 8443.

```
server.port=8443
```

To add an additional connector to the embedded Tomcat, you will need to add the `TomcatServletWebServerFactory` as a bean to our context. Normally Spring Boot would detect the container and select the `WebServerFactory` to use; however, as a customization needs to be done we need to add it manually. This bean can be added to a `@Configuration` annotated class or to the `LibraryApplication` class.

```
@Bean
public TomcatServletWebServerFactory tomcatServletWebServerFactory() {
    var factory = new TomcatServletWebServerFactory();
    factory.addAdditionalTomcatConnectors(httpConnector());
    return factory;
}
```

```
private Connector httpConnector() {
    var connector = new Connector(TomcatServletWebServerFactory.DEFAULT_PROTOCOL);
    connector.setScheme("http");
    connector.setPort(8080);
    connector.setSecure(false);
    return connector;
}
```

This will add an additional connector on port 8080. The application will now be usable from both port 8080 and 8443. Using Spring Security, you could now force access to parts of your application over HTTPS instead of HTTP.

Tip If you don't want to explicitly configure the TomcatServletWebServerFactory you could also use a BeanPostProcessor to register the additional Tomcat Connector with the TomcatServletWebServerFactory. That way you could implement this for different embedded containers instead of being tied to a single container.

```
@Bean
public BeanPostProcessor addHttpConnectorProcessor() {
    return new BeanPostProcessor() {
        @Override
        public Object postProcessBeforeInitialization(Object bean, String
beanName)
        throws BeansException {
            if (bean instanceof TomcatServletWebServerFactory) {
                var factory = (TomcatServletWebServerFactory) bean;
                factory.addAdditionalTomcatConnectors(httpConnector());
            }
            return bean;
        }
    };
}
```

Redirect HTTP to HTTPS

Instead of supporting both HTTP and HTTPS, another option is to support only HTTPS and redirect the traffic from HTTP to HTTPS. The configuration is similar to that as when supporting both HTTP and HTTPS. However, you now configure the connector to redirect all traffic from 8080 to 8443.

```
@Bean
public TomcatServletWebServerFactory tomcatServletWebServerFactory() {
    var factory = new TomcatServletWebServerFactory();
    factory.addAdditionalTomcatConnectors(httpConnector());
    factory.addContextCustomizers(securityCustomizer());
    return factory;
}

private Connector httpConnector() {
    var connector = new Connector(TomcatServletWebServerFactory.DEFAULT_PROTOCOL);
    connector.setScheme("http");
    connector.setPort(8080);
    connector.setSecure(false);
    connector.setRedirectPort(8443);
    return connector;
}

private TomcatContextCustomizer securityCustomizer() {
    return context -> {
        var securityConstraint = new SecurityConstraint();
        securityConstraint.setUserConstraint("CONFIDENTIAL");
        var collection = new SecurityCollection();
        collection.addPattern("/*");
        securityConstraint.addCollection(collection);
        context.addConstraint(securityConstraint);
    };
}
```

CHAPTER 3 SPRING MVC

The `httpConnector` now has a `redirectPort` set so that it knows which port to use. Finally, you need to secure all URLs with a `SecurityConstraint`. With Spring Boot you can use a specialized `TomcatContextCustomizer` to post-process the Context from Tomcat before it is started. The constraint makes everything (due to the use of `/*` as pattern) confidential (one of `NONE`, `INTEGRAL`, or `CONFIDENTIAL` is allowed) and the result is that everything will be redirected to https.

CHAPTER 4

Spring MVC - Async

When the Servlet API was released, the majority of the implementing containers used a thread per request. This meant a thread was blocked until the request processing finished and the response was sent to the client.

In those early days there weren't as many devices connected to the Internet as nowadays. Due to the increased number of devices, the number of HTTP requests handled has grown. Due to this increase for a lot of web applications, keeping a thread blocked isn't workable anymore.

As of the Servlet 3 specification, it is possible to handle a HTTP request asynchronously. This is done by releasing the thread that handled the HTTP request. The new thread will run in the background, and as soon as the result is available it's written to the client. This all can be done in a nonblocking way on a Servlet 3.1-compliant servlet container. All used resources then also would have to be nonblocking.

In the last couple of years there has also been an uprising in Reactive programming. As of Spring 5 it is possible to write Reactive web applications. To be reactive, spring utilizes Project Reactor as an implementation of the Reactive Streams API. It goes beyond the scope of this book to do a full dive into Reactive programming. Reactive programming is, in short, a way of doing nonblocking functional programming.

When working with web applications there would be a request, HTML would render on the server, and that got sent back to the client. In the last couple of years, rendering of HTML moved to the client. Communication is done by returning JSON, XML, or another representation to the client.

This was still a request and response cycle, although driven through an async call by the client through the XMLHttpRequest. There are also other ways of communicating between the client and server. One could use Server-Sent-Events to have one-way communication. For full duplex communication, one could use Web Sockets.

Add the `spring-boot-starter-web` as a dependency when building a regular web application. Add `spring-boot-starter-webflux` when building a reactive application.

4-1 Asynchronous Request Handling with Controllers and TaskExecutor

Problem

To reduce the throughput on the servlet container, you want to asynchronously handle the request.

Solution

When a request comes in it is handled synchronously, which blocks the HTTP request handling thread. The response stays open and is available to be written to. This is useful when, for instance, a call takes some time to finish and instead of blocking threads you can have this processed in the background and return a value to the user when finished.

How It Works

Spring MVC supports a number of return types from methods; the return types in Table 4-1 are processed in an asynchronous way.

Table 4-1. Asynchronously Return Types

Type	Description
DeferredResult<V>	Async result produced from another thread
ListenableFuture<V>	Async result produced from another thread, an equivalent alternative for DeferredResult
CompletionStage<V> / CompletableFuture<V>	Async result produced from another thread, an equivalent alternative for DeferredResult
Callable<V>	An async computation with the result produced after the computation finishes
ResponseBodyEmitter	Can be used to write multiple objects to the response asynchronously
SseEmitter	Can be used to write Server-Sent Event asynchronously
StreamingResponseBody	Can be used to write to the OutputStream asynchronously.

Writing an Asynchronous Controller

Writing a controller and having it handle the request asynchronously is as simple as changing the return type of the controllers handler method (see Table 4-1). Let's imagine that the call to `HelloWorldController.hello` takes quite some time but we don't want to block the server for that.

Use a Callable

```
package com.apress.springbootrecipes.library;

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

import java.util.concurrent.Callable;
import java.util.concurrent.ThreadLocalRandom;

@RestController
public class HelloWorldController {

    @GetMapping
    public Callable<String> hello() {
        return () -> {
            Thread.sleep(ThreadLocalRandom.current().nextInt(5000));
            return "Hello World, from Spring Boot 2!";
        };
    }
}
```

The `hello` method now returns a `Callable<String>` instead of returning a `String` directly. Inside the newly constructed `Callable<String>` there is a random wait to simulate a delay before the message is returned to the client.

Now when making a reservation, you will see something in the logs similar to Figure 4-1.

```
2018-09-09 10:06:14.957 DEBUG 82160 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : GET "/hello", parameters={}
2018-09-09 10:06:14.972 DEBUG 82160 --- [nio-8080-exec-1] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped to public java.util.concurrent.Callable<java.lang.String>
2018-09-09 10:06:14.996 DEBUG 82160 --- [nio-8080-exec-1] o.s.w.c.request.async.WebAsyncManager : Started async request
2018-09-09 10:06:14.998 DEBUG 82160 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Exiting but response remains open for further handling
2018-09-09 10:06:15.708 DEBUG 82160 --- [nio-8080-exec-1] o.s.w.c.request.async.WebAsyncManager : Async result set, dispatched to /hello
2018-09-09 10:06:15.716 DEBUG 82160 --- [nio-8080-exec-2] o.s.web.servlet.DispatcherServlet : "ASYNC" dispatch for GET "/hello", parameters={}
2018-09-09 10:06:15.716 DEBUG 82160 --- [nio-8080-exec-2] s.w.s.m.m.a.RequestMappingHandlerAdapter : Resum with async result ["Hello World, from Spring Boot 2!"]
2018-09-09 10:06:15.730 DEBUG 82160 --- [nio-8080-exec-2] n.m.a.RequestResponseBodyMethodProcessor : Using 'text/plain', given ['/*'] and supported [text/plain]
2018-09-09 10:06:15.731 DEBUG 82160 --- [nio-8080-exec-2] n.m.a.RequestResponseBodyMethodProcessor : Writing ["Hello World, from Spring Boot 2!"]
2018-09-09 10:06:15.740 DEBUG 82160 --- [nio-8080-exec-2] o.s.web.servlet.DispatcherServlet : Exiting from "ASYNC" dispatch, status 200
```

Figure 4-1. Logging Output

You notice that request handling is started on a certain thread (here `nio-8080-exec-1`). Another thread is doing the processing and returning the result (here, `task-1`). Finally, the request is dispatched to the `DispatcherServlet` again to handle the result on yet another thread (here `nio-8080-exec-2`).

Use a CompletableFuture

Change the signature of the method to return `CompletableFuture<String>` and use the `TaskExecutor` to asynchronously execute the code.

```
@RestController
public class HelloWorldController {

    private final TaskExecutor taskExecutor;

    public HelloWorldController(TaskExecutor taskExecutor) {
        this.taskExecutor = taskExecutor;
    }

    @GetMapping
    public CompletableFuture<String> hello() {
        return CompletableFuture.supplyAsync(() -> {
            randomDelay();
            return "Hello World, from Spring Boot 2!";
        }, taskExecutor);
    }

    private void randomDelay() {
        try {
            Thread.sleep(ThreadLocalRandom.current().nextInt(5000));
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}
```

Calling the `supplyAsync` (or when using `void`, use `runAsync`) a task is submitted. It returns a `CompletableFuture`. Here you use the `supplyAsync` method that takes both a `Supplier` and an `Executor`. This way you can reuse the `TaskExecutor` for async processing. If you use the `supplyAsync` method, which only takes a `Supplier`, it will be executed using the default fork/join pool available on the JVM.

When returning a `CompletableFuture`, you can take advantage of all the features of it like composing and chaining multiple `CompletableFuture` instances.

Testing Async Controllers

Just as with regular controllers, the Spring MVC Test framework can be used to test async controllers.

Create a test class and annotate it with `@WebMvcTest(HelloworldController.class)` and `@RunWith(SpringRunner.class)`. The `@WebMvcTest` will bootstrap a minimal Spring Boot application containing the things needed to test the controller. It automatically configures Spring MockMvc, which is autowired into the test.

```
@RunWith(SpringRunner.class)
@WebMvcTest(HelloworldController.class)
public class HelloworldControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @Test
    public void testHelloWorldController() throws Exception {
        MvcResult mvcResult = mockMvc.perform(get("/"))
            .andExpect(request().asyncStarted())
            .andDo(MockMvcResultHandlers.log())
            .andReturn();

        mockMvc.perform(asyncDispatch(mvcResult))
            .andExpect(status().isOk())
            .andExpect(content().contentTypeCompatibleWith(TEXT_PLAIN))
            .andExpect(content().string("Hello World, from Spring Boot 2!"));
    }
}
```

The main difference between an async web test and a regular web test is the fact that the async dispatching needs to be initiated. First the initial request is performed and validated for async to be started. For debugging purposes you could log the result with `MockMvcResultHandlers.log()`. Next, the `asyncDispatch` is applied. Finally we can assert the expected response.

Configuring Async Processing

Depending on your needs, you probably also want to configure an explicit `TaskExecutor` for the asynchronous request handling instead of using the default `TaskExecutor` provided by Spring Boot.

To configure async processing, you override the `configureAsyncSupport` method of `WebMvcConfigurer`. Overriding this method gives you access to the `AsyncSupportConfigurer`. This allows you to set the `AsyncTaskExecutor` to use (among others).

```
@SpringBootApplication
public class HelloWorldApplication implements WebMvcConfigurer {

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

    @Override
    public void configureAsyncSupport(AsyncSupportConfigurer configurer) {
        configurer.setTaskExecutor(mvcTaskExecutor());
    }

    @Bean
    public ThreadPoolTaskExecutor mvcTaskExecutor() {
        ThreadPoolTaskExecutor taskExecutor = new ThreadPoolTaskExecutor();
        taskExecutor.setThreadNamePrefix("mvc-task-");
        return taskExecutor;
    }
}
```

4-2 Response Writers

Problem

You have a service, or multiple calls, and want to collect the results and send the response to the client.

Solution

Use a `ResponseBodyEmitter` (or its subclass `SseEmitter`) to collect and send the response to the client.

How It Works

Send Multiple Results in a Response

Spring MVC has a class named `ResponseBodyEmitter`, which is useful if instead of a single result you want to return multiple objects to the client. When sending an object, it is converted to a result using an `HttpMessageConverter`. To use the `ResponseBodyEmitter` you have to construct it and return it from the request handling method.

Create an `OrderController` with an `orders` method that returns a `ResponseBodyEmitter` and send the results one by one to the client.

First create the `Order` and `OrderService`.

```
public class Order {

    private String id;
    private BigDecimal amount;

    public Order() {
    }

    public Order(String id, BigDecimal amount) {
        this.id=id;
        this.amount = amount;
    }
}
```

```

public String getId() {
    return id;
}

public void setId(String id) {
    this.id = id;
}

public BigDecimal getAmount() {
    return amount;
}

public void setAmount(BigDecimal amount) {
    this.amount = amount;
}

@Override
public String toString() {
    return String.format("Order [id=%s, amount=%4.2f]", id, amount);
}
}

```

Then add a simple OrderService to hold the orders.

```

@Service
public class OrderService {

    private final List<Order> orders = new ArrayList<>();

    @PostConstruct
    public void setup() {
        createOrders();
    }

    public Iterable<Order> findAll() {
        return List.copyOf(orders);
    }
}

```

```

private Iterable<Order> createOrders() {
    for (int i = 0; i < 25; i++) {
        this.orders.add(createOrder());
    }
    return orders;
}

private Order createOrder() {
    String id = UUID.randomUUID().toString();
    double amount = ThreadLocalRandom.current().nextDouble(1000.00d);
    return new Order(id, BigDecimal.valueOf(amount));
}
}

```

Now create the OrderController.

```

@RestController
public class OrderController {

    private final OrderService orderService;

    public OrderController(OrderService orderService) {
        this.orderService = orderService;
    }

    @GetMapping("/orders")
    public ResponseBodyEmitter orders() {
        var emitter = new ResponseBodyEmitter();
        var executor = Executors.newSingleThreadExecutor();
        executor.execute(() -> {
            var orders = orderService.findAll();
            try {
                for (var order : orders) {
                    randomDelay();
                    emitter.send(order);
                }
            }
        });
    }
}

```

```
        emitter.complete();
    } catch (IOException e) {
        emitter.completeWithError(e);
    }
});
executor.shutdown();
return emitter;
}

private void randomDelay() {
    try {
        Thread.sleep(ThreadLocalRandom.current().nextInt(150));
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
}
}
```

First a `ResponseBodyEmitter` is created, and in the end returned from this method. Next a task is executed that will query the reservations using the `OrderService.findAll` method. All the results from that call are returned one by one using the `send` method of the `ResponseBodyEmitter`. When all the objects have been sent, the `complete()` method needs to be called so that the thread responsible for sending the response can complete the request and be freed up for the next response to handle. When an exception occurs and you want to inform the user of this, you call the `completeWithError`; the exception will pass through the normal exception handling of Spring MVC and after that the response is completed.

When using a tool like httpie or curl, calling the URL `http://localhost:8080/orders` will yield a result similar to that depicted in Figure 4-2.

```
marten -- marten@imac-van-marten -- ~ -- zsh -- 122×32
[~] http :8080/orders -v
GET /orders HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
Connection: keep-alive
Host: localhost:8080
User-Agent: HTTPie/0.9.9

HTTP/1.1 200
Date: Sun, 09 Sep 2018 08:34:38 GMT
Transfer-Encoding: chunked

{"id":"a3e2c659-6d61-4f72-81a2-9386767901c7","amount":964.3303264760648}{"id":"68e353b9-69b0-44cc-a0a5-c9d811a61f5e","amount":630.5819945932226}{"id":"7cd568aa-4168-40cb-996c-051e5fa82457","amount":247.3223213836674}{"id":"964a6124-f993-43eb-ba17-908bf5097aa","amount":808.549237068922}{"id":"5ecc62bc-9f3c-473b-9966-d5c9f37555fd","amount":661.3744699952401}{"id":"f676d1bc-dde0-4f8c-83b5-e3101642c123","amount":857.9818651414307}{"id":"2c4f7232-83f5-41b3-9633-4f71a48e3364","amount":891.7136746089751}{"id":"d2000c31-8ddc-46d1-aea3-9d0fac9f14c0","amount":332.49993916867015}{"id":"b683f8ee-011b-4390-8973-80a17509d7a,"amount":800.6509712001965}{"id":"3cd778f0-9e25-4d2a-b29e-92d239a9189d","amount":801.3667308438916}{"id":"ab08e7da-9d76-42d9-a0f0-0455e0518fb1,"amount":554.321571430265}{"id":"7769e9a4-f951-447c-961c-d715c8a18ab8","amount":115.50183059747442}{"id":"5b1fc94f-853b-4f32-b01c-1f4af7ce41dd","amount":686.6941002940033}{"id":"05d36d52-786a-4a5e-b02a-915958808937e","amount":748.0382636058531}{"id":"6564f567-d9ad-4486-b5dd-b77ce1e1b06c","amount":879.4838692645402}{"id":"be6c103d-f09a-4afe-9f5f-992d522b42ba,"amount":447.45108559657245} {"id":"46a39e54-bb29-43dc-9417-c17fc7462574","amount":419.5476492449695} {"id":"51a7c85c-b35a-4269-9b52-1d592204dd30","amount":421.8349910257171} {"id":"35564f73-b7e4-46ca-9438-fdc9637d63d4","amount":975.0530454514776} {"id":"6f0e0bfa-1c60-4a9c-8299-1d592204dd30","amount":338.3559152812614} {"id":"5542e1a6-d500-444d-b889-ad0cd98d17a7,"amount":402.66337491090167} {"id":"47087519-3e98-4f10-8719-61aa581f06a6,"amount":702.9394273793404} {"id":"11b966b15-d4eb-4f56-b14b-1b3f9ff77d61,"amount":176.0339216118647} {"id":"697b74c3-8e35-4514-beb9-3e35d4a7b935,"amount":297.7516547391046} {"id":"c058b00d-b09f-442f-8f7b-3c8469f49720,"amount":588.2887029367946}
```

Figure 4-2. Output of ResponseBodyEmitter

Finally, let's write a test for this controller. Annotate a class with @RunWith(SpringRunner.class) and @WebMvcTest(OrderController.class) to get a web slice test. The OrderController needs an OrderService, which is mocked by using @MockBean on the OrderService field.

```
@RunWith(SpringRunner.class)
@WebMvcTest(OrderController.class)
public class OrderControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private OrderService orderService;

    @Test
    public void foo() throws Exception {
        when(orderService.findAll())
            .thenReturn(List.of(new Order("1234", BigDecimal.TEN)));
    }
}
```

```

MvcResult mvcResult =
    mockMvc.perform(get("/orders"))
        .andExpect(request().asyncStarted())
        .andDo(MockMvcResultHandlers.log())
        .andReturn();

mockMvc.perform(asyncDispatch(mvcResult))
    .andDo(MockMvcResultHandlers.log())
    .andExpect(status().isOk())
    .andExpect(content().json("{\"id\":\"1234\", \"amount\":10}"));
}

}
}

```

The test method first registers behavior on the mocked OrderService to return a single instance of an Order. Next we use MockMvc to perform a get on the /orders endpoint. As this is an asynchronous controller, the request should start asynchronous processing. Next we mimic the async dispatching and write assertions for the actual response. The result should be a single JSON element containing the id and amount.

Sending Multiple Results as Events

The SseEmitter can deliver events from the server to the client. Server-Sent-Events are messages from the server to the client. They have a Content-Type header of text/event-stream. They are quite lightweight, with only four fields (Table 4-2).

Table 4-2. Allowed Fields For Server-Sent-Events

Field	Description
id	The ID of the event
event	the type of event
data	The event data
retry	Reconnection time for the event stream

To send events from a request handling method, you need to create an instance of `SseEmitter` and return it from the request handling method. Then use the `send` method to send individual elements to the client.

```
@GetMapping("/orders")
public SseEmitter orders() {
    SseEmitter emitter = new SseEmitter();
    ExecutorService executor = Executors.newSingleThreadExecutor();
    executor.execute(() -> {
        var orders = orderService.findAll();
        try {
            for (var order : orders) {
                randomDelay();
                emitter.send(order);
            }
            emitter.complete();
        } catch (IOException e) {
            emitter.completeWithError(e);
        }
    });
    executor.shutdown();
    return emitter;
}
```

Note There is a delay in sending each item to the client; just so you can see the different events coming in you wouldn't do this in real code.

Now when the URL `http://localhost:8080/orders` is called, you see events coming in one by one (Figure 4-3).

```
[> http :8080/orders
HTTP/1.1 200
Content-Type: text/event-stream;charset=UTF-8
Date: Wed, 11 Jul 2018 10:06:56 GMT
Transfer-Encoding: chunked

data: {"id": "941a263f-3f48-49cb-9801-ca931623a355", "amount": 730.4383307235909}
data: {"id": "0b0aefda-afdf-47da-853d-25ce67977597", "amount": 154.7487508307738}
data: {"id": "03504d98-ce57-4950-9627-8dd582b0f43f", "amount": 689.9288545237567}
data: {"id": "bc2da6aa-d6dc-4201-b880-35f114ba6f51", "amount": 581.8911724684582}
data: {"id": "a25f2d92-ff06-4b00-b08b-f9b85fa4355c", "amount": 779.151699482505}
data: {"id": "a70c3311-cc88-4a82-93ab-06e25a490cbb", "amount": 198.71754307506419}
data: {"id": "334bc6c5-8cc5-40f7-bc3b-1035b99dcc68", "amount": 371.8884523273436}
data: {"id": "416ddcad-4149-42ec-bf79-50a8f10c81a9", "amount": 955.9374666581973}
data: {"id": "e64d821f-6a6f-4c81-9698-99e135cc8fd4", "amount": 549.3973604937846}
data: {"id": "a220b6a9-53f0-4a3e-b34c-938966007727", "amount": 179.1484242467024}
data: {"id": "598448a5-2120-4f37-9896-b66f1a09d0d5", "amount": 138.47217193639062}
data: {"id": "82224f61-34f4-4ee4-8066-91e9f119c36c", "amount": 976.5754411843043}
```

Figure 4-3. *Server-Sent-Events Result*

Note the Content-Type header has a value of `text/event-stream` to indicate that we get a stream of events. The stream can be kept open and receiving event notifications. Each object written is converted to JSON with an `HttpMessageConverter`. Each object is written in the `data` tag as the event data.

If you want to add more info to the event (fill in one of the other fields as mentioned in Table 4-2), use the `SseEventBuilder`. The `event()` factory-method of the `SseEmitter` creates an instance. Use it to fill the `id` and `event` fields.

```
@GetMapping("/orders")
public SseEmitter orders() {
    SseEmitter emitter = new SseEmitter();
    ExecutorService executor = Executors.newSingleThreadExecutor();
    executor.execute(() -> {
        var orders = orderService.findAll();
        try {
            for (var order : orders) {
                randomDelay();
                emitter.event().id(order.getId())
                    .name("order")
                    .data(order)
                    .send();
            }
        } catch (Exception e) {
            emitter.error(e);
        }
    });
    return emitter;
}
```

```

var eventBuilder = event();
emitter.send(
    eventBuilder
        .data(order)
        .name("order-created")
        .id(String.valueOf(order.hashCode())));
}
emitter.complete();
} catch (IOException e) {
    emitter.completeWithError(e);
}
});
executor.shutdown();
return emitter;
}

```

Now when calling the URL `http://localhost:8080/orders`, events contain an id, event, and data field.



The screenshot shows a terminal window titled "marten — martens-imac-van-marten — ~ — zsh — 92x29". The command entered is "curl http://localhost:8080/orders". The response is a text-based event stream. The header information includes:

- HTTP/1.1 200
- Content-Type: text/event-stream; charset=UTF-8
- Date: Wed, 11 Jul 2018 10:07:49 GMT
- Transfer-Encoding: chunked

The body of the response contains five events, each representing a created order:

- data:{"id":"8b32ffa1-9108-4135-a1f1-b576ffab9cbc","amount":571.7930067888439}
event:order-created
id:1253900499
- data:{"id":"2be2f24d-fd82-4f56-994b-c7c0f65de1ed","amount":337.4862782201099}
event:order-created
id:1820233767
- data:{"id":"17153887-5304-43ff-9088-62b1a4cb63db","amount":489.9123343382062}
event:order-created
id:2096994301
- data:{"id":"e515b3b8-a5bc-4f8d-aa3f-2c6576e77913","amount":234.7290986199768}
event:order-created
id:1747749104
- data:{"id":"93b87a4a-b145-4306-a589-c0eae8458757","amount":167.13027374188226}
event:order-created
id:616835188
- data:{"id":"d48b5a1f-d5b7-41da-8374-00a3d9f56c1f","amount":467.42611999725835}
event:order-created
id:1239837839

Figure 4-4. Orders Endpoint Output

Note Server-Sent-Events aren't supported on Microsoft's browsers (Internet Explorer or Edge); to make it work with Microsoft browser you would have to use a polyfill.¹

4-3 WebSockets

Problem

You want to use bidirectional communication from the client to the server over the web.

Solution

Use WebSockets to communicate from the client to the server and vice versa.

WebSockets provide full duplex communication, unlike HTTP.

How It Works

A full explanation of WebSockets goes beyond the scope of this recipe; however, one thing worth mentioning is that the relation between HTTP and WebSockets is actually quite thin. The only usage of HTTP for WebSockets is that the initial handshake uses HTTP, to upgrade the connection from plain HTTP to a TCP socket connection.

Configure WebSocket Support

The first step is to add a dependency on `spring-boot-starter-websocket`, which brings in the needed dependencies and will auto-configure websocket support in Spring Boot.

To enable the use of WebSockets, it is just a matter of adding `@EnableWebSocket` to your application. Do this by adding the annotation to the `@SpringBootApplication` annotated class or an `@Configuration` annotated class.

```
@SpringBootApplication  
@EnableWebSocket  
public class EchoApplication implements WebSocketConfigurer { ... }
```

¹<https://github.com/remy/polyfills>

Creating a WebSocketHandler

To handle WebSocket messages and lifecycle events (handshake, connection established, etc.) you need to create a `WebSocketHandler` and register it to an endpoint URL.

The `WebSocketHandler` defines five methods, which need implementing (Table 4-3) if you would implement this interface directly. However, Spring already provides a nice class hierarchy that you can use to your advantage. When writing your own custom handlers, it is often enough to extend one of `TextWebSocketHandler` or `BinaryWebSocketHandler` which, as their names imply, can handle either text or binary messages.

Table 4-3. *WebSocketHandler methods*

Method	Description
<code>afterConnectionEstablished</code>	Invoked when the websocket connection is open and ready for use
<code>handleMessage</code>	Called when a WebSocket message arrives for this handler
<code>handleTransportError</code>	Called when an error occurs
<code>afterConnectionClosed</code>	Invoked after the websocket connection has been closed
<code>supportsPartialMessages</code>	If this handler supports partial messages: if set true the WebSocket messages can arrive over multiple calls.

Create EchoHandler by extending the `TextWebSocketHandler` and implement the `afterConnectionEstablished` and `handleTextMessage` methods.

```
package com.apress.springbootrecipes.echo;

import org.springframework.web.socket.TextMessage;
import org.springframework.web.socket.WebSocketSession;
import org.springframework.web.socket.handler.TextWebSocketHandler;

public class EchoHandler extends TextWebSocketHandler {
```

```

@Override
public void afterConnectionEstablished(WebSocketSession session)
throws Exception {
    session.sendMessage(new TextMessage("CONNECTION ESTABLISHED"));
}

@Override
protected void handleTextMessage(WebSocketSession session,
                                TextMessage message) throws Exception {
    var msg = message.getPayload();
    session.sendMessage(new TextMessage("RECEIVED: " + msg));
}
}

```

When a connection is established, a TextMessage will be sent back to the client telling it that the connection was established. When a TextMessage is received, the payload (the actual message) is extracted and prefixed with RECEIVED and sent back to the client.

Next you need to register this handler with a URI; to do so you can create a @Configuration class that implements WebSocketConfigurer and register it in the registerWebSocketHandlers method. Add this interface to the EchoApplication class.

```

package com.apress.springbootrecipes.echo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.web.socket.config.annotation.EnableWebSocket;
import org.springframework.web.socket.config.annotation.
WebSocketConfigurer;
import org.springframework.web.socket.config.annotation.
WebSocketHandlerRegistry;

@SpringBootApplication
@EnableWebSocket
public class EchoApplication implements WebSocketConfigurer {

```

```

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

@Bean
public EchoHandler echoHandler() {
    return new EchoHandler();
}

@Override
public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {
    registry.addHandler(echoHandler(), "/echo");
}
}

```

First you register the EchoHandler as a bean so that it can be used to attach it to a URI. In the registerWebSocketHandlers you can use the WebSocketHandlerRegistry to register the handler. Using the addHandler method you can register the handler to a URI, in this case /echo. With this configuration we could use the ws://localhost:8080/echo URL to open a WebSocket connection from the client.

Now that the server is ready, we need a client to connect to our WebSocket endpoint. For this you will need some JavaScript and HTML. Write the following app.js and place it in the src/main/resources/static directory.

```

var ws = null;
var url = "ws://localhost:8080/echo";

function setConnected(connected) {
    document.getElementById('connect').disabled = connected;
    document.getElementById('disconnect').disabled = !connected;
    document.getElementById('echo').disabled = !connected;
}

function connect() {
    ws = new WebSocket(url);

    ws.onopen = function () {
        setConnected(true);
    };
}

```

```
ws.onmessage = function (event) {
    log(event.data);
};

ws.onclose = function (event) {
    setConnected(false);
    log('Info: Closing Connection.');
};

function disconnect() {
    if (ws != null) {
        ws.close();
        ws = null;
    }
    setConnected(false);
}

function echo() {
    if (ws != null) {
        var message = document.getElementById('message').value;
        log('Sent: ' + message);
        ws.send(message);
    } else {
        alert('connection not established, please connect.');
    }
}

function log(message) {
    var console = document.getElementById('logging');
    var p = document.createElement('p');
    p.appendChild(document.createTextNode(message));
    console.appendChild(p);
    while (console.childNodes.length > 12) {
        console.removeChild(console.firstChild);
    }
    console.scrollTop = console.scrollHeight;
}
```

There are a few functions here. The first connect will be invoked when pressing the **Connect** button; this will open a WebSocket connection to ws://localhost:8080/echo, which is the handler created and registered earlier. Connecting to the server will create a WebSocket JavaScript object and that gives you the ability to listen to messages on the client. Here the onopen, onmessage, and onclose callbacks are defined. The most important is the onmessage because that will be invoked whenever a message comes in from the server; this method simply calls the log function, which will add the received message to the logging element on the screen.

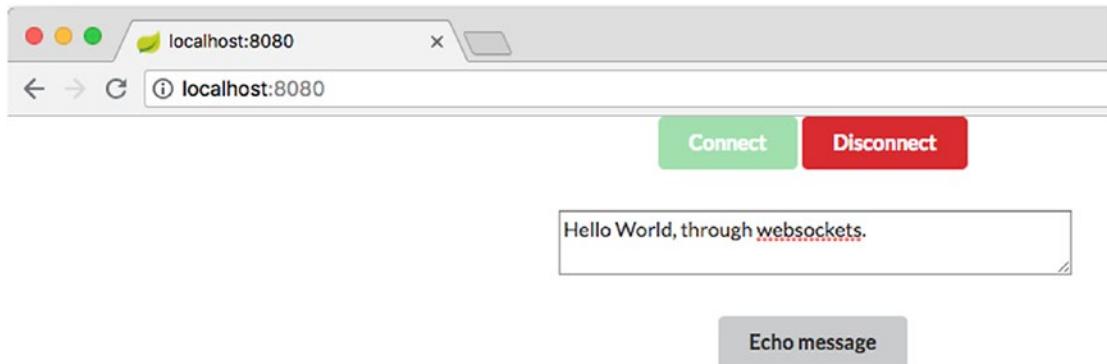
Next there is disconnect, which will close the WebSocket connection and clean up the JavaScript objects. Finally there is the echo function, which will be invoked whenever the **Echo Message** button is pressed. The given message will be sent to the server (and eventually will be returned).

To use the app.js add the next index.html in src/main/resources/static.

```
<!DOCTYPE html>
<html>
<head>
  <link type="text/css" rel="stylesheet" href="https://cdnjs.cloudflare.
  com/ajax/libs/semantic-ui/2.2.10/semantic.min.css" />
  <script type="text/javascript" src="app.js"></script>
</head>
<body>
<div>
  <div id="connect-container" class="ui centered grid">
    <div class="row">
      <button id="connect" onclick="connect();" class="ui green
      button ">Connect</button>
      <button id="disconnect" disabled="disabled"
      onclick="disconnect();" class="ui red button">Disconnect</
      button>
    </div>
    <div class="row">
      <textarea id="message" style="width: 350px" class="ui input"
      placeholder="Message to Echo"></textarea>
    </div>
  </div>
</body>
</html>
```

```
<div class="row">
    <button id="echo" onclick="echo();" disabled="disabled"
        class="ui button">Echo message</button>
</div>
</div>
<div id="console-container">
    <h3>Logging</h3>
    <div id="logging"></div>
</div>
</div>
</body>
</html>
```

Now when deploying the application, you can connect to the echo WebSocket service on <http://localhost:8080> and send a message and have them send it back (Figure 4-5).



Logging

CONNECTION ESTABLISHED

Sent: Spring Boot 2 Recipes

RECEIVED: Spring Boot 2 Recipes

Sent: Hello World, through websockets.

RECEIVED: Hello World, through websockets.

Figure 4-5. WebSocket client output

Unit Testing a WebSocketHandler

You want to write a unit test to make sure that the EchoHandler actually does what you want it to do. To unit test the EchoHandler, write a test and use Mockito (or another mocking framework) to mock the websocket parts.

```
package com.apress.springbootrecipes.echo;

import org.junit.Test;
import org.springframework.web.socket.TextMessage;
import org.springframework.web.socket.WebSocketSession;

import static org.mockito.ArgumentMatchers.eq;
import static org.mockito.Mockito.*;

public class EchoHandlerTest {

    private final EchoHandler handler = new EchoHandler();

    @Test
    public void shouldEchoMessage() throws Exception {
        var mockSession = mock(WebSocketSession.class);
        var msg = new TextMessage("Hello World!");
        handler.handleTextMessage(mockSession, msg);

        verify(mockSession, times(1))
            .sendMessage(eq(new TextMessage("RECEIVED: Hello World!")));
    }
}
```

The test constructs an instance of the EchoHandler. In the test method the WebSocketSession is mocked. The TextMessage can simply be constructed. With the mocked WebSocketSession and TextMessage we call the handleTextMessage. To verify that it does what it is meant to do, we verify the behavior on the mocked WebSocketSession.

Integration Testing Plain WebSockets

Writing an integration test is a little harder. A websocket connection needs to be made to the server, and we need to manually send messages and check the response. But before that the server needs to be started.

To start the server, annotate a class with `@RunWith(SpringRunner.class)` and `@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)`. We actually want to start the application instead of the default WebEnvironment.MOCK. Use the `@LocalServerPort` annotation on an `int` field to get the actual port value. This value is needed to construct the URI to connect to.

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class EchoHandlerIntegrationTest {

    @LocalServerPort
    private int port;
}
```

Testing websockets is fairly easy with the default Java WebSocket API. To test, you can write a basic websocket client that records the received messages and session. Annotate a class with `@ClientEndpoint` and methods in that class with `@OnOpen`, `@OnClose`, and `@OnMessage`. These annotated methods will receive callbacks when the connection is opened, closed, and when a message is received, respectively. Next there are two helper methods: `sendTextAndWait` and `closeAndWait`. The `sendTextAndWait` will send a message using the `Session` and wait for a response. The `closeAndWait` will close the session and wait until that has been confirmed. Finally, there are some getter methods to receive the state and validate it in the test methods.

Add the `SimpleTestClientEndpoint` as a static embedded class in the `EchoHandlerIntegrationTest`.

```
@ClientEndpoint
public static class SimpleTestClientEndpoint {

    private List<String> received = new ArrayList<>();
    private Session session;
    private CloseReason closeReason;
    private boolean closed = false;

    @OnOpen
    public void onOpen(Session session) {
        this.session = session;
    }
}
```

```
@OnClose
public void onClose(Session session, CloseReason reason) {
    this.closeReason = reason;
    this.closed = true;
}

@OnMessage
public void onMessage(String message) {
    this.received.add(message);
}

public void sendTextAndWait(String text, long timeout)
    throws IOException, InterruptedException {
    var current = received.size();
    session.getBasicRemote().sendText(text);
    wait(() -> received.size() == current, timeout);
}

public void closeAndWait(long timeout)
    throws IOException, InterruptedException {
    if (session != null && !closed) {
        session.close();
    }
    wait(() -> closeReason == null, timeout);
}

private void wait(Supplier<Boolean> condition, long timeout)
    throws InterruptedException {
    var waited = 0;
    while (condition.get() && waited < timeout) {
        Thread.sleep(1);
        waited += 1;
    }
}

public CloseReason getCloseReason() {
    return closeReason;
}
```

```
public List<String> getReceived() {
    return this.received;
}

public boolean isClosed() {
    return closed;
}

}
```

Now that the helper classes are in place, write a test. Use the WebSocket ContainerProvider to obtain the container. Before making the actual connection, you have to construct the URI using the port field. Next, use the connectToServer method to connect to the server, using an instance of the SimpleTestClientEndpoint. After connecting, send a text message to the server and wait for some time and close the connection (just to cleanup resources). The final thing is to actually do some assertions on the received messages. Here we expect exactly two messages to be received.

```
@Test
public void sendAndReceiveMessage() throws Exception {
    var container = ContainerProvider.getWebSocketContainer();
    var uri = URI.create("ws://localhost:" + port + "/echo");
    var testClient = new SimpleTestClientEndpoint();
    container.connectToServer(testClient, uri);

    testClient.sendTextAndWait("Hello World!", 200);
    testClient.closeAndWait(2);

    assertThat(testClient.getReceived())
        .containsExactly("CONNECTION ESTABLISHED", "RECEIVED: Hello World!");
}
```

4-4 WebSockets with STOMP

Problem

You want to use STOMP (**S**imple/**S**treaming **T**ext **O**riented **M**essage **P**rotocol) over WebSockets to communicate with the server.

Solution

Configure the message broker and use `@MessageMapping` annotated methods in an `@Controller` annotated class to handle the messages.

How It Works

Let's talk about using WebSockets to create an application that more or less implies messaging. Although you can use the WebSocket protocol as is, the protocol allows you to use subprotocols. One of those protocols, supported by Spring WebSocket, is STOMP.

STOMP is a very simple text-oriented protocol. It was created for scripting languages like Ruby and Python to connect to message brokers. STOMP can be used over any reliable bidirectional network protocol like TCP and also WebSocket. The protocol itself is text-oriented, but the payload of the messages isn't strictly bound to this; it can also contain binary data.

When configuring and using STOMP with Spring WebSocket support, the WebSocket application acts as a broker for all connected clients. The broker can be an in-memory broker or an actual full-blown enterprise solution that supports the STOMP protocol (like RabbitMQ or ActiveMQ). In the latter case the Spring WebSocket application acts as a relay for the actual broker. To add messaging over WebSocket, Spring uses the Spring Messaging abstraction.

Using STOMP and MessageMapping

To be able to receive messages, you need to mark a method in a `@Controller` with `@MessageMapping` and tell it from which destination it will receive messages. Let's modify the `EchoHandler` (from the previous recipe) to work with annotations.

```
package com.apress.springbootrecipes.echo;

import org.springframework.messaging.handler.annotation.MessageMapping;
import org.springframework.messaging.handler.annotation.SendTo;
import org.springframework.stereotype.Controller;
```

```

@Controller
public class EchoHandler {

    @MessageMapping("/echo")
    @SendTo("/topic/echo")
    public String echo(String msg) {
        return "RECEIVED: " + msg;
    }
}

```

When a message is received on the /app/echo destination, it will be passed on to the @MessageMapping annotated method. Notice the @SendTo("/topic/echo") on the method as well, this instructs Spring to put the result, a String, on said topic.

Configure the message broker and add an endpoint for receiving messages. For this, add @EnableWebSocketMessageBroker to the EchoApplication and let it implement the WebSocketMessageBrokerConfigurer.

```

package com.apress.springbootrecipes.echo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.messaging.simp.config.MessageBrokerRegistry;
import org.springframework.web.socket.config.annotation.
EnableWebSocketMessageBroker;
import org.springframework.web.socket.config.annotation.
StompEndpointRegistry;
import org.springframework.web.socket.config.annotation.
WebSocketMessageBrokerConfigurer;

@SpringBootApplication
@EnableWebSocketMessageBroker
public class EchoApplication implements WebSocketMessageBrokerConfigurer {

    @Override
    public void configureMessageBroker(MessageBrokerRegistry registry) {
        registry.enableSimpleBroker("/topic");
        registry.setApplicationDestinationPrefixes("/app");
    }
}

```

```

@Override
public void registerStompEndpoints(StompEndpointRegistry registry) {
    registry.addEndpoint("/echo-endpoint");
}

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

```

The `@EnableWebSocketMessageBroker` will enable the use of messaging over WebSocket. The broker is configured in the `configureMessageBroker` method. Here the simple message broker is used. To connect to an enterprise broker, use the `registry.enableStompBrokerRelay` to connect to the actual broker.

To distinguish between messages handled by the broker or by the app, there are different prefixes (Figure 4-6). Anything on a destination starting with `/topic` will be passed on to the broker; anything on a destination starting with `/app` will be sent to a message handler (i.e., `@MessageMapping` annotated method).

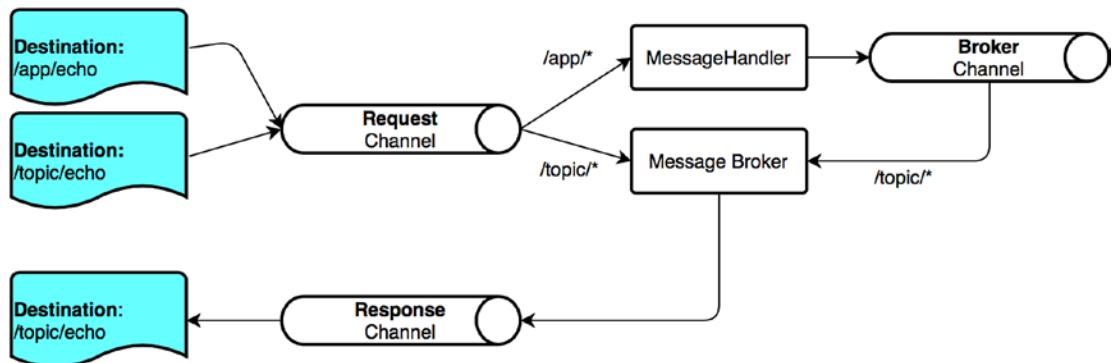


Figure 4-6. WebSockets Stomp channels and routing

The final part is the registration of a WebSocket endpoint that listens to incoming STOMP messages, in this case that is mapped to `/echo-endpoint`.

Modify the client to use STOMP instead of plain WebSocket. The HTML can remain pretty much the same. You need an additional library to be able to work with STOMP in the browser; this recipe uses [webstomp-client](#)² but there are different libraries that you can use.

```
<head>
  <link type="text/css" rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/semantic-ui/2.2.10/semantic.min.css" />
  <script type="text/javascript" src="webstomp.js"></script>
  <script type="text/javascript" src="app.js"></script>
</head>
```

The biggest change is in the app.js file.

```
var ws = null;
var url = "ws://localhost:8080/echo";

function setConnected(connected) {
  document.getElementById('connect').disabled = connected;
  document.getElementById('disconnect').disabled = !connected;
  document.getElementById('echo').disabled = !connected;
}

function connect() {
  ws = webstomp.client(url, {protocols: ['v11.stomp', 'v12.stomp']});
  ws.connect({}, function(frame) {
    setConnected(true);
    log(frame);
    ws.subscribe('/topic/echo', function(message){
      log(message.body);
    })
  });
}
```

²<https://github.com/JSteunou/webstomp-client>

```

function disconnect() {
    if (ws != null) {
        ws.disconnect();
        ws = null;
    }
    setConnected(false);
}

function echo() {
    if (ws != null) {
        var message = document.getElementById('message').value;
        log('Sent: ' + message);
        ws.send("/app/echo", message);
    } else {
        alert('connection not established, please connect.');
    }
}

function log(message) {
    var console = document.getElementById('logging');
    var p = document.createElement('p');
    p.appendChild(document.createTextNode(message));
    console.appendChild(p);
    while (console.childNodes.length > 12) {
        console.removeChild(console.firstChild);
    }
    console.scrollTop = console.scrollHeight;
}

```

The connect function now uses `webstomp.client` to create a STOMP client connection to our broker. When connected, the client will subscribe to `/topic/echo` and receive the messages put on the topic. The echo function has been modified to use the `send` method of the client to send the message to the `/app/echo` destination.

When starting the application and opening the client, you are still able to send and receive messages but now using the STOMP subprotocol. You could even connect multiple browsers and each browser would see the messages on the `/topic/echo` destination, as it acts as a topic.

When writing @MessageMapping annotated methods, you can use a variety of method arguments and annotations (Table 4-4) to receive more or less information on the message. By default it is assumed that a single argument will map to the payload of the message; a MessageConverter will be used to convert the message payload to the desired type.

Table 4-4. Supported Method Arguments and Annotations

Type	Description
Message	The actual underlying message including the headers and body
@Payload	The payload of the message (default); arguments can also be annotated with @Validated to be validated.
@Header	Get the given header from the Message
@Headers	Can be placed on a Map argument to get all Message headers
MessageHeaders	All the Message headers
Principal	The current user, if set

Unit Testing the Handler

Writing a unit test for the handler is relative simple. The method accepts a String and expects a String. To test, simply call the method and assert the returned value.

```
package com.apress.springbootrecipes.echo;

import org.junit.Test;

import static org.assertj.core.api.Assertions.assertThat;

public class EchoHandlerTest {

    private final EchoHandler handler = new EchoHandler();

    @Test
    public void shouldEchoMessage() throws Exception {
        var msg = "Hello World!";
        assertThat(handler.echo(msg)).isEqualTo("RECEIVED: " + msg);
    }
}
```

Integration Testing with STOMP

To do integration testing you would need to start the application, probably on a random port, construct a STOMP client, and send a message. The last part of the test (sending and receiving) is asynchronous, so that makes it a bit more difficult to do testing. First, start the application on a random port and get the random port in the test class.

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class EchoHandlerIntegrationTest {

    @LocalServerPort
    private int port;
}
```

The `@RunWith` instructs JUnit to use the `SpringRunner` to execute the test. The `@SpringBootTest` will start the application, and due to the `WebEnvironment.RANDOM_PORT` it will start at a random port. With the `@LocalServerPort` annotation on an `int` field, we can get the actual port.

Next, to test you can use the `WebSocketStompClient` from the Spring client library to connect to the server and subscribe to a topic. To receive messages on a topic, write a class that extends from `StompSessionHandlerAdapter` and override the `getPayloadType` and `handleFrame` methods. This class, as we don't reuse it, can be a `static` nested class inside the `EchoHandlerIntegrationTest` class.

```
private static class TestStompFrameHandler extends
StompSessionHandlerAdapter {

    private final CompletableFuture<String> answer;

    private TestStompFrameHandler(CompletableFuture<String> answer) {
        this.answer = answer;
    }

    @Override
    public Type getPayloadType(StompHeaders headers) {
        return byte[].class;
    }
}
```

```

@Override
public void handleFrame(StompHeaders headers, Object payload) {
    answer.complete(new String((byte[]) payload));
}
}

```

Due to the asynchronous nature of WebSockets, we need something we can use to block until we receive a response. To transfer the result we use a `CompletableFuture`, and when asserting we can use the `get` method to block until we receive an answer (or timeout).

For the test case, we need to create a STOMP client and use that to connect to the STOMP broker, subscribe to the `/topic/echo` to receive messages, and finally send a message and check the result.

```

@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class EchoHandlerIntegrationTest {

    @LocalServerPort
    private int port;

    private WebSocketStompClient stompClient;

    private List<StompSession> sessions = new ArrayList<>();

    @Before
    public void setup() {
        var webSocketClient = new StandardWebSocketClient();
        stompClient = new WebSocketStompClient(webSocketClient);
    }

    @After
    public void cleanUp() {
        this.sessions.forEach(StompSession::disconnect);
        this.sessions.clear();
    }
}

```

```

@Test
public void shouldSendAndReceiveMessage() throws Exception {
    CompletableFuture<String> answer = new CompletableFuture<>();
    var stompSession = connectAndSubscribe(answer);

    stompSession.send("/app/echo", "Hello World!".getBytes());

    var result = answer.get(1, TimeUnit.SECONDS);
    assertThat(result).isEqualTo("RECEIVED: Hello World!");
}

private StompSession connectAndSubscribe(CompletableFuture<String> answer)
    throws InterruptedException, ExecutionException, TimeoutException {
    var uri = "ws://localhost:" + port + "/echo-endpoint";
    var stompSession =
        stompClient.connect(uri, new StompSessionHandlerAdapter() {})
            .get(1, TimeUnit.SECONDS);

    stompSession.subscribe("/topic/echo",
        new TestStompFrameHandler(answer));
    this.sessions.add(stompSession);
    return stompSession;
}
...
}

```

In the @Before the WebSocketStompClient is created, and it uses a StandardWebSocketClient as a transport. In the @After method, we clean up any sessions that have connected to the broker (just to be sure not to leave anything behind). The connectAndSubscribe method uses the WebSocketStompClient to connect to the broker and here it is configured to wait 1 second to be connected. The StompSession is then used to subscribe the earlier created TestStompFrameHandler using the passed in CompletableFuture. Of course you could do without the helper method, but generally you need more integration tests and with this you can create a reusable piece of code. Finally the actual test: it first connects and subscribes, then it sends Hello World! to the server and expects an answer in 1 second (or less). The result is checked with the expected result.

CHAPTER 5

Spring WebFlux

5-1 Developing a Reactive Application with Spring WebFlux

Problem

You want to develop a simple reactive web application with Spring WebFlux to learn the basic concepts and configurations of this framework.

Solution

The lowest component of Spring WebFlux is the `HttpHandler`, an interface with a single `handle` method.

```
public interface HttpHandler {  
    Mono<Void> handle(ServerHttpRequest request, ServerHttpResponse response);  
}
```

The `handle` method returns a `Mono<Void>`, which is the reactive way of saying it returns `void`. It takes both a `ServerHttpRequest` and `ServerHttpResponse` from the `org.springframework.http.server.reactive` package. These are again interfaces and depending on the container used for running, an instance of the interface is created. For this, several adapters or bridges for containers exist; when running on a Servlet 3.1 container (supporting nonblocking IO) the `ServletHttpHandlerAdapter` (or one of its subclasses) is used to adapt from the plain Servlet world to the Reactive world. When running on a native Reactive engine like Netty¹ the `ReactorHttpHandlerAdapter` is used.

¹<https://netty.io>

When a web request is sent to a Spring WebFlux application, the `HandlerAdapter` first receives the request. Then it organizes the different components configured in Spring’s application context—all needed to handle the request.

To define a controller class in Spring WebFlux, a class has to be marked with the `@Controller` or `@RestController` annotation (just as with Spring MVC; see Chapter 3).

When a `@Controller` annotated class (i.e., a controller class) receives a request, it looks for an appropriate handler method to handle the request. This requires that a controller class map each request to a handler method by one or more handler mappings. In order to do so, a controller class’s methods are decorated with the `@RequestMapping` annotation, making them handler methods.

The signature for these handler methods—as you can expect from any standard class—is open ended. You can specify an arbitrary name for a handler method and define a variety of method arguments. Equally, a handler method can return any of a series of values (e.g., `String` or `void`), depending on the application logic it fulfills. The following is only a partial list of valid argument types, just to give you an idea.

1. `ServerHttpRequest` or `ServerHttpResponse`
2. Request parameters from the URL of arbitrary type, annotated with `@RequestParam`
3. Cookie values included in an incoming request, annotated with `@CookieValue`
4. Request header values of arbitrary type, annotated with `@RequestHeader`
5. Request attribute of arbitrary type, annotated with `@RequestAttribute`
6. `Map` or `ModelMap`, for the handler method to add attributes to the model
7. `WebSession`, for the session

Once the controller class has picked an appropriate handler method, it invokes the handler method’s logic with the request. Usually, a controller’s logic invokes back-end services to handle the request. In addition, a handler method’s logic is likely to add or remove information from the numerous input arguments (e.g., `ServerHttpRequest`, `Map` or `Errors`) that will form part of the ongoing flow.

After a handler method has finished processing the request, it delegates control to a view, which is represented as the handler method’s return value. To provide a flexible approach, a handler method’s return value doesn’t represent a view’s implementation (e.g., `user.html` or `report.pdf`) but rather a logical view (e.g., `user` or `report`)—note the lack of file extension.

A handler method’s return value can be either a `String`—representing a logical view name—or `void`, in which case a default logical view name is determined on the basis of a handler method’s or controller’s name.

In order to pass information from a controller to a view, it’s irrelevant that a handler’s method returns a logical view name—`String` or a `void`—since the handler method input arguments will be available to a view.

For example, if a handler method takes `Map` and `Model` objects as input parameters—modifying their contents inside the handler method’s logic—these same objects will be accessible to the view returned by the handler method.

When the controller class receives a view, it resolves the logical view name into a specific view implementation (e.g., `user.html` or `report.fxml`) by means of a view resolver. A view resolver is a bean configured in the web application context that implements the `ViewResolver` interface. Its responsibility is to return a specific view implementation for a logical view name.

Once the controller class has resolved a view name into a view implementation, per the view implementation’s design, it renders the objects (e.g., `ServerHttpRequest`, `Map`, `Errors`, or `WebSession`) passed by the controller’s handler method. The view’s responsibility is to display the objects added in the handler method’s logic to the user.

How It Works

Let’s write a reactive version of the `HelloWorldApplication` from Recipe 4-1.

```
package com.apress.springboottrecipes.helloworld;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
import reactor.core.publisher.Mono;

@RestController
public class HelloWorldController {
```

```

@GetMapping
public Mono<String> hello() {
    return Mono.just("Hello World, from Spring Boot 2!");
}
}

```

Notice the `Mono<String>` is a return type for the `hello` method, instead of a plain `String`. The `Mono` is what makes it reactive.

Setting Up a Spring WebFlux Application

To be able to handle requests in a reactive way, you need to enable webflux. This is done by adding a dependency on `spring-boot-starter-webflux`.

```

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>

```

This brings in the needed dependencies like `spring-webflux` and the project Reactor (<http://www.projectreactor.io>) dependencies. It also includes a reactive runtime, which by default is Netty.

Now that everything is configured, the final thing to do is create an application class.

```

package com.apress.springbootrecipes.library;

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

@SpringBootApplication
public class HelloWorldApplication {

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

```

Spring Boot will detect the reactive runtime and configure it using the `server.*` properties (see Recipe 3-7).

Creating Spring WebFlux Controllers

An annotation-based controller class can be an arbitrary class that doesn't implement a particular interface or extend a particular base class. You can annotate it with the `@Controller` or `@RestController` annotation. There can be one or more handler methods defined in a controller to handle single or multiple actions. The signature of the handler methods is flexible enough to accept a range of arguments.

```
package com.apress.springbootrecipes.helloworld;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
import reactor.core.publisher.Mono;

@RestController
public class HelloWorldController {

    @GetMapping("/hello")
    public Mono<String> hello() {
        return Mono.just("Hello World, from Reactive Spring Boot 2!");
    }
}
```

The annotation `@GetMapping` is used to decorate the `hello` method as the controller's HTTP GET handler method. It's worth mentioning that if no default HTTP GET handler method is declared, a `ServletException` is thrown—hence the importance of a controller having at a minimum a URL route and at least one handler method. The method is bound to `/hello` due to the expression in the `@GetMapping`.

When a request is made on `/hello` it will reactively return `Hello World, from Reactive Spring Boot 2!`, although the client won't notice this. For the client it still is a regular HTTP request.

Unit Test for Reactive Controllers

You have two ways of doing an integration test for a controller. The first approach is to simply write a test that creates an instance of the `HelloWorldController`, call the method, and do expectations on the result. The second is to use the `@WebFluxTest` annotation to create the test. The latter will start a minimal application context

containing the web infrastructure and you can use MockMvc to test the controller. This last approach sits between a plain unit test and a full-blown integration test.

```
package com.apress.springbootrecipes.helloworld;

import org.junit.Test;
import reactor.core.publisher.Mono;
import reactor.test.StepVerifier;

public class HelloWorldControllerTest {

    private final HelloWorldController controller = new
    HelloWorldController();

    @Test
    public void shouldSayHello() {
        Mono<String> result = controller.hello();

        StepVerifier.create(result)
            .expectNext("Hello World, from Reactive Spring Boot 2!")
            .verifyComplete();
    }
}
```

This is a basic unit test. It instantiates the controller, and simply calls the method to be tested. It uses the StepVerifier from the reactive-test module to make it easier to test. The hello method is called, and the result is then verified using the StepVerifier.

Adding the reactive-test dependency is done by adding the following dependency:

```
<dependency>
    <groupId>io.projectreactor</groupId>
    <artifactId>reactor-test</artifactId>
    <scope>test</scope>
</dependency>
```

The second option is to use @WebFluxTest for a specific controller.

```
@RunWith(SpringRunner.class)
@WebFluxTest(HelloWorldController.class)
public class HelloWorldControllerSliceTest {
```

```

@Autowired
private WebTestClient webClient;

@Test
public void shouldSayHello() {
    webClient.get().uri("/hello").accept(MediaType.TEXT_PLAIN)
        .exchange()
        .expectStatus().isOk()
        .expectBody(String.class)
        .isEqualTo("Hello World, from Reactive Spring Boot 2!");
}
}

```

This test will start a minimal Spring Boot context and auto detect all web related beans in your project, like `@ControllerAdvice`, `@Controller`, etc. Instead of now directly calling the `HelloWorldController`, you can use the special `WebTestClient` to declare a request `.get().uri("/hello")` and send that nonblocking using `exchange()`. Finally, the assertions/expectations are verified. The request is expected to have an OK status and to contain the specified body.

Integration Test for Reactive Controllers

The integration test looks very much like the `@WebFluxTest` in the previous section. The main difference is the use of `@SpringBootTest` instead of `@WebFluxText`. Using `@SpringBootTest` this will bootstrap the full application, including all other beans (services, repositories, etc.). With the `webEnvironment` you can specify which environment to use: values are `RANDOM_PORT`, `MOCK` (default), `DEFINED_PORT`, and `NONE`. Here we use a random port and again use the `WebTestClient` to fire off a request.

```

@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class HelloWorldControllerIntegrationTest {

    @Autowired
    private WebTestClient webClient;
}

```

```

@Test
public void shouldSayHello() {
    webClient.get().uri("/hello").accept(MediaType.TEXT_PLAIN)
        .exchange()
        .expectStatus().isOk()
        .expectBody(String.class)
        .isEqualTo("Hello World, from Reactive Spring Boot 2!");
}
}

```

The request is sent to the embedded server, after which the result is verified to have the correct status and body content.

Note When using MOCK (also the default) as the webEnvironment you must add the @AutoConfigureWebTestClient annotation to get the WebTestClient for testing.

5-2 Publishing and Consuming with Reactive Rest Services

Problem

You want to write a reactive REST endpoint that will produce JSON.

Solution

Just as with a regular @RestController, you can return a regular object or list of objects and those will be sent to the client. To make them reactive, you will have to wrap those return values in their reactive counterparts: a Mono or a Flux.

How it Works

Let's start by writing a Reactive OrderService. Each method will return either a Mono<Order> or a Flux<Order>.

```
package com.apress.springbootrecipes.order;

@Service
public class OrderService {

    private final Map<String, Order> orders = new ConcurrentHashMap<>(10);

    @PostConstruct
    public void init() {
        OrderGenerator generator = new OrderGenerator();
        for (int i = 0; i < 25; i++) {
            var order = generator.generate();
            orders.put(order.getId(), order);
        }
    }

    public Mono<Order> findById(String id) {
        return Mono.justOrEmpty(orders.get(id));
    }

    public Mono<Order> save(Mono<Order> order) {
        return order.map(this::save);
    }

    private Order save(Order order) {
        orders.put(order.getId(), order);
        return order;
    }

    public Flux<Order> orders() {
        return Flux.fromIterable(orders.values()).delayElements(Duration.ofMillis(128));
    }
}
```

The OrderService creates 25 random orders at startup. It has some simple methods to obtain or save an order. When retrieving all orders, it will delay them for 128 milliseconds. Next to this service, create a basic Order class and OrderGenerator.

```
package com.apress.springbootrecipes.order;
// Imports omitted
public class Order {

    private String id;
    private BigDecimal amount;

    public Order() {
    }

    public Order(String id, BigDecimal amount) {
        this.id=id;
        this.amount = amount;
    }

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public BigDecimal getAmount() {
        return amount;
    }

    public void setAmount(BigDecimal amount) {
        this.amount = amount;
    }

    @Override
    public String toString() {
        return String.format("Order [id='%s', amount=%4.2f]", id, amount);
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
```

```

Order order = (Order) o;
return Objects.equals(id, order.id) &&
    Objects.equals(amount, order.amount);
}

@Override
public int hashCode() {
    return Objects.hash(id, amount);
}
}

```

This is the full Order class; it holds nothing more than an id and a total amount.

The OrderGenerator is a simple component for creating Order instances.

```

package com.apress.springbootrecipes.order;

import java.math.BigDecimal;
import java.util.UUID;
import java.util.concurrent.ThreadLocalRandom;

public class OrderGenerator {

    public Order generate() {
        var amount = ThreadLocalRandom.current().nextDouble(1000.00);
        return new Order(UUID.randomUUID().toString(),
            BigDecimal.valueOf(amount));
    }
}

```

An OrderController is needed to expose the Order as a REST resource.

```

package com.apress.springbootrecipes.order.web;

import com.apress.springbootrecipes.order.Order;
import com.apress.springbootrecipes.order.OrderService;
import org.springframework.web.bind.annotation.*;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

```

```

@RestController
@RequestMapping("/orders")
public class OrderController {

    private final OrderService orderService;

    OrderController(OrderService orderService) {
        this.orderService = orderService;
    }

    @PostMapping
    public Mono<Order> store(@RequestBody Mono<Order> order) {
        return orderService.save(order);
    }

    @GetMapping("/{id}")
    public Mono<Order> find(@PathVariable("id") String id) {
        return orderService.findById(id);
    }

    @GetMapping
    public Flux<Order> list() {
        return orderService.orders();
    }
}

```

The OrderController is mapped to /orders and supports listing all orders or a single one and adding/modifying an order. To bootstrap everything, a simple OrderApplication is needed.

```

package com.apress.springbootrecipes.order;

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

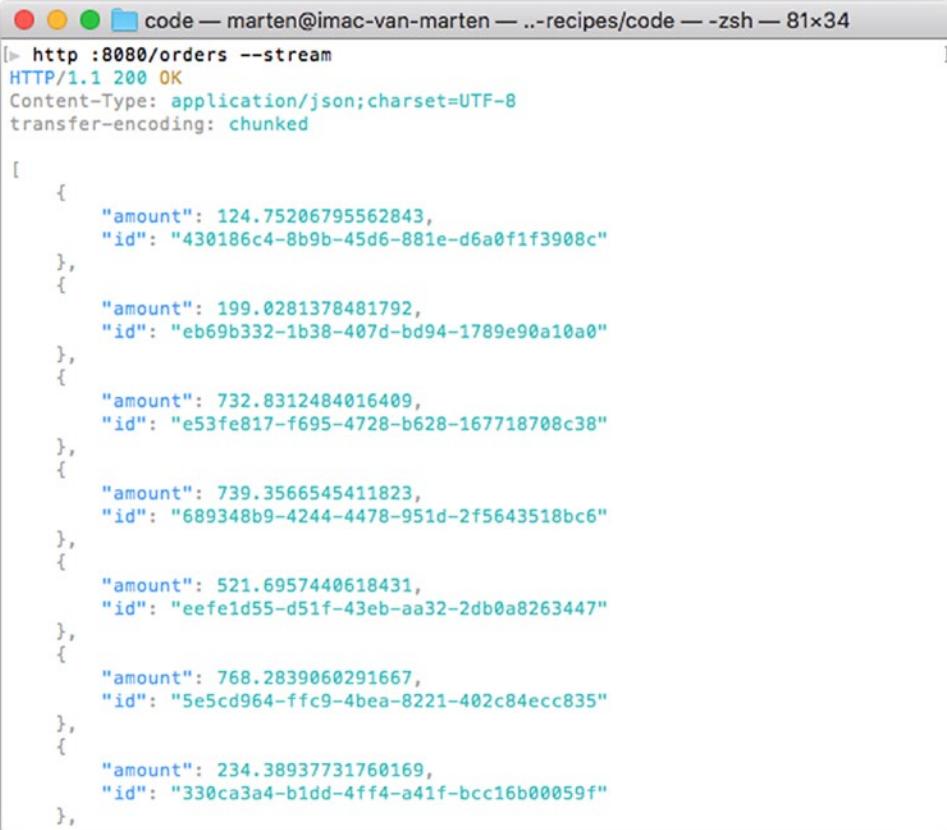
@SpringBootApplication
public class OrderApplication {

```

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

Using something like cUrl or httpie, you can query the endpoint.

An `http http://localhost:8080/orders --stream` should list all the orders in the system (Figure 5-1) and `http http://localhost:8080/orders/{some-id}` should list a single one.



The screenshot shows a terminal window on a Mac OS X desktop. The title bar says "code — marten@imac-van-marten — ..-recipes/code — -zsh — 81x34". The terminal content is as follows:

```
[> http :8080/orders --stream
HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8
transfer-encoding: chunked

[{"id": "430186c4-8b9b-45d6-881e-d6a0f1f3908c", "amount": 124.75206795562843}, {"id": "eb69b332-1b38-407d-bd94-1789e90a10a0", "amount": 199.0281378481792}, {"id": "e53fe817-f695-4728-b628-167718708c38", "amount": 732.8312484016409}, {"id": "689348b9-4244-4478-951d-2f5643518bc6", "amount": 739.3566545411823}, {"id": "eefe1d55-d51f-43eb-aa32-2db0a8263447", "amount": 521.6957440618431}, {"id": "5e5cd964-ffc9-4bea-8221-402c84ecc835", "amount": 768.2839060291667}, {"id": "330ca3a4-b1dd-4ff4-a41f-bcc16b00059f", "amount": 234.38937731760169}]
```

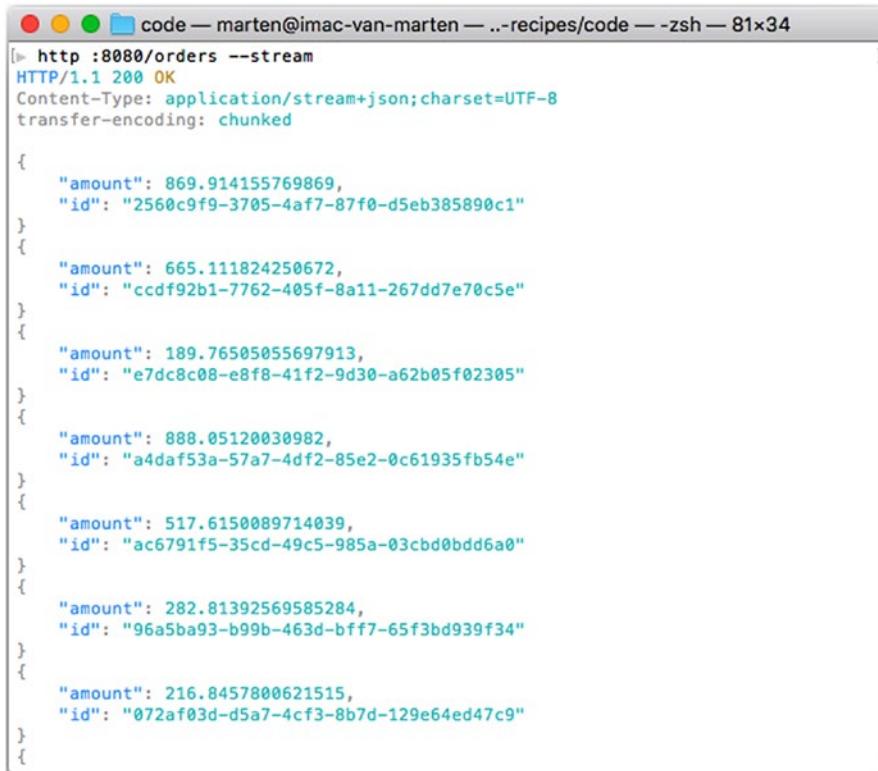
Figure 5-1. Result of getting all orders

Streaming JSON

The result of the call to `/orders` isn't streaming, but rather it is blocking. It first collects all the results before sending them out. See also the Content-Type header in Figure 5-1; it is set to `application/json`. To stream the result, it should be `application/stream+json`. For this, modify the controller's `list` method.

```
@GetMapping(produces = MediaType.APPLICATION_STREAM_JSON_VALUE)
public Flux<Order> list() {
    return orderService.orders();
}
```

Notice the `produces = MediaType.APPLICATION_STREAM_JSON_VALUE`. This instructs Spring to stream the results when a part is ready. Restart the application and again issue an `http://localhost:8080/orders --stream`. The result will now gently stream in until there is nothing more to consume (Figure 5-2).



The screenshot shows a terminal window with the following content:

```
[~] http :8080/orders --stream
HTTP/1.1 200 OK
Content-Type: application/stream+json; charset=UTF-8
transfer-encoding: chunked

{
    "amount": 869.914155769869,
    "id": "2560c9f9-3705-4af7-87f0-d5eb385890c1"
}
{
    "amount": 665.111824250672,
    "id": "ccdf92b1-7762-405f-8a11-267dd7e70c5e"
}
{
    "amount": 189.76505055697913,
    "id": "e7dc8c08-e8f8-41f2-9d30-a62b05f02305"
}
{
    "amount": 888.05120030982,
    "id": "a4daf53a-57a7-4df2-85e2-0c61935fb54e"
}
{
    "amount": 517.6150089714039,
    "id": "ac6791f5-35cd-49c5-985a-03cbd0bdd6a0"
}
{
    "amount": 282.81392569585284,
    "id": "96a5ba93-b99b-463d-bff7-65f3bd939f34"
}
{
    "amount": 216.8457800621515,
    "id": "072af03d-d5a7-4cf3-8b7d-129e64ed47c9"
}
```

Figure 5-2. Result of streaming all orders

The result also slightly changed: instead of returning an array or orders (see Figure 5-1) it now returns single orders (see Figure 5-2).

Server Sent Events

Instead of using a stream of JSON, one can also use Server Sent Events. With WebFlux this is as easy as changing the produces in the @GetMapping to MediaType.TEXT_EVENT_STREAM_VALUE. Then events will be written.

```
@GetMapping(produces = MediaType.TEXT_EVENT_STREAM_VALUE)
public Flux<Order> list() {
    return orderService.orders();
}
```

Restart the application and again issue an `http http://localhost:8080/orders --stream`. The result, as events, will now gently stream in until there is nothing more to consume (Figure 5-3). Notice that the content type has changed to `text/event-stream`.



```
manuscript — marten@imac-van-marten — ..es/manuscript — zsh — 108x24
> http --stream :8080/orders
HTTP/1.1 200 OK
Content-Type: text/event-stream;charset=UTF-8
transfer-encoding: chunked

data:{"id":"c2d6772d-882d-4675-a651-4570a9af0125","amount":297.2396366549731}
data:{"id":"fddff6cd7-dbe9-4ff0-8f6f-dfb8ee3663df","amount":305.2656834693431}
data:{"id":"2f4a4c14-2ecc-4c18-b551-9fb87167052","amount":696.706842452895}
data:{"id":"94c1772e-42f7-4de9-bd5a-af5a5e59df62","amount":0.1775470518441402}
data:{"id":"136fce3-538d-4336-978d-bd890dcabcf3","amount":93.55790629640903}
data:{"id":"91bb34ab-021f-4507-ae20-9a16a947b561","amount":924.8909365656588}
data:{"id":"ebef3aad-0df1-4877-bd83-8cbf04cee72e","amount":103.94281039673325}
data:{"id":"e4002f19-2215-454f-b451-88637f886079","amount":194.99013045150792}
data:{"id":"55f42c4d-832b-4fef-b2dd-54fc6179b0bc","amount":920.3041222593154}
data:{"id":"a6503643-defa-4c2c-acfe-de6be42e09e9","amount":121.79381700959424}
```

Figure 5-3. Result of streaming all orders as events

Write an Integration Test

An integration test for the OrderController can be written quite easily using the WebTestClient and a MOCK web environment (you could also use a RANDOM_PORT instead of MOCK).

```
package com.apress.springbootrecipes.order.web;

import com.apress.springbootrecipes.order.Order;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.reactive.
AutoConfigureWebTestClient;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.annotation.DirtiesContext;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.web.reactive.server.WebTestClient;

import java.math.BigDecimal;

@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = SpringApplication.WebEnvironment.MOCK)
@AutoConfigureWebTestClient
@DirtiesContext(classMode = DirtiesContext.ClassMode.AFTER_EACH_TEST_METHOD)
public class OrderControllerIntegrationTest {

    @Autowired
    private WebTestClient webTestClient;

    @Test
    public void listOrders() {
        webTestClient.get().uri("/orders")
            .exchange()
            .expectStatus().isOk()
            .expectBodyList(Order.class).hasSize(25);
    }

    @Test
    public void addAndGetOrder() {
```

```

var order = new Order("test1", BigDecimal.valueOf(1234.56));
webTestClient.post().uri("/orders").syncBody(order)
    .exchange()
    .expectStatus().isOk()
    .expectBody(Order.class).isEqualTo(order);

webTestClient.get().uri("/orders/{id}", order.getId())
    .exchange()
    .expectStatus().isOk()
    .expectBody(Order.class).isEqualTo(order);
}

}

```

The `@DirtiesContext` is needed here because the `OrderService` is a stateful bean. So after adding an `Order` to the collection, it needs to be reset for the next test. The test will start the full application, due to the `@SpringBootTest` with a mocked environment. The `@AutoConfigureWebTestClient` is needed for a mocked environment to get a `WebTestClient`. The `WebTestClient` makes it easy to build a request and send that to the server. After sending the response, it can do expectations on the result.

5-3 Use Thymeleaf as a Template Engine

Problem

You want to render a view in a WebFlux based application.

Solution

Use Thymeleaf to create a view and reactively return the view name and fill the model.

How It Works

Add a dependency on `spring-boot-starter-webflux` and `spring-boot-starter-thymeleaf`. This is enough for Spring Boot to automatically configure Thymeleaf for use in a WebFlux application.

```

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>

```

With the addition of this dependency, you will get the Thymeleaf library as well as the Thymeleaf Spring Dialect so that they integrate nicely. Due to the existence of these two libraries, Spring Boot will automatically configure the ThymeleafReactiveViewResolver.

The ThymeleafReactiveViewResolver requires a Thymeleaf ISpringWebFluxTemplateEngine to be able to resolve and render the views. A special SpringWebFluxTemplateEngine will be preconfigured with the SpringDialect so that you can use SpEL inside Thymeleaf pages.

To configure Thymeleaf, Spring Boot exposes several properties in the `spring.thymeleaf` and `spring.thymeleaf.reactive` namespaces (Table 5-1).

Table 5-1. Common Thymeleaf Properties

Property	Description
<code>spring.thymeleaf.prefix</code>	The prefix to use for the ViewResolver, default <code>classpath:/templates/</code>
<code>spring.thymeleaf.suffix</code>	The suffix to use for the ViewResolver, default <code>.html</code>
<code>spring.thymeleaf.encoding</code>	The encoding of the templates, default UTF-8
<code>spring.thymeleaf.check-template</code>	Check if the template exists before rendering, default true.
<code>spring.thymeleaf.check-template-location</code>	Check if the template location exists. Default is true.
<code>spring.thymeleaf.mode</code>	Thymeleaf TemplateMode to use, default HTML
<code>spring.thymeleaf.cache</code>	Should resolved templates be cached or not, default true

(continued)

Table 5-1. (continued)

Property	Description
spring.thymeleaf.template-resolver-order	Order of the ViewResolver default is 1
spring.thymeleaf.view-names	The view names (comma separated) that can be resolved with this ViewResolver
spring.thymeleaf.exclude-view-names	The view names (comma separated) that are excluded from being resolved
spring.thymeleaf.enabled	Should Thymeleaf be enabled, default true
spring.thymeleaf.enable-spring-el-compiler	Enable the compilation of SpEL expressions, default false
spring.thymeleaf.reactive.max-chunk-size	Maximum size of databuffers used to write the response in bytes. Default 0
spring.thymeleaf.reactive.media-types	Media types supported by this view technology, like text/html
spring.thymeleaf.reactive.full-mode-view-names	Comma-separated list of view names that should operate in FULL mode. Default none. Full mode means basically blocking mode.
spring.thymeleaf.reactive.chunked-mode-view-names	Comma-separated list of view names that should operate in chunked mode

Use Thymeleaf Views

First create an `index.html` in the `src/main/resources/templates` directory.

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Spring Boot - Orders</title>
</head>
```

```
<body>
```

```
<h1>Order Management System</h1>
```

```
<a th:href="@{/orders}" href="#">List of orders</a>
```

```
</body>
```

```
</html>
```

This page will render a simple page with a single link, pointing to /orders. This URL is rendered using the th:href tag, which will expand /orders to a proper URL. Next, write a controller that will select the page to render.

```
package com.apress.springbootrecipes.order.web

@Controller
class IndexController {

    @GetMapping
    public String index() {
        return "index";
    }
}
```

Write an OrderController, which will return the orders/list as the name of the view and adds Flux<Order> to the model.

```
package com.apress.springbootrecipes.order.web

@Controller
@RequestMapping("/orders")
class OrderController {

    private final OrderService orderService;

    OrderController(OrderService orderService) {
        this.orderService = orderService;
    }
}
```

```

@GetMapping
public Mono<String> list(Model model) {
    var orders = orderService.orders();
    model.addAttribute("orders", orders);
    return Mono.just("orders/list");
}
}

```

For the Order and OrderService and related objects, see Recipe 5.2.

Now that the controller and the other needed components are ready, a view is required. Create a `list.html` in the `src/main/resources/templates/orders` directory. This page will render a table with the orders showing the id and the amount of the different orders.

```

<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Orders</title>
</head>
<body>
    <h1>Orders</h1>

    <table>
        <thead>
            <tr>
                <th></th>
                <th>Id</th>
                <th>Amount</th>
            </tr>
        </thead>
        <tbody>
            <tr th:each="order : ${orders}">
                <td th:text="${orderStat.count}">1</td>
                <td th:text="${order.id}"></td>
            
```

```
<td th:text="#{numbers.formatCurrency(order.amount)}"  
style="text-align: right"></td>  
</tr>  
</tbody>  
</table>  
</body>  
</html>
```

Finally, the OrderApplication to start the application.

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

Now when launching the application and clicking the link, you will end up with a page showing the orders. See Figure 5-4.

Id	Amount
3e1a5584-1911-4ecc-be77-9f8efa2a5468	¤589.98
48a50717-c48e-4638-9979-62e115d5b1a6	¤609.99
94bc003c-d3fb-40f2-8ce9-fdca5b7a15b3	¤119.21
c91d4978-c07c-4a43-a137-e6e7de848cc9	¤90.38
e0d7ad70-6cc8-4449-95d6-54ec4f595c2d	¤593.93
1ba2f188-410f-4965-929a-82031d9d409a	¤939.13
9d650d3a-fe6f-4f3a-a623-89eaf8bb71a2	¤246.51
ca4b8fc4-2860-4a21-956c-d479591d08a5	¤640.66
22051056-0bf2-43f7-9305-2e1a262a84cc	¤805.20
4c5b1842-7f5d-411c-92f0-c21f400ef13b	¤454.02
2391c5f2-fc31-4ab8-8a73-9a7513bec70c	¤602.36
72228870-7386-4868-80b7-41a6cbc2a6aa	¤283.09
4e2d2ee1-227c-4368-9a21-63b788366968	¤230.91
6f445cab-cb0c-4785-a7dc-6e5cce082ac29	¤667.87
801eca2b-95a8-4680-b05c-c975d9f1926f	¤680.43
71963289-5691-47b9-815e-f58a87f92dac	¤112.41
ffbfdf04-c58f-4149-bc50-b7378e43cc41	¤793.51
93325d72-3ed6-4870-a5ad-054dcdbbeec3f	¤401.11
b79e6585-c459-4973-9b35-3a211979b07d	¤958.63
4f24c442-7e46-4870-9be8-9385bd0fe6f6	¤487.76
3172fed7-96d2-44a0-9b10-1bdccb6511c1e	¤988.62
98b16dce-50f2-4e29-96f9-73a9c9b6d65d	¤712.26
6b7a6bc1-4e12-435c-a121-25498cd17e96	¤766.22
f2c1403b-abf0-4427-bbe8-9a3c7bdd60e	¤148.60
41075133-5a2a-4163-a6ee-1d23f1fcdb4b	¤676.51

Figure 5-4. Orders list

Making it More Reactive

When running the application and opening the page, you have to wait some time before the page starts to render. When rendering a page and the model contains Flux, it will by default wait until the whole Flux is consumed. After that it will start rendering the page. It basically behaves like you would retrieve a Collection instead of a Flux. To start rendering the page quicker and get the page streaming, wrap the Flux in a `ReactiveDataDriverContextVariable`.

```

@GetMapping
public Mono<String> list(Model model) {
    var orders = orderService.orders();
    model.addAttribute("orders", new ReactiveDataContextVariable(orders, 5));
    return Mono.just("orders/list");
}

```

The list method looks the same, but notice that the Flux is wrapped in the `ReactiveDataContextVariable`. It will start rendering as soon as five elements are received and will continue to render the page until everything is received. Now when opening the orders page you will see the table grow until all orders are read.

5-4 WebFlux and WebSockets

Problem

You want to use WebSockets in a Reactive application.

Solution

Include the `javax.websocket-api` as a dependency and use the `ReactiveWebSocketHandler` interface to implement the handler.

How It Works

Adding a dependency on `javax.websocket-api` next to `spring-boot-starter-webflux` is enough to get WebSockets support running in a reactive application.

```

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
<dependency>
    <groupId>javax.websocket</groupId>
    <artifactId>javax.websocket-api</artifactId>
    <version>1.1</version>
</dependency>

```

Spring WebFlux expects that WebSockets version 1.1 is used. When using 1.0, the code won't run.

Next, use the `WebSocketHandler` to implement a reactive handler.

```
package com.apress.springbootrecipes.echo;

import org.springframework.web.reactive.socket.WebSocketHandler;
import org.springframework.web.reactive.socket.WebSocketSession;
import reactor.core.publisher.Mono;

public class EchoHandler implements WebSocketHandler {

    @Override
    public Mono<Void> handle(WebSocketSession session) {
        return session.send(
            session.receive()
                .map( msg -> "RECEIVED: " + msg.getPayloadAsText())
                .map(session::textMessage));
    }
}
```

The `EchoHandler` will receive a message and return it prefixed with “RECEIVED:” (see also recipe 4.4 for the nonreactive version).

Setting up WebSockets with WebFlux requires a few components to be registered. First we need the handler. The handler needs to be mapped to a URL using the `SimpleUrlHandlerMapping`. We need something that can invoke our handler, in this case the `WebSocketHandlerAdapter`. The final part is to let the `WebSocketHandlerAdapter` understand the incoming reactive runtime request. As we use Netty (the default), we need to configure a `WebSocketService` with the `ReactorNettyRequestUpgradeStrategy`.

```
package com.apress.springbootrecipes.echo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.core.Ordered;
import org.springframework.web.reactive.HandlerMapping;
import org.springframework.web.reactive.handler.SimpleUrlHandlerMapping;
```

```
import org.springframework.web.reactive.socket.WebSocketHandler;
import org.springframework.web.reactive.socket.server.WebSocketService;
import org.springframework.web.reactive.socket.server.support.
HandshakeWebSocketService;
import org.springframework.web.reactive.socket.server.support.
WebSocketHandlerAdapter;
import org.springframework.web.reactive.socket.server.upgrade.
ReactorNettyRequestUpgradeStrategy;

import java.util.HashMap;
import java.util.Map;

@SpringBootApplication
public class EchoApplication {

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

    @Bean
    public EchoHandler echoHandler() {
        return new EchoHandler();
    }

    @Bean
    public HandlerMapping handlerMapping() {
        Map<String, WebSocketHandler> map = new HashMap<>();
        map.put("/echo", echoHandler());

        var mapping = new SimpleUrlHandlerMapping();
        mapping.setUrlMap(map);
        mapping.setOrder(Ordered.HIGHEST_PRECEDENCE);
        return mapping;
    }

    @Bean
    public WebSocketHandlerAdapter handlerAdapter() {
        return new WebSocketHandlerAdapter(webSocketService());
    }
}
```

```

@Bean
public WebSocketService webSocketService() {
    var strategy = new ReactorNettyRequestUpgradeStrategy();
    return new HandshakeWebSocketService(strategy);
}
}

```

Now you can launch the server and handle WebSockets with WebFlux.

HTML & JavaScript Echo Client

Now that the server is ready, we need a client to connect to our WebSocket endpoint. For this you will need some JavaScript and HTML. Write the following `app.js` and place it in the `src/main/resources/static` directory.

```

var ws = null;
var url = "ws://localhost:8080/echo";

function setConnected(connected) {
    document.getElementById('connect').disabled = connected;
    document.getElementById('disconnect').disabled = !connected;
    document.getElementById('echo').disabled = !connected;
}

function connect() {
    ws = new WebSocket(url);

    ws.onopen = function () {
        setConnected(true);
    };

    ws.onmessage = function (event) {
        log(event.data);
    };

    ws.onclose = function (event) {
        setConnected(false);
        log('Info: Closing Connection.');
    };
}

```

```

function disconnect() {
    if (ws != null) {
        ws.close();
        ws = null;
    }
    setConnected(false);
}

function echo() {
    if (ws != null) {
        var message = document.getElementById('message').value;
        log('Sent: ' + message);
        ws.send(message);
    } else {
        alert('connection not established, please connect.');
    }
}

function log(message) {
    var console = document.getElementById('logging');
    var p = document.createElement('p');
    p.appendChild(document.createTextNode(message));
    console.appendChild(p);
    while (console.childNodes.length > 12) {
        console.removeChild(console.firstChild);
    }
    console.scrollTop = console.scrollHeight;
}

```

There are a few functions here. The first `connect` will be invoked when pressing the **Connect** button; this will open a WebSocket connection to `ws://localhost:8080/echo`, which is the handler created and registered earlier. Connecting to the server will create a WebSocket JavaScript object, and that gives you the ability to listen to messages and events on the client. Here the `onopen`, `onmessage`, and `onclose` callbacks are defined. The most important is the `onmessage` because that will be invoked whenever a message comes in from the server; this method simply calls the `log` function, which will add the received message to the `logging` element on the screen.

Next there is disconnect, which will close the WebSocket connection and clean up the JavaScript objects. Finally, there is the echo function, which will be invoked whenever the **Echo message** button is pressed. The given message will be sent to the server (and eventually will be returned).

To use the app.js add the next index.html.

```
<!DOCTYPE html>
<html>
<head>
    <link type="text/css" rel="stylesheet" href="https://cdnjs.cloudflare.
    com/ajax/libs/semantic-ui/2.2.10/semantic.min.css" />
    <script type="text/javascript" src="app.js"></script>
</head>
<body>
<div>
    <div id="connect-container" class="ui centered grid">
        <div class="row">
            <button id="connect" onclick="connect();" class="ui green
            button ">Connect</button>
            <button id="disconnect" disabled="disabled"
            onclick="disconnect();" class="ui red button">Disconnect
            </button>
        </div>
        <div class="row">
            <textarea id="message" style="width: 350px" class="ui input"
            placeholder="Message to Echo"></textarea>
        </div>
        <div class="row">
            <button id="echo" onclick="echo();" disabled="disabled"
            class="ui button">Echo message</button>
        </div>
    </div>
    <div id="console-container">
        <h3>Logging</h3>
        <div id="logging"></div>
    </div>
```

```
</div>
</body>
</html>
```

Now when deploying the application you can connect to the echo WebSocket service and send some message and have them send it back (Figure 5-5).

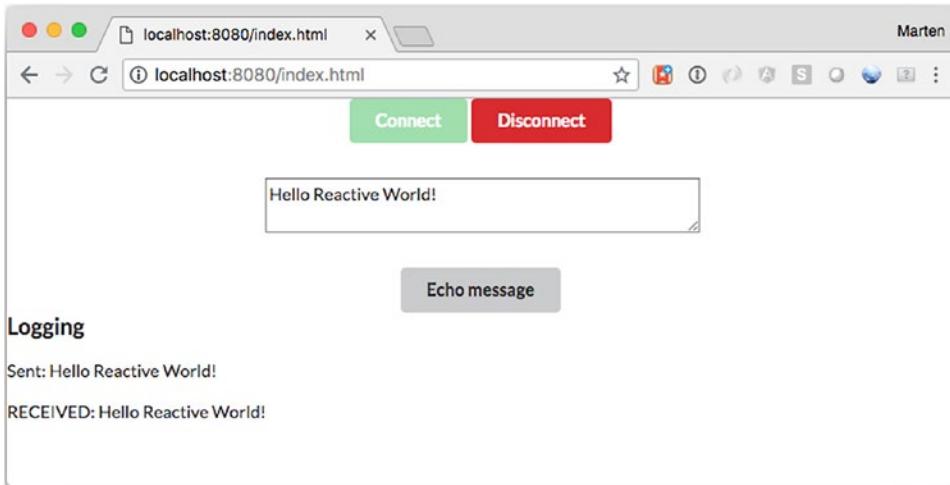


Figure 5-5. WebSocket client output

Write an Integration Test

Writing an integration test requires some work. A WebSocket connection needs to be made to the server, and we need to manually send messages and check the response. But before that the server needs to be started.

To start the server, annotate a class with `@RunWith(SpringRunner.class)` and `@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)`. We actually want to start the application instead of the default `WebEnvironment.MOCK`. Use the `@LocalServerPort` annotation on an `int` field to get the actual port value. This value is needed to construct the URI to connect to.

```
package com.apress.springbootrecipes.echo;

@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class EchoHandlerIntegrationTest {
```

```

@LocalServerPort
private int port;
}

```

Testing websockets is fairly easy with the default Java WebSocket API. To test, you can write a basic WebSockets client that records the received messages and session. For this you would need a WebSocket client, like Tomcat.

```

<dependency>
  <groupId>org.apache.tomcat.embed</groupId>
  <artifactId>tomcat-embed-websocket</artifactId>
  <scope>test</scope>
</dependency>

```

Adding this client, however, will result in bootstrapping a Tomcat server instead of the reactive Netty that we want. Add the following to the EchoHandlerIntegrationTest to circumvent this.

```

public class EchoHandlerIntegrationTest {

    // Remainder of class omitted

    @Configuration
    @Import(EchoApplication.class)
    public static class EchoHandlerIntegrationTestConfiguratation {

        @Bean
        public NettyReactiveWebServerFactory webServerFactory() {
            return new NettyReactiveWebServerFactory();
        }
    }
}

```

This additional `@Configuration` class will explicitly launch Netty instead of Tomcat. This way we test against the same runtime as we would do when normally running our application. Depending on your client (or way of testing) this might not be needed.

Now annotate a class with `@ClientEndpoint` and methods in that class with `@OnOpen`, `@OnClose`, and `@OnMessage`. These annotated methods will receive callbacks when the connection is opened, closed, and when a message is received, respectively. Next there are two helper methods: `sendTextAndWait` and `closeAndWait`. The `sendTextAndWait` will send a message using the `Session` and wait for a response. The `closeAndWait` will close the session and wait until that has been confirmed. Finally, there are some getter methods to receive the state and validate it in the test methods.

```
@ClientEndpoint
public static class SimpleTestClientEndpoint {

    private List<String> received = new ArrayList<>();
    private Session session;
    private CloseReason closeReason;
    private boolean closed = false;

    @OnOpen
    public void onOpen(Session session) {
        this.session = session;
    }

    @OnClose
    public void onClose(Session session, CloseReason reason) {
        this.closeReason = reason;
        this.closed = true;
    }

    @OnMessage
    public void onMessage(String message) {
        this.received.add(message);
    }

    public void sendTextAndWait(String text, long timeout)
        throws IOException, InterruptedException {
        var current = received.size();
        session.getBasicRemote().sendText(text);
        wait(() -> received.size() == current, timeout);
    }
}
```

```

public void closeAndWait(long timeout)
    throws IOException, InterruptedException {
    if (session != null && !closed) {
        session.close();
    }
    wait(() -> closeReason == null, timeout);
}

private void wait(Supplier<Boolean> condition, long timeout)
    throws InterruptedException {
    var waited = 0;
    while (condition.get() && waited < timeout) {
        Thread.sleep(1);
        waited += 1;
    }
}

public CloseReason getCloseReason() {
    return closeReason;
}

public List<String> getReceived() {
    return this.received;
}

public boolean isClosed() {
    return closed;
}
}

```

Now that the helper classes are in place, write the test. Use the WebSocket ContainerProvider to obtain the container. Before making the actual connection, you have to construct the URI using the port field. Next, use the connectToServer method to connect to the server, using an instance of the SimpleTestClientEndpoint. After connecting, send a text message to the server and wait for some time and close the connection (just to cleanup resources). The final thing is to actually do some assertions on the received messages.

```
@Test
public void sendAndReceiveMessage() throws Exception {
    var container = ContainerProvider.getWebSocketContainer();
    var uri = URI.create("ws://localhost:" + port + "/echo");
    var testClient = new SimpleTestClientEndpoint();
    container.connectToServer(testClient, uri);

    testClient.sendTextAndWait("Hello World!", 200);
    testClient.closeAndWait(2);

    assertThat(testClient.getReceived())
        .containsExactly("RECEIVED: Hello World!");

}
```

CHAPTER 6

Spring Security

In this chapter we will take a look at the Spring Security¹ integration for Spring Boot. Spring Security can be used for both authentication and authorization of users for your application. Spring Security has a pluggable mechanism for both the authentication and authorization process and by default, supports different mechanisms. For authentication, Spring Security out-of-the-box has support for JDBC, LDAP, and property files.

6-1 Enable Security in Your Spring Boot Application

Problem

You have a Spring Boot-based application and you want to enable security in this application.

Solution

Add the `spring-boot-starter-security` as a dependency to have security automatically configured and set up for your application.

How it Works

To get started you will need to get the libraries for Spring Security into your application; to do this you can add the `spring-boot-starter-security` to your list of dependencies.

¹<https://projects.spring.io/spring-security/>

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

This will add the `spring-security-core`, `spring-security-config`, and `spring-security-web` dependencies to your project. Spring Boot detects the availability of some of the classes inside these JAR files, and with that will automatically enable security.

Spring Boot will configure Spring Security with the following: authentication with Basic authentication and form login; enable HTTP headers for security; Servlet API integration; anonymous login; and disable caching of resources.

Warning Spring Boot will add a default user, named `user`, with a random generated password, visible in the startup logs. This is only intended to be used for testing, prototyping, or demos; do not use the generated user in a live system!.

When adding the dependency `spring-boot-starter-security` to Recipe 3.2, it will automatically secure all exposed endpoints. At startup the generated password will be logged (Figure 6-1).

```
2018-09-23 19:58:38.882  INFO 58204 --- [           main] .s.s.UserDetailsServiceAutoConfiguration : Using generated security password: 033397ce-d724-4c7a-a717-edab7747f99d
2018-09-23 19:58:38.932 DEBUG 58204 --- [           main] s.s.c.a.w.c.WebSecurityConfigurerAdapter :
```

Figure 6-1. Generated password Output

Spring Boot exposes some properties to configure the default user; those can be found in the `spring.security` namespace (Table 6-1).

Table 6-1. Properties for Default User

Property	Description
<code>spring.security.user.name</code>	Default user name, default user
<code>spring.security.user.password</code>	Password for the default user, default generated UUID
<code>spring.security.user.roles</code>	Roles for the default user. Default none

After adding the dependency and starting the LibraryApplication, the endpoints are secured. When trying to obtain a list of books from `http://localhost:8080/books` the result will be an HTTP result with status 401 - Unauthorized (Figure 6-2).

```
code/ch06/recipe_6_1_i  master ✘
▶ http :8080/books
HTTP/1.1 401
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Content-Type: application/json; charset=UTF-8
Date: Sun, 23 Sep 2018 18:08:46 GMT
Expires: 0
Pragma: no-cache
Set-Cookie: JSESSIONID=EB5370933EB2198BE3B766B4F9A2C339; Path=/; HttpOnly
Transfer-Encoding: chunked
WWW-Authenticate: Basic realm="Realm"
X-Content-Type-Options: nosniff
X-Frame-Options: DENY
X-XSS-Protection: 1; mode=block

{
    "error": "Unauthorized",
    "message": "Unauthorized",
    "path": "/books",
    "status": 401,
    "timestamp": "2018-09-23T18:08:46.032+0000"
}
```

Figure 6-2. Unauthenticated access result

When adding the correct authentication headers (username user, password from the logging or as specified in the `spring.security.user.password` property), the result will be the regular list of books (Figure 6-3).

```
~/Repositories/spring-boot-recipes master ✘
▶ http -a user:033397ce-d724-4c7a-a717-edab7747f99d :8080/books
HTTP/1.1 200
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Content-Type: application/json;charset=UTF-8
Date: Mon, 24 Sep 2018 17:41:50 GMT
Expires: 0
Pragma: no-cache
Set-Cookie: JSESSIONID=D434219EEF940FB083C1DC2B44BB1717; Path=/; HttpOnly
Transfer-Encoding: chunked
X-Content-Type-Options: nosniff
X-Frame-Options: DENY
X-XSS-Protection: 1; mode=block

[
  {
    "authors": [
      "J.R.R. Tolkien"
    ],
    "isbn": "9780618260300",
    "title": "The Hobbit"
  },
  {
    "authors": [
      "George Orwell"
    ],
    "isbn": "9780451524935",
    "title": "1984"
  }
]
```

Figure 6-3. Authenticated access result

Testing Security

When using Spring Security to secure your endpoints together with an `@WebMvcTest`, the security infrastructure will be automatically applied. Spring Security provides some nice annotations to help in writing tests (Table 6-2).

Note If you want to test your controllers without security, you can disable it by setting the `secure` attribute on `@WebMvcTest` to `false`; it defaults to `true`.

Table 6-2. Spring Security Annotations for Testing

Annotation	Description
@WithMockUser	Run as the user with the given username, password, and roles/authorities
@WithAnonymousUser	Run as an Anonymous user
@WithUserDetails	Run as the user with the configured name, does a lookup in the UserDetailsService

To use these annotations, a dependency to `spring-security-test` must be added.

```
<dependency>
<groupId>org.springframework.security</groupId>
<artifactId>spring-security-test</artifactId>
<scope>test</scope>
</dependency>
```

With this dependency the `BookControllerTest` from Recipe 3.2 can be extended and fixed again. If you don't mind disabling security for your test, you could add `@WebMvcTest (value = BookController.class, secure = false)` to the test class. With it, the security filter won't be added and thus security will be disabled. The tests will run and succeed.

```
@RunWith(SpringRunner.class)
@WebMvcTest(value = BookController.class, secure = false)
public class BookControllerUnsecuredTest { ... }
```

If, however, you want to test with security enabled, you will need to make some minor modifications to the test class. First, add `@WithMockUser` to run with an authenticated user. Second, as Spring Security by default enabled CSRF² protection, a header or parameter needs to be added to the request. When using Mock MVC, Spring Security provides a `RequestPostProcessor` for that, the `CsrfRequestPostProcessor`. The `SecurityMockMvcRequestPostProcessors` contains factory methods to easily use them.

²Cross Site Request Forgery

```

@RunWith(SpringRunner.class)
@WebMvcTest(BookController.class)
@WithMockUser
public class BookControllerSecuredTest {

    @Test
    public void shouldAddBook() throws Exception {
        when(bookService.create(any(Book.class))).thenReturn(new
Book("123456789", "Test Book Stored", "T. Author"));

        mockMvc.perform(post("/books")
                .contentType(MediaType.APPLICATION_JSON)
                .content("{\"isbn\" : \"123456789\"}, \"title\" : \"Test
Book\", \"authors\" : [\"T. Author\"]}")
                .with(csrf())
                .andExpect(status().isCreated())
                .andExpect(header()
                        .string("Location", "http://localhost/books/123456789"));
    }
}

```

The test now uses the user as specified in `@WithMockUser`; here it uses the default user, with `user` as the username and `password` as the password. The line `with(csrf())` takes care of adding the CSRF token to the request.

Which option to use depends on the needs you have. If for instance in your controller you need the current user, then security should probably be enabled and an `@WithMockUser` or `@WithUserDetails` should be used. If that isn't the case and you can test your controller without the security (and you don't have additional security rules [see Recipe 6.2]), you can run with disabled security.

Integration Testing Security

When using `@SpringBootTest` to write an integration test, it depends on if you can use the `@With*` annotations. With the default mocked environment, it still will work.

```

@RunWith(SpringRunner.class)
@SpringBootTest

```

```
@WithMockUser
@AutoConfigureMockMvc
public class BookControllerIntegrationMockTest { ... }
```

This test will create an almost full-blown application but still uses Mock MVC to access the endpoints. It still runs in the same process as the tests, and that is why the `@WithMockUser` and `with(csrf())` still work. When running the test on an external port, it won't work anymore.

To test the application on a port, you would need to run the tests through the test client `TestRestTemplate` and/or `WebTestClient` and pass the authentication headers or implement the flow by first doing a form-based login in your integration test. To write a successful integration test, inject the `TestRestTemplate`, and before doing the actual request use the `withBasicAuth` helper method to set the basic authentication header.

Tip When writing this test and using the default user, you might want to set a default password using the `spring.security.user.password`. This recipe uses a `@TestPropertySource` to do so, but you could add it to the `application.properties` as well.

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
@TestPropertySource(properties = "spring.security.user.password=s3cr3t")
public class BookControllerIntegrationTest {

    @Autowired
    private TestRestTemplate testRestTemplate;

    @MockBean
    private BookService bookService;

    @Test
    public void shouldReturnListOfBooks() throws Exception {
        when(bookService.findAll()).thenReturn(Arrays.asList(
            new Book("123", "Spring 5 Recipes", "Marten Deinum", "Josh Long"),
            new Book("321", "Pro Spring MVC", "Marten Deinum", "Colin
Yates")));
    }
}
```

```

    ResponseEntity<Book[]> books = testRestTemplate
        .withBasicAuth("user", "s3cr3t")
        .getForEntity("/books", Book[].class);

    assertThat(books.getStatusCode()).isEqualTo(HttpStatus.OK);
    assertThat(books.getBody()).hasSize(2);
}
}

```

The test uses the default-configured `TestRestTemplate` to issue a request. The `withBasicAuth` uses the default user and preset `s3cr3t` as the username and password to send to the server. The `getForEntity` can be used to get the result, including some additional information about the response. Using the `ResponseEntity` it is also possible to validate the status code, etc.

When testing a WebFlux-based application instead of the `TestRestTemplate`, you would need the `WebTestClient` (see also Chapter 5 for more information on Spring WebFlux). The `WebTestClient` has the `headers()` function to add additional headers to the request. It exposes the `HttpHeaders`, which in turn has the convenient `setBasicAuth` method to apply the basic authentication.

```

@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
@TestPropertySource(properties = "spring.security.user.password=s3cr3t")
public class BookControllerIntegrationWebClientTest {

    @Autowired
    private WebTestClient webTestClient;

    @MockBean
    private BookService bookService;

    @Test
    public void shouldReturnListOfBooks() throws Exception {
        when(bookService.findAll()).thenReturn(Arrays.asList(
            new Book("123", "Spring 5 Recipes", "Marten Deinum", "Josh Long"),
            new Book("321", "Pro Spring MVC", "Marten Deinum", "Colin
Yates")));
    }
}

```

```
webTestClient
    .get()
        .uri("/books")
        .headers( headers -> headers.setBasicAuth("user", "s3cr3t"))
    .exchange()
        .expectStatus().isOk()
        .expectBodyList(Book.class).hasSize(2);
}
}
```

The request is built and “fired” using `exchange()`. Then the result is expected to be an HTTP 200 (OK) and the result contains two books.

6-2 Logging into Web Applications

Problem

A secure application requires its users to log in before they can access certain secure functions. This is especially important for applications running on the open Internet, because hackers can easily reach them. Most applications have to provide a way for users to input their credentials to log in.

Solution

Spring Security supports multiple ways for users to log into a web application. It supports form-based login by providing a default web page that contains a login form. You can also provide a custom web page as the login page. In addition, Spring Security supports HTTP Basic authentication by processing the Basic authentication credentials presented in HTTP request headers. HTTP Basic authentication can also be used for authenticating requests made with remoting protocols and web services.

Some parts of your application may allow for anonymous access (e.g., access to the welcome page). Spring Security provides an anonymous login service that can assign a principal and grant authorities to an anonymous user, so that you can handle an anonymous user like a normal user when defining security policies.

Spring Security also supports remember-me login, which is able to remember a user's identity across multiple browser sessions so that a user needn't log in again after logging in for the first time.

How It Works

Spring Boot enables the default security settings when no explicit `WebSecurityConfigurerAdapter` can be found. When one or more are found, it will use those to configure the security.

To help you better understand the various login mechanisms in isolation, let's first disable the default security configuration.

Caution You generally want to stick with the defaults and just disable what you don't want, like `httpBasic().disable()`, instead of disabling all of the security defaults!

```
@Configuration
public class LibrarySecurityConfig extends WebSecurityConfigurerAdapter {

    public LibrarySecurityConfig() {
        super(true); // disable default configuration
    }
}
```

Note that the login services introduced next will be registered automatically if you enable HTTP auto-config. However, if you disable the default configuration or you want to customize these services, you have to configure the corresponding features explicitly.

Before enabling the authentication features you will have to enable basic Spring Security requirements; you need at least to configure exception handling and security context integration.

```
@Override
protected void configure(HttpSecurity http) {

    http.securityContext()
        .and()
        .exceptionHandling();
}
```

Without these basics, Spring Security wouldn't store the user after doing a login and it wouldn't do proper exception translation for security-related exceptions (they would simply bubble up, which might expose some of your internals to the outside world). You also might want to enable the Servlet API integration so that you can use the methods on the `HttpServletRequest` to do checks in your view.

@Override

```
protected void configure(HttpSecurity http) {
    http.servletApi();
}
```

HTTP Basic Authentication

The HTTP Basic authentication support can be configured via the `httpBasic()` method. When HTTP Basic authentication is required, a browser will typically display a login dialog or a browser-specific login page for users to log in.

@Configuration

```
public class LibrarySecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            ...
            .httpBasic();
    }
}
```

Form-Based Login

The form-based login service will render a web page that contains a login form for users to input their login details and process the login form submission. It's configured via the `formLogin` method.

@Configuration

```
public class LibrarySecurityConfig extends WebSecurityConfigurerAdapter {
```

```

@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        ...
        .formLogin()
}
}

```

By default, Spring Security automatically creates a login page and maps it to the URL /login. So, you can add a link to your application (e.g., in index.html of Recipe 3.3), referring to this URL for login:

```
<a th:href="/login" href="#">Login</a>
```

If you don't prefer the default login page, you can provide a custom login page of your own. For example, you can create the following login.html file in src/main/resources/templates (when using Thymeleaf). As by default the CSRF protection is on, you need to add a CSRF token to the form. That is what the hidden field is for. If you disabled CSRF (not recommended), you should remove this line.

```

<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Login</title>
    <link type="text/css" rel="stylesheet"
          href="https://cdnjs.cloudflare.com/ajax/libs/semantic-ui/2.2.10/
          semantic.min.css">
    <style type="text/css">
        body {
            background-color: #DADADA;
        }
        body > .grid {
            height: 100%;
        }
        .column {
            max-width: 450px;
        }
    </style>

```

```

</head>

<body>
<div class="ui middle aligned center aligned grid">
  <div class="column">
    <h2 class="ui header">Log-in to your account</h2>
    <form method="POST" th:action="@{/login}" class="ui large form">
      <input type="hidden"
        th:name="${_csrf.parameterName}" th:value="${_csrf.token}"/>
      <div class="ui stacked segment">
        <div class="field">
          <div class="ui left icon input">
            <i class="user icon"></i>
            <input type="text" name="username" placeholder="E-
              mail address">
          </div>
        </div>
        <div class="field">
          <div class="ui left icon input">
            <i class="lock icon"></i>
            <input type="password" name="password"
              placeholder="Password">
          </div>
        </div>
        <button class="ui fluid large submit green button">Login</
          button>
      </div>
    </form>
  </div>
</div>
</body>
</html>

```

In order for Spring Security to display your custom login page when a login is requested, you have to specify its URL in the `loginPage` configuration method.

```

@Configuration
public class LibrarySecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            ...
            .formLogin().loginPage("/login");
    }
}

```

Finally, add a view resolver to map /login to the login.html page. For this you can have the LibrarySecurityConfig implement the WebMvcConfigurer and override the addViewControllers methods.

```

@Configuration
public class LibrarySecurityConfig extends WebSecurityConfigurerAdapter
    implements WebMvcConfigurer {

    ...

    public void addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController("/login").setViewName("login");
    }
}

```

If the login page is displayed by Spring Security when a user requests a secure URL, the user will be redirected to the target URL once the login succeeds. However, if the user requests the login page directly via its URL, by default the user will be redirected to the context path's root (i.e., `http://localhost:8080/`) after a successful login. If you have not defined a welcome page in your web deployment descriptor, you may wish to redirect the user to a default target URL when the login succeeds.

```

@Configuration
public class LibrarySecurityConfig extends WebSecurityConfigurerAdapter
    implements WebMvcConfigurer {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            ...
            .formLogin().loginPage("/login").defaultSuccessUrl("/books");
    }
}

```

If you use the default login page created by Spring Security, then when a login fails, Spring Security will render the login page again with the error message. However, if you specify a custom login page, you will have to configure the authentication-failure-url to specify which URL to redirect to on login error. For example, you can redirect to the custom login page again with the error request parameter.

```

@Configuration
public class LibrarySecurityConfig extends WebSecurityConfigurerAdapter
    implements WebMvcConfigurer {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            ...
            .formLogin()
                .loginPage("/login.html")
                .defaultSuccessUrl("/books")
                .failureUrl("/login.html?error=true");
    }
}

```

Then your login page should test whether the error request parameter is present. If an error has occurred, you will have to display the error message by accessing the session scope attribute SPRING_SECURITY_LAST_EXCEPTION, which stores the last Exception thrown by Spring Security for the current user.

```

<form>
    ...
    <div th:if="${param.error}">
        <div class="ui error message" style="display: block;">
            Authentication Failed<br/>
            Reason :
            <span th:text="${session.SPRING_SECURITY_LAST_EXCEPTION.message}" />
        </div>
    </div>
</form>

```

The Logout Service

The logout service provides a handler to handle logout requests. It can be configured via the `logout()` configuration method.

```

@Configuration
public class LibrarySecurityConfig extends WebSecurityConfigurerAdapter
    implements WebMvcConfigurer {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            ...
            .and()
            .logout();
    }
}

```

By default, it's mapped to the URL `/logout` and will react to POST requests only. You can add a small html form to your page to log out.

```
<form th:action="/logout" method="post"><button>Logout</button></form>
```

Note When using CSRF protection, don't forget to add the CSRF token to the form, else logout will fail.

By default, a user will be redirected to the context path's root when the logout succeeds, but sometimes you may wish to direct the user to another URL, which you can do by using the `logoutSuccessUrl` configuration method.

```
@Configuration
public class LibrarySecurityConfig extends WebSecurityConfigurerAdapter
    implements WebMvcConfigurer {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            ...
            .and()
            .logout().logoutSuccessUrl("/");
    }
}
```

After logout you might notice that when using the browser back button you will still be able to see the previous pages, even if your logout was successful. This has to do with the fact that the browser caches the pages. By enabling the security headers with the `headers()` configuration method, the browser will be instructed to not cache the page.

```
@Configuration
public class LibrarySecurityConfig extends WebSecurityConfigurerAdapter
    implements WebMvcConfigurer {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            ...
            .and()
            .headers();
    }
}
```

Next to the no-cache headers this will also disable content sniffing and enable x-frame protection. With this enabled and using the browser back button, you will be redirected to the login page again.

Anonymous Login

The anonymous login service can be configured via the `anonymous()` method in Java config, where you can customize the username and authorities of an anonymous user, whose default values are `anonymousUser` and `ROLE_ANONYMOUS`.

```
@Configuration
public class LibrarySecurityConfig extends WebSecurityConfigurerAdapter
    implements WebMvcConfigurer {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            ...
            .and()
            .anonymous().principal("guest").authorities("ROLE_GUEST");
    }
}
```

Remember-Me Support

Remember-me support can be configured via the `rememberMe()` method in Java config. By default, it encodes the username, password, remember-me expiration time, and a private key as a token, and stores it as a cookie in the user's browser. The next time the user accesses the same web application, this token will be detected so that the user can log in automatically.

```
@Configuration
public class LibrarySecurityConfig extends WebSecurityConfigurerAdapter
    implements WebMvcConfigurer {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            ...
    }
}
```

```
.and()  
    .rememberMe();  
}  
}
```

However, static remember-me tokens can cause security issues because they may be captured by hackers. Spring Security supports rolling tokens for more advanced security needs, but this requires a database to persist the tokens. For details about rolling remember-me token deployment, please refer to the Spring Security reference documentation.

6-3 Authenticating Users

Problem

When a user attempts to log into your application to access its secure resources, you have to authenticate the user's principal and grant authorities to this user.

Solution

In Spring Security, authentication is performed by one or more `AuthenticationProviders`, connected as a chain. If any of these providers authenticates a user successfully, that user will be able to log into the application. If any provider reports that the user is disabled or locked or that the credential is incorrect, or if no provider can authenticate the user, then the user will be unable to log into this application.

Spring Security supports multiple ways of authenticating users and includes built-in provider implementations for them. You can easily configure these providers with the built-in XML elements. Most common authentication providers authenticate users against a user repository storing user details (e.g., in an application's memory, a relational database, or an LDAP repository).

When storing user details in a repository, you should avoid storing user passwords in clear text because that makes them vulnerable to hackers. Instead, you should always store encrypted passwords in your repository. A typical way of encrypting passwords is to use a one-way hash function to encode the passwords. When a user enters a password to log in, you apply the same hash function to this password and compare the result

with the one stored in the repository. Spring Security supports several algorithms for encoding passwords (including BCrypt and SCrypt) and provides built-in password encoders for these algorithms.

How It Works

Authenticating Users with In-Memory Definitions

If you have only a few users in your application and you seldom modify their details, you can consider defining the user details in Spring Security's configuration file so that they will be loaded into your application's memory.

```
@Configuration
public class LibrarySecurityConfig extends WebSecurityConfigurerAdapter {

    ...
    @Override
    protected void configure(AuthenticationManagerBuilder auth)
        throws Exception {
        UserDetails adminUser = User.withDefaultPasswordEncoder()
            .username("admin@books.io")
            .password("secret")
            .authorities("ADMIN", "USER").build();

        UserDetails normalUser = User.withDefaultPasswordEncoder()
            .username("marten@books.io")
            .password("user")
            .authorities("USER").build();

        UserDetails disabledUser = User.withDefaultPasswordEncoder()
            .username("marten@books.io")
            .password("user")
            .disabled(true)
            .authorities("USER").build();
    }
}
```

```

        auth.inMemoryAuthentication()
            .withUser(adminUser)
            .withUser(normalUser)
            .withUser(disabledUser);
    }
}

```

Using the `UserBuild`, obtained through `User.withDefaultPasswordEncoder()`, you can construct users with encrypted passwords. It will use the Spring Security default password encoded (which by is by default encoding with BCrypt). You can add user details with the `inMemoryAuthentication()` method; using the `withUser` method, you can define the users. For each user you can specify a username, a password, a disabled status, and a set of granted authorities. A disabled user cannot log into an application.

Authenticating Users Against a Database

More typically, user details should be stored in a database for easy maintenance. Spring Security has built-in support for querying user details from a database. By default, it queries user details, including authorities, with the following SQL statements:

```

SELECT username, password, enabled
FROM users
WHERE username = ?

SELECT username, authority
FROM authorities
WHERE username = ?

```

In order for Spring Security to query user details with these SQL statements, you have to create the corresponding tables in your database. For example, you can create them in the database with the following SQL statements:

```

CREATE TABLE USERS (
    USERNAME    VARCHAR(50)    NOT NULL,
    PASSWORD   VARCHAR(50)    NOT NULL,
    ENABLED     SMALLINT       NOT NULL,
    PRIMARY KEY (USERNAME)
);

```

```
CREATE TABLE AUTHORITIES (
    USERNAME      VARCHAR(50)      NOT NULL,
    AUTHORITY     VARCHAR(50)      NOT NULL,
    FOREIGN KEY (USERNAME) REFERENCES USERS
);
```

Next, you can input some user details into these tables for testing purposes. The data for these two tables is shown in Tables 6-3 and 6-4.

Table 6-3. Testing User Data for the USERS Table

USERNAME	PASSWORD	ENABLED
admin@books.io	{noop}secret	1
marten@books.io	{noop}user	1
jdoe@books.net	{noop}unknown	0

Note The {noop} in the password field indicates that no encryption has been applied to the stored password. Spring Security uses delegation to determine which encoding method to use; values can be {bcrypt}, {scrypt}, {pbkdf2} and {sha256}. The {sha256} is mainly there for compatibility reasons and should be considered unsecure.

Table 6-4. Testing User Data for the AUTHORITIES Table

USERNAME	AUTHORITY
admin@books.io	ADMIN
admin@books.io	USER
marten@books.io	USER
jdoe@books.net	USER

In order for Spring Security to access these tables, you have to declare a data source to be able to create connections to this database.

For Java Config use the `jdbcTemplateAuthentication()` configuration method and pass it a `DataSource`. Generally this will be the Spring Boot-configured `DataSource`, which can be configured through the `spring.datasource` properties (see Recipe 7.1 for more details).

```
@Configuration
public class LibrarySecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    private DataSource dataSource;

    @Override
    protected void configure(AuthenticationManagerBuilder auth)
        throws Exception {
        auth.jdbcAuthentication().dataSource(dataSource);
    }
}
```

However, in some cases you may already have your own user repository defined in a legacy database. For example, suppose that the tables are created with the following SQL statements, and that all users in the `MEMBER` table have the enabled status:

```
CREATE TABLE MEMBER (
    ID          BIGINT      NOT NULL,
    USERNAME   VARCHAR(50) NOT NULL,
    PASSWORD   VARCHAR(32) NOT NULL,
    PRIMARY KEY (ID)
);

CREATE TABLE MEMBER_ROLE (
    MEMBER_ID   BIGINT      NOT NULL,
    ROLE        VARCHAR(10) NOT NULL,
    FOREIGN KEY (MEMBER_ID) REFERENCES MEMBER
);
```

Suppose you have the legacy user data stored in these tables as shown in Tables 6-5 and 6-6.

Table 6-5. Legacy User Data in the MEMBER Table

ID	USERNAME	PASSWORD
1	admin@ya2do.io	{noop}secret
2	marten@ya2do.io	{noop}user

Table 6-6. Legacy User Data in the MEMBER_ROLE Table

MEMBER_ID	ROLE
1	ROLE_ADMIN
1	ROLE_USER
2	ROLE_USER

Fortunately, Spring Security also supports using custom SQL statements to query a legacy database for user details. You can specify the statements for querying a user's information and authorities using `usersByUsernameQuery()` and `authoritiesByUsernameQuery()` configuration methods.

```
@Configuration
public class LibrarySecurityConfig extends WebSecurityConfigurerAdapter {

    ...

    @Override
    protected void configure(AuthenticationManagerBuilder auth)
        throws Exception {
        auth.jdbcAuthentication()
            .dataSource(dataSource)
            .usersByUsernameQuery(
                "SELECT username, password, 'true' as enabled " +
                "FROM member WHERE username = ?")
    }
}
```

```

    .authoritiesByUsernameQuery(
        "SELECT member.username, member_role.role as authorities " +
        "FROM member, member_role " +
        "WHERE member.username = ? AND member.id = member_role.member_id");
    }
}

```

Encrypting Passwords

Until now, you have been storing user details with clear-text passwords. But this approach is vulnerable to hacker attacks, so you should encrypt the passwords before storing them. Spring Security supports several algorithms for encrypting passwords. For example, you can choose BCrypt, a one-way hash algorithm, to encrypt your passwords.

Note You may need a helper to calculate BCrypt hashes for your passwords. You can do this online through, for example, www.browserling.com/tools/bcrypt, or you can simply create a class with a main method that uses Spring Security's `BCryptPasswordEncoder`.

Of course, you have to store the encrypted passwords in the database tables instead of the clear-text passwords, as shown in Table 6-7. To store BCrypt hashes in the password field, the length of the field has to be at least 68 chars long (that is the length of the BCrypt hash + the encryption type `{bcrypt}`).

Table 6-7. Testing User Data with Encrypted Passwords for the USERS Table

USERNAME	PASSWORD	ENABLED
admin@ya2do.io	{bcrypt}\$2a\$10\$E3mPTZb50e7sSW15fDx8Ne7hDZpfDjrmMPTTUp8wVjLTu.G5oPYCO	1
marten@ya2do.io	{bcrypt}\$2a\$10\$5VWqjwoMYnFRTTmbWCRZTiY3WW8ny27kQuUL9yPK1/WJcPcBLFW0	1
jdoe@does.net	{bcrypt}\$2a\$10\$cFKh0.XCUOA9L.in5smli02QIOT8.6ufQSwiC.AVz26WctxhSWC6	0

6-4 Making Access Control Decisions

Problem

In the authentication process, an application will grant a successfully authenticated user a set of authorities. When this user attempts to access a resource in the application, the application has to decide whether the resource is accessible with the granted authorities or other characteristics.

Solution

The decision on whether a user is allowed to access a resource in an application is called an access control decision. It is made based on the user's authentication status, and the resource's nature and access attributes.

How It Works

With Spring Security it is possible to use Spring Expression Language (SpEL) to create powerful access control rules. Spring Security supports a couple of expressions out of the box (see Table 6-8 for a list). Using constructs like and, or, and not one can create very powerful and flexible expressions.

Table 6-8. Spring Security Built-in Expressions

Expression	Description
hasRole('role') or hasAuthority('authority')	Returns true if the current user has the given role
hasAnyRole('role1','role2') / hasAnyAuthority('auth1','auth2')	Returns true if the current user has at least one of the given roles
hasIpAddress('ip-address')	Returns true if the current user has the given ip-address
principal	The current user
Authentication	Access to the Spring Security authentication object
permitAll	Always evaluates to true

(continued)

Table 6-8. (continued)

Expression	Description
denyAll	Always evaluates to false
isAnonymous()	Returns true if the current user is anonymous
isRememberMe()	Returns true if the current user logged in by the means of remember-me functionality
isAuthenticated()	Returns true if this is not an anonymous user
isFullyAuthenticated()	Returns true if the user is not an anonymous or a remember-me user

Caution Although role and authority are almost the same, there is a slight, but important, difference in how they are processed. When using hasRole the passed in value for the role will be checked if it starts with ROLE_ (the default role prefix); if not, this will be added before checking the authority. So hasRole('ADMIN') will actually check if the current user has the authority ROLE_ADMIN. When using hasAuthority it will check the value as is.

The following expression would give access to deletion of a book if someone had the ADMIN role or was logged in on the local machine. Writing such an expression can be done through the access method instead of one of the has* methods when defining a matcher.

```

@Configuration
public class LibrarySecurityConfig extends WebSecurityConfigurerAdapter {

    ...
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests()
            .antMatchers(HttpMethod.GET, "/books*").hasAnyRole("USER", "GUEST")
            .antMatchers(HttpMethod.POST, "/books*").hasRole("USER")
    }
}

```

```

    .antMatchers(HttpMethod.DELETE, "/books*")
        .access("hasRole('ROLE_ADMIN') or hasIpAddress('127.0.0.1') " +
            "or hasIpAddress('0:0:0:0:0:0:0:1')")
        ...
    }
}

```

Using Expression to Make Access Control Decisions Using Spring Beans

Using the @ syntax in the expression, you can call any bean in the application context. So you could write an expression like @accessChecker.hasLocalAccess (authentication) and provide a bean named accessChecker, which has a hasLocalAccess method that takes an Authentication object.

```

package com.apress.springbootrecipes.library.security;

import org.springframework.security.core.Authentication;
import org.springframework.security.web.authentication.
WebAuthenticationDetails;

@Component
public class AccessChecker {

    public boolean hasLocalAccess(Authentication authentication) {
        boolean access = false;
        if (authentication.getDetails() instanceof
WebAuthenticationDetails) {
            WebAuthenticationDetails details =
                (WebAuthenticationDetails) authentication.getDetails();
            String address = details.getRemoteAddress();
            access = address.equals("127.0.0.1") ||
                address.equals("0:0:0:0:0:0:0:1");
        }
        return access;
    }
}

```

The AccessChecker does the same checks as the earlier written custom expression handler but doesn't extend the Spring Security classes.

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
        .antMatchers(HttpMethod.POST, "/books*").hasAuthority("USER")
        .antMatchers(HttpMethod.DELETE, "/books*")
        .access("hasAuthority('ADMIN') " +
            "or @accessChecker.hasLocalAccess(authentication));
    ...
}
```

Securing Methods with Annotations and Expressions

You can use the `@PreAuthorize` and `@PostAuthorize` annotations to secure method invocations instead of only securing URLs. The `@PreAuthorize` is checked before the method is invoked and `@PostAuthorize` after the method invocation and could be used to do security checks on the returned value. With those you can write security-based expressions just as with the URL-based security. To enable the annotation processing, add the `@EnableGlobalMethodSecurity` annotation to the security configuration and set the `prePostEnabled` attribute to `true`.

```
@Configuration
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class LibrarySecurityConfig extends WebSecurityConfigurerAdapter {
    ...
}
```

Now you can use the `@PreAuthorize` annotation to secure your application.

```
package com.apress.springbootrecipes.library;

import org.springframework.security.access.prepost.PreAuthorize;
import org.springframework.stereotype.Service;

import java.util.Map;
import java.util.Optional;
import java.util.concurrent.ConcurrentHashMap;
```

```
@Service
class InMemoryBookService implements BookService {

    private final Map<String, Book> books = new ConcurrentHashMap<>();

    @Override
    @PreAuthorize("isAuthenticated()")
    public Iterable<Book> findAll() {
        return books.values();
    }

    @Override
    @PreAuthorize("hasAuthority('USER')")
    public Book create(Book book) {
        books.put(book.getIsbn(), book);
        return book;
    }

    @Override
    @PreAuthorize("hasAuthority('ADMIN') or @accessChecker.hasLocalAccess(authentication)")
    public void remove(Book book) {
        books.remove(book.getIsbn());
    }

    @Override
    @PreAuthorize("isAuthenticated()")
    public Optional<Book> find(String isbn) {
        return Optional.ofNullable(books.get(isbn));
    }
}
```

The `@PreAuthorize` annotation will trigger Spring Security to validate expression in there. If it is successful, then access is granted, else an exception will be thrown and indicated to the user that he doesn't have access.

6-5 Adding Security to a WebFlux Application Problem

You have an application built with Spring Web Flux (see Chapter 5) and you want to secure it using Spring Security.

Solution

When adding Spring Security as a dependency to a WebFlux-based application, Spring Boot will automatically enable security. It will add an `@EnableWebFluxSecurity` annotated configuration class to the application. The `@EnableWebFluxSecurity` annotation then imports the default Spring Security `WebFluxSecurityConfiguration`.

How It Works

A Spring WebFlux application is very different in nature than a regular Spring MVC application; nonetheless, Spring Boot and Spring Security strive to make it easier to build secure WebFlux-based applications.

To get security enabled, add the `spring-boot-starter-security` to your WebFlux application (from Recipe 5.3).

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

This will add the `spring-security-core`, `spring-security-config`, and `spring-security-web` dependencies to your project. Spring Boot detects the availability of some of the classes inside these JAR files, and with that will automatically enable security.

Spring Boot will configure Spring Security with the following: authentication with Basic Authentication and form login' enable HTTP headers for security; and require a login to access any resource.

Warning Spring Boot will add a default user, named `user`, with a generated password, visible in the startup logs. This is only intended to be used for testing, prototyping, or demos; do not use the generated user in a live system!.

When adding the dependency `spring-boot-starter-security` to Recipe 5.3, it will automatically secure all exposed endpoints. At startup the generated password will be logged (Figure 6-4).

Figure 6-4. Secure WebFlux output

Now when trying to access `http://localhost:8080/` a login page will be shown (Figure 6-5).



Figure 6-5. Default login page

Securing URL Access

Access rules can be configured by adding a custom `SecurityWebFilterChain`. First let's create an `OrdersSecurityConfiguration`.

```
@Configuration
public class OrdersSecurityConfiguration { ... }
```

The `WebFluxSecurityConfiguration` class from Spring Security detects instances of `SecurityWebFilterChain` (containing the security configuration), which is wrapped as a `WebFilter` which in turn is used by `WebFlux` to add behavior to incoming request (just as a normal Servlet Filter).

Currently the configuration only enables security; let's add some security rules.

```
@Bean
SecurityWebFilterChain springWebFilterChain(ServerHttpSecurity http) throws
Exception {
    return http
        .authorizeExchange()
        .pathMatchers("/").permitAll()
        .pathMatchers("/orders*").hasRole("USER")
        .anyExchange().authenticated()
        .and().build();
}
```

The `ServerHttpSecurity` should look familiar (see other recipes in this chapter) and is used to add security rules and do further configuration (like adding/removing headers and configure the login method). With the `authorizeExchange` it is possible to write rules, where with secure URLs the `/` is permitted for everyone and the `/orders` URLs are only available for the role `USER`. For other requests you have at least to be authenticated. Finally you need to call `build()` to actually build the `SecurityWebFilterChain`.

Next to the `authorizeExchange` it is also possible to use the `headers()` configuration method to add security headers to requests, `csrf()` for adding CSRF protection, etc.

Logging in to Web Flux Applications

You could override parts of the default configuration by explicitly configuring them; you could override the authentication manager used and the repository used to store the security context. The authentication manager is detected automatically; you just need to register a bean of type `ReactiveAuthenticationManager` or of `UserDetailsRepository`.

You can also configure the location where the `SecurityContext` is stored by configuring the `ServerSecurityContextRepository`. The default implementation used is the `WebSessionServerSecurityContextRepository`, which stores the context in the `WebSession`. The other default implementation is the `NoOpServerSecurityContextRepository`, which is used in stateless applications.

```
@Bean
SecurityWebFilterChain springWebFilterChain(HttpSecurity http) throws
Exception {
    return http
        .httpBasic()
        .and().formLogin()
        .authenticationManager(new CustomReactiveAuthenticationManager())
        .securityContextRepository(
            new ServerWebExchangeAttributeSecurityContextRepository())
        .and().build();
}
```

This would override the defaults with a `CustomReactiveAuthenticationManager` and the stateless `NoOpServerSecurityContextRepository`. However, for our application we are going to stick with the defaults.

Authenticating Users

Authenticating users in a Spring WebFlux-based application is done through a `ReactiveAuthenticationManager`; this is an interface with a single `authenticate` method. You can either provide your own implementation or use one of the two provided implementations. The first is the `UserDetailsRepositoryAuthenticationManager`, which wraps an instance of `ReactiveUserDetailsService`.

Note The `ReactiveUserDetailsService` has only a single implementation, the `MapReactiveUserDetailsService`, which is an in-memory implementation. You could provide your own implementation based on a Reactive datastore (like MongoDB or Couchbase).

The other implementation, the `ReactiveAuthenticationAdapter`, is actually a wrapper for a regular `AuthenticationManager`. It will wrap a regular instance and because of that you can use the blocking implementations in a reactive way. This doesn't make them reactive: they still block, but they are reusable this way. With this you could use JDBC, LDAP, etc. also for your reactive application.

When configuring Spring Security in a Spring WebFlux application, you can either add an instance of a `ReactiveAuthenticationManager` to your Java configuration class or a `UserDetailsRepository`. When the latter is detected, it will automatically be wrapped in a `UserDetailsRepositoryAuthenticationManager`.

```
@Bean
```

```
public MapUserDetailsRepository userDetailsService() {
    UserDetails marten =
        User.withUsername("marten").password("secret")
            .roles("USER").build();
    UserDetails admin =
        User.withUsername("admin").password("admin")
            .roles("USER", "ADMIN").build();
    return new MapUserDetailsRepository(marten, admin);
}
```

When you now run the application you would be free to access the / page but when accessing a URL starting with /orders you would be greeted by a login form (see Figure 6-5). When entering the credentials of one of the predefined users, you should be allowed access to the requested URL.

Making Access Control Decisions

At some point in your application you need to grant access to a user, based on the authorities or roles the user has. Spring Security provides some built-in expressions (Table 6-9) to handle that.

Table 6-9. Spring Security WebFlux Built-in Expressions

Expression	Description
hasRole('role') or hasAuthority('authority')	Returns true if the current user has the given role
permitAll()	Always evaluates to true
denyAll()	Always evaluates to false
authenticated()	Returns true if the user is authenticated
access()	Use a function to determine if access is granted

Caution Although role and authority are almost the same, there is a slight, but important, difference in how they are processed. When using hasRole the passed in value for the role will be checked if it starts with ROLE (the default role prefix); if not, this will be added before checking the authority. So hasRole('ADMIN') will actually check if the current user has the authority ROLE_ADMIN. When using hasAuthority it will check the value as is.

```
@Bean
SecurityWebFilterChain springWebFilterChain(HttpSecurity http) throws
Exception {
    return http
        .authorizeExchange()
        .pathMatchers("/").permitAll()
        .pathMatchers("/orders*").access(this::ordersAllowed)
        .anyExchange().authenticated()
        .and()
        .build();
}
```

```
private Mono<AuthorizationDecision> ordersAllowed(Mono<Authentication>
authentication, AuthorizationContext context) {
    return authentication
        .map( a.getAuthorities()
            .contains(new SimpleGrantedAuthority("ROLE_ADMIN")))
        .map( AuthorizationDecision::new);
}
```

The `access()` expression can be used to write a very powerful expression. The preceding snippet allows access if the current user has the `ROLE_ADMIN` authority. The `Authentication` contains the collection of `GrantedAuthorities`, which you can check for the `ROLE_ADMIN`. Of course you can write as many complex expressions as you like: you could check for the ip-address, request headers, etc.

6-6 Summary

In this chapter you learned how to secure Spring Boot applications with Spring Security. It can be used to secure any Java application, but it's mostly used for web-based applications. The concepts of authentication, authorization, and access control are essential in the security area, so you should have a clear understanding of them.

You often have to secure critical URLs by preventing unauthorized access to them. Spring Security can help you to achieve this in a declarative way. It handles security by applying servlet filters, which can be configured with simple Java-based configuration. Spring Security will automatically configure the basic security services for you and tries to be as secure as possible by default.

Spring Security supports multiple ways for users to log into a web application, such as form-based login and HTTP Basic authentication. It also provides an anonymous login service that allows you to handle an anonymous user just like a normal user. Remember-me support allows an application to remember a user's identity across multiple browser sessions.

Spring Security supports multiple ways of authenticating users and has built-in provider implementations for them. For example, it supports authenticating users against in-memory definitions, a relational database, and an LDAP repository. You should always store encrypted passwords in your user repository, because clear-text passwords are vulnerable to hacker attacks. Spring Security also supports caching user details locally to save you the overhead of performing remote queries.

Decisions on whether a user is allowed to access a given resource are made by access decision managers. Spring Security comes with three access decision managers that are based on the voting approach. All of them require a group of voters to be configured for voting on access control decisions.

Spring Security enables you to secure method invocations in a declarative way using the `@PreAuthorize` and `@PostAuthorize` annotations.

Spring Security also has support for securing Spring WebFlux-based applications. In the last recipe, you explored how you could add security to such an application.

CHAPTER 7

Data Access

When working with a database the first thing you need is a connection to the database. A connection in Java is obtained through a `javax.sql.DataSource`. Spring provides out-of-the-box several implementations of a `DataSource` like the `DriverManagerDataSource` and `SimpleDriverDataSource`. However, these implementations aren't connections pool and should be considered mainly for testing but not production use. For a live system, you want to use a proper connection pool like `HikariCP`.¹

Tip In the db folder there is a Dockerfile that will build a PostgreSQL² with the database automatically created. Build it using `docker build -t sb2r-postgres`. You can then run it using `docker run -p 5432:5432 -it sb2r-postgres`.

7-1 Configuring a `DataSource`

Problem

You need access to a database from your application.

Solution

Use the `spring.datasource.url`, `spring.datasource.username`, and `spring.datasource.password` properties to let Spring Boot configure a `DataSource`.

¹<https://brettwooldridge.github.io/HikariCP/>

²<https://www.postgresql.org>

How It Works

To configure a DataSource, Spring Boot requires the presence of a connection pool or an embedded database like H2, HSQLDB, or Derby. Spring Boot automatically detects HikariCP, Tomcat JDBC, and Commons DBCP2-based connection pools (in that order). To use the connection pool, configure the `spring.datasource.url`, `spring.datasource.username`, and `spring.datasource.password` properties. The JDBC support can be enabled by adding the `spring-boot-starter-jdbc` dependency. This will pull in all required dependencies to work with JDBC.

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

Dependencies included are `spring-jdbc`, `spring-tx`, and HikariCP as the default connection pool.

Use an Embedded DataSource

When Spring Boot detects the presence of H2, HSQLDB, or Derby it will by default start an embedded database using the detected embedded implementation. This is very useful when writing a test or preparing a demonstration for a business. Letting Spring Boot create this is a simple matter of including the desired dependency.

```
<dependency>
  <groupId>org.apache.derby</groupId>
  <artifactId>derby</artifactId>
  <scope>runtime</scope>
</dependency>
```

Spring Boot will now detect Derby and bootstrap an embedded DataSource. Let's write an application that lists the tables in the database.

```
package com.apress.springboot2recipes.jdbc;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.ApplicationArguments;
```

```

import org.springframework.boot.ApplicationRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.stereotype.Component;

import javax.sql.DataSource;

@SpringBootApplication
public class JdbcApplication {

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

@Component
class TableLister implements ApplicationRunner {

    private final Logger logger = LoggerFactory.getLogger(getClass());
    private final DataSource dataSource;

    TableLister(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    @Override
    public void run(ApplicationArguments args) throws Exception {
        try (var con = dataSource.getConnection();
            var rs = con.getMetaData().getTables(null, null, "%", null)) {
            while (rs.next()) {
                logger.info("{} {}", rs.getString(3));
            }
        }
    }
}

```

When the application runs, it will create an instance of the TableLister and the instance will receive the configured DataSource. Spring Boot will then detect the fact it is an ApplicationRunner and will call the run method. The run method obtains a

Connection from the `DataSource` and uses the `DatabaseMetaData` (from JDBC) to obtain the tables in the database. Now when running the application it should display output similar to that as in Figure 7-1.

```

2018-09-10 19:27:18.543  WARN 96243 --- [           main] com.zaxxer.hikari.util.DriverDataSource : Registered driver with driverClass
2018-09-10 19:27:18.946  INFO 96243 --- [           main] com.zaxxer.hikari.pool.PoolBase      : HikariPool-1 - Driver does not support auto-commit mode.
2018-09-10 19:27:18.949  INFO 96243 --- [           main] com.zaxxer.hikari.HikariDataSource   : HikariPool-1 - Start completed.
2018-09-10 19:27:19.192  INFO 96243 --- [           main] c.a.springboot2recipes.jdbc.TableLister : SYSALIASES
2018-09-10 19:27:19.192  INFO 96243 --- [           main] c.a.springboot2recipes.jdbc.TableLister : SYSCHECKS
2018-09-10 19:27:19.192  INFO 96243 --- [           main] c.a.springboot2recipes.jdbc.TableLister : SYSCOLPERMS
2018-09-10 19:27:19.193  INFO 96243 --- [           main] c.a.springboot2recipes.jdbc.TableLister : SYSCOLUMNS
2018-09-10 19:27:19.193  INFO 96243 --- [           main] c.a.springboot2recipes.jdbc.TableLister : SYSCONGLOMERATES
2018-09-10 19:27:19.193  INFO 96243 --- [           main] c.a.springboot2recipes.jdbc.TableLister : SYSCONSTRAINTS
2018-09-10 19:27:19.193  INFO 96243 --- [           main] c.a.springboot2recipes.jdbc.TableLister : SYSDEPENDS
2018-09-10 19:27:19.193  INFO 96243 --- [           main] c.a.springboot2recipes.jdbc.TableLister : SYSFILES
2018-09-10 19:27:19.193  INFO 96243 --- [           main] c.a.springboot2recipes.jdbc.TableLister : SYSFOREIGNKEYS
2018-09-10 19:27:19.196  INFO 96243 --- [           main] c.a.springboot2recipes.jdbc.TableLister : SYSKEYS
2018-09-10 19:27:19.197  INFO 96243 --- [           main] c.a.springboot2recipes.jdbc.TableLister : SYSPERMS
2018-09-10 19:27:19.197  INFO 96243 --- [           main] c.a.springboot2recipes.jdbc.TableLister : SYSROLES
2018-09-10 19:27:19.197  INFO 96243 --- [           main] c.a.springboot2recipes.jdbc.TableLister : SYSROUTINEPERMS
2018-09-10 19:27:19.197  INFO 96243 --- [           main] c.a.springboot2recipes.jdbc.TableLister : SYSSCHEMAS
2018-09-10 19:27:19.197  INFO 96243 --- [           main] c.a.springboot2recipes.jdbc.TableLister : SYSSEQUENCES
2018-09-10 19:27:19.197  INFO 96243 --- [           main] c.a.springboot2recipes.jdbc.TableLister : SYSSTATEMENTS
2018-09-10 19:27:19.197  INFO 96243 --- [           main] c.a.springboot2recipes.jdbc.TableLister : SYSSTATISTICS
2018-09-10 19:27:19.197  INFO 96243 --- [           main] c.a.springboot2recipes.jdbc.TableLister : SYSTABLEPERMS
2018-09-10 19:27:19.197  INFO 96243 --- [           main] c.a.springboot2recipes.jdbc.TableLister : SYSTABLES
2018-09-10 19:27:19.197  INFO 96243 --- [           main] c.a.springboot2recipes.jdbc.TableLister : SYSTRIGGERS
2018-09-10 19:27:19.198  INFO 96243 --- [           main] c.a.springboot2recipes.jdbc.TableLister : SYSUSERS
2018-09-10 19:27:19.198  INFO 96243 --- [           main] c.a.springboot2recipes.jdbc.TableLister : SYSVIEWS
2018-09-10 19:27:19.198  INFO 96243 --- [           main] c.a.springboot2recipes.jdbc.TableLister : SYSUMMY1

```

Figure 7-1. TableLister output for Derby

Using an External Database

To connect to a database, you need a JDBC driver; this recipe uses PostgreSQL, so you need to include the driver for this database.

```

<dependency>
  <groupId>org.postgresql</groupId>
  <artifactId>postgresql</artifactId>
</dependency>

```

Now configuring a single datasource is a matter of including the properties in the `application.properties`. For the dockerized PostgreSQL database shipped with this recipe, the datasource configuration is shown as follows; replace with our own configuration as needed.

```

spring.datasource.url=jdbc:postgresql://localhost:5432/customers
spring.datasource.username=customers
spring.datasource.password=customers

```

The `spring.datasource.url` tells the datasource where to connect to, and `spring.datasource.username` and `spring.datasource.password` configure the username and password to use when connecting. You could also specify the `spring.datasource.driver-class-name` to specify the JDBC driver class to use; generally Spring Boot will detect the driver to use from the passed in URL. If you want to use a nondefault driver (for performance or logging), you could specify this as well.

When running the `JdbcApplication` the output should be like that of Figure 7-2; there are now a bunch of different tables (compared with Derby).

```
2018-09-10 19:30:15.711 INFO 97421 --- [           main] com.zaxxer.hikari.HikariDataSource      : HikariPool-1 - Starting...
2018-09-10 19:30:15.996 INFO 97421 --- [           main] com.zaxxer.hikari.HikariDataSource      : HikariPool-1 - Start completed.
2018-09-10 19:30:16.012 INFO 97421 --- [           main] c.a.springboot2recipes.jdbc.TableLister   : pg_aggregate_foid_index
2018-09-10 19:30:16.012 INFO 97421 --- [           main] c.a.springboot2recipes.jdbc.TableLister   : pg_am_name_index
2018-09-10 19:30:16.012 INFO 97421 --- [           main] c.a.springboot2recipes.jdbc.TableLister   : pg_am_oid_index
2018-09-10 19:30:16.012 INFO 97421 --- [           main] c.a.springboot2recipes.jdbc.TableLister   : pg_amop_fam_strat_index
2018-09-10 19:30:16.012 INFO 97421 --- [           main] c.a.springboot2recipes.jdbc.TableLister   : pg_amop_oid_index
2018-09-10 19:30:16.012 INFO 97421 --- [           main] c.a.springboot2recipes.jdbc.TableLister   : pg_amop_opr_fam_index
2018-09-10 19:30:16.013 INFO 97421 --- [           main] c.a.springboot2recipes.jdbc.TableLister   : pg_amproc_fam_proc_index
2018-09-10 19:30:16.013 INFO 97421 --- [           main] c.a.springboot2recipes.jdbc.TableLister   : pg_amproc_oid_index
2018-09-10 19:30:16.013 INFO 97421 --- [           main] c.a.springboot2recipes.jdbc.TableLister   : pg_attrdef_adrelid_dnum_index
2018-09-10 19:30:16.013 INFO 97421 --- [           main] c.a.springboot2recipes.jdbc.TableLister   : pg_attrdef_oid_index
2018-09-10 19:30:16.013 INFO 97421 --- [           main] c.a.springboot2recipes.jdbc.TableLister   : pg_attribute_relid_attnam_index
2018-09-10 19:30:16.013 INFO 97421 --- [           main] c.a.springboot2recipes.jdbc.TableLister   : pg_attribute_relid_attnum_index
2018-09-10 19:30:16.013 INFO 97421 --- [           main] c.a.springboot2recipes.jdbc.TableLister   : pg_auth_members_member_role_index
2018-09-10 19:30:16.013 INFO 97421 --- [           main] c.a.springboot2recipes.jdbc.TableLister   : pg_auth_members_role_member_index
2018-09-10 19:30:16.013 INFO 97421 --- [           main] c.a.springboot2recipes.jdbc.TableLister   : pg_authid_oid_index
2018-09-10 19:30:16.013 INFO 97421 --- [           main] c.a.springboot2recipes.jdbc.TableLister   : pg_authid_relname_index
2018-09-10 19:30:16.013 INFO 97421 --- [           main] c.a.springboot2recipes.jdbc.TableLister   : pg_cast_oid_index
2018-09-10 19:30:16.013 INFO 97421 --- [           main] c.a.springboot2recipes.jdbc.TableLister   : pg_cast_source_target_index
2018-09-10 19:30:16.013 INFO 97421 --- [           main] c.a.springboot2recipes.jdbc.TableLister   : pg_class_oid_index
2018-09-10 19:30:16.014 INFO 97421 --- [           main] c.a.springboot2recipes.jdbc.TableLister   : pg_class_rename_nsp_index
2018-09-10 19:30:16.014 INFO 97421 --- [           main] c.a.springboot2recipes.jdbc.TableLister   : pg_class_tblspc_relfilenode_index
2018-09-10 19:30:16.014 INFO 97421 --- [           main] c.a.springboot2recipes.jdbc.TableLister   : pg_collation_name_enc_nsp_index
2018-09-10 19:30:16.014 INFO 97421 --- [           main] c.a.springboot2recipes.jdbc.TableLister   : pg_collation_oid_index
```

Figure 7-2. TableLister output for PostgreSQL

Obtaining a DataSource from JNDI

If you are deploying your Spring Boot application to an application server (or if you have a remote JNDI server) and want to use a preconfigured `DataSource`, you can use the `spring.datasource.jndi-name` property to let Spring Boot know you want to obtain a `DataSource` from JNDI.

```
spring.datasource.jndi-name=java:jdbc/customers
```

Configure the Connection Pool

The default connection pool used by Spring Boot is HikariCP. This comes automatically when including the `spring-boot-starter-jdbc` dependency (or one of the other database-related dependencies). Spring Boot will configure the connection pool with some default settings; however, you might want to override this (increase the max

connections or reduce it, set the timeouts, etc.). The configuration options for HikariCP are in the `spring.datasource.hikari` namespace (Table 7-1).

Table 7-1. HikariCP Common Connection Pool Settings

Property	Description
<code>spring.datasource.hikari.connection-timeout</code>	Maximum number of milliseconds that a client will wait for a connection from the pool. Default 30 seconds
<code>spring.datasource.hikari.leak-detection-threshold</code>	Amount of milliseconds that a connection can be out of the pool before a message is logged indicating a possible connection leak. Default 0 (effectively disabled)
<code>spring.datasource.hikari.idle-timeout</code>	Maximum number of milliseconds that a connection is allowed to sit idle in the pool. Default 10 minutes
<code>spring.datasource.hikari.validation-timeout</code>	Maximum number of milliseconds that the pool will wait for a connection to be validated as alive. Default 5 seconds
<code>spring.datasource.hikari.connection-test-query</code>	SQL query to be executed to test the validity of connections. NOTE: Generally not needed with JDBC 4.0 (or higher) drivers! Default is none.
<code>spring.datasource.hikari.maximum-pool-size</code>	Maximum number of connections that will be kept in the pool. Default 10 connections
<code>spring.datasource.hikari.minimum-idle</code>	Minimum number of idle connections that are maintained in the pool. Default 10 connections

There are more properties you can use, but the list is quite long and the properties mentioned in Table 7-1 are the most commonly used properties.

Note The properties for Tomcat JDBC are in the `spring.datasource.tomcat` namespace and for Commons DBCP2 in the `spring.datasource.dbcp2` namespace.

To configure the datasource with a maximum of five connections, minimum of two, and leak detection with a threshold of 20 seconds, the following configuration needs to be added.

```
spring.datasource.hikari.maximum-pool-size=5
spring.datasource.hikari.minimum-idle=2
spring.datasource.hikari.leak-detection-threshold=20000
```

Initializing the Database with Spring Boot

When working with an existing database, you probably already have existing tables, views, and procedures. However, when you create a new database it is empty and you will need to create the tables yourself. Using Spring Boot, this is supported out-of-the-box. You can add a `schema.sql` to initialize the schema (tables, views, etc.) and a `data.sql` to insert data into the tables. Spring Boot will also allow you to provide a `schema-<db-platform>.sql` and `data-<db-platform>.sql` to do database-specific initialization. The value for `<db-platform>` is read from the `spring.datasource.platform` property (see also Table 7-2). When using Derby, you could add a `schema-derby.sql`, etc. The name of the schema and data files can be changed through the `spring.datasource.schema` and `spring.datasource.data` properties. See Table 7-2 for a description of available properties.

Table 7-2. *DataSource Initialize Properties*

Property	Description
<code>spring.datasource.continue-on-error</code>	Whether to stop if an error occurs while initializing the database, default <code>false</code>
<code>spring.datasource.data</code>	Data (DML) script resource references, default <code>classpath:data</code>
<code>spring.datasource.data-password</code>	Password of the database to execute DML scripts, defaults to the normal password
<code>spring.datasource.data-username</code>	Username of the database to execute DML scripts, defaults to the normal username

(continued)

Table 7-2. (continued)

Property	Description
spring.datasource.initialization-mode	Initialize the datasource with available DDL and DML scripts. Default EMBEDDED, can be changed to NEVER or ALWAYS
spring.datasource.platform	Platform to use in the DDL or DML scripts (such as schema-\${platform}.sql or data-\${platform}.sql). Defaults is all.
spring.datasource.schema	Data (DDL) script resource references, default classpath:schema
spring.datasource.schema-password	Password of the database to execute DDL scripts, defaults to the normal password.
spring.datasource.schema-username	Username of the database to execute DDL scripts, defaults to the normal username
spring.datasource.separator	Statement separator in SQL initialization scripts. Default ;
spring.datasource.sql-script-encoding	SQL scripts encoding. Defaults to platform encoding

Let's create a table `customer` and insert some data into this. To create the table, add the following `schema.sql` to the `src/main/resources` directory:

```
DROP TABLE IF EXISTS customer;

CREATE TABLE customer (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    email VARCHAR(255) NOT NULL,
    UNIQUE(name)
);
```

To insert the data, add the `data.sql` to the `src/main/resources` directory.

```
INSERT INTO customer (name, email) VALUES
('Marten Deinum', 'marten.deinum@conspect.nl'),
('Josh Long', 'jlong@pivotal.com'),
('John Doe', 'john.doe@island.io'),
('Jane Doe', 'jane.doe@island.io');
```

To see if this is working, let's add another `ApplicationRunner` that prints the content of the `customer` table using the `DataSource`.

```
package com.apress.springboot2recipes.jdbc;  
  
// Imports removed  
  
@SpringBootApplication  
public class JdbcApplication {  
  
    public static void main(String[] args) {  
        SpringApplication.run(JdbcApplication.class, args);  
    }  
}  
  
... // Other code removed  
@Component  
class CustomerLister implements ApplicationRunner {  
  
    private final Logger logger = LoggerFactory.getLogger(getClass());  
    private final DataSource dataSource;  
  
    CustomerLister(DataSource dataSource) {  
        this.dataSource = dataSource;  
    }  
  
    @Override  
    public void run(ApplicationArguments args) throws Exception {  
        var query = "SELECT id, name, email FROM customer";  
        try (var con = dataSource.getConnection();  
             var stmt = con.createStatement();  
             var rs = stmt.executeQuery(query)) {  
            while (rs.next()) {  
                logger.info("Customer [id={}, name={}, email={}]",  
                           rs.getLong(1), rs.getString(2), rs.getString(3));  
            }  
        }  
    }  
}
```

Database initialization is always on for embedded databases, so when using Derby, H2, or HSQLDB, this is default on. When using an external database, the initialization doesn't happen by default. To change this you can toggle the `spring.datasource.initialization-mode` property to `always`, so that it will always run.

```
spring.datasource.initialization-mode=always
```

When the application starts, you will now see the list of customers from the database being printed in the logs.

Initializing the Database with Flyway

When developing an application, you want more control of the database migration. Using a `schema.sql` and `data.sql` can work very well and quickly, but it eventually will be cumbersome to maintain. Spring Boot also supports Flyway³ which is, simply said, a version control for your database schema. It allows you to incrementally change/update your database schema. To use Flyway the first thing to do is to add the dependency on Flyway itself.

```
<dependency>
  <groupId>org.flywaydb</groupId>
  <artifactId>flyway-core</artifactId>
</dependency>
```

Spring Boot will detect the presence of Flyway and it will assume you want to use it to do database migrations. The migration scripts should be in the `db/migration` folder in `src/main/resources`.

```
CREATE TABLE customer (
  id SERIAL PRIMARY KEY,
  name VARCHAR(100) NOT NULL,
  email VARCHAR(255) NOT NULL,
  UNIQUE(name)
);
```

³<https://flywaydb.org>

```
INSERT INTO customer (name, email) VALUES
```

```
('Marten Deinum', 'marten.deinum@conspect.nl'),
('Josh Long', 'jlong@pivotal.com'),
('John Doe', 'john.doe@island.io'),
('Jane Doe', 'jane.doe@island.io');
```

This SQL, when put in a `V1__first.sql` in the `db/migration` folder, will be executed on startup (assuming an empty database). The naming convention by default is `V<sequence>_<name>.sql` and is used to determine what to execute. Once a script has been executed, you cannot (and shouldn't) modify the script. That will lead to Flyway preventing your application from starting. It detects changes in scripts already executed.

When running the application the customers should still be listed. You will notice an additional table in the table listing (Figure 7-3), the `flyway_schema_history`. This table contains the metadata used by Flyway to detect (and guard) database changes.

```
2018-09-10 19:57:22.714 INFO 98933 --- [           main] c.a.springboot2recipes.jdbc.TableLister : pg_user
2018-09-10 19:57:22.714 INFO 98933 --- [           main] c.a.springboot2recipes.jdbc.TableLister : pg_user_mappings
2018-09-10 19:57:22.714 INFO 98933 --- [           main] c.a.springboot2recipes.jdbc.TableLister : pg_views
2018-09-10 19:57:22.714 INFO 98933 --- [           main] c.a.springboot2recipes.jdbc.TableLister : customer
2018-09-10 19:57:22.714 INFO 98933 --- [           main] c.a.springboot2recipes.jdbc.TableLister : flyway_schema_history
2018-09-10 19:57:22.714 INFO 98933 --- [           main] c.a.springboot2recipes.jdbc.TableLister : pg_toast_12242
2018-09-10 19:57:22.714 INFO 98933 --- [           main] c.a.springboot2recipes.jdbc.TableLister : pg_toast_12247
2018-09-10 19:57:22.715 INFO 98933 --- [           main] c.a.springboot2recipes.jdbc.TableLister : pg_toast_12252
```

Figure 7-3. Table listing with Flyway

There are also several properties you can use to configure Flyway with Spring Boot. See Table 7-3 for the most commonly used properties.

Table 7-3. Commonly Used Flyway Properties

Property	Description
<code>spring.flyway.enabled</code>	Should Flyway be enabled, default true
<code>spring.flyway.locations</code>	The locations of migrations scripts, default <code>classpath:db/migration</code>
<code>spring.flyway.url</code>	JDBC URL of the database to migrate, when not set uses the default configured <code>DataSource</code>
<code>spring.flyway.user</code>	Username to use for the database if Flyway uses its own <code>DataSource</code> configuration
<code>spring.flyway.password</code>	Password to use for the database if Flyway uses its own <code>DataSource</code> configuration

7-2 Use JdbcTemplate

Problem

You want to use `JdbcTemplate` or `NamedParameterJdbcTemplate` to have a better JDBC experience.

Solution

Use the automatically configured `JdbcTemplate` or `NamedParameterJdbcTemplate` to execute the queries and handle the results.

How It Works

Spring Boot will configure a `JdbcTemplate` and `NamedParameterJdbcTemplate` by default and does so when it can detect a single candidate DataSouce. A single candidate DataSource means there is either only a single DataSource or there is one marked as primary resource using `@Primary`. As the `JdbcTemplate` is already available, you can use that to write your JDBC code. Rewriting the `CustomerLister` to use the `JdbcTemplate` instead of a plain `DataSource` will make the code easier to read and thus easier to handle.

Note The `NamedParameterJdbcTemplate` is similar to the `JdbcTemplate`, with the major advantage of being able to use named parameters in a query instead of the regular JDBC placeholders.

```
@Component
class CustomerLister implements ApplicationRunner {
    private final Logger logger = LoggerFactory.getLogger(getClass());
    private final JdbcTemplate jdbc;
    CustomerLister(JdbcTemplate jdbc) {
        this.jdbc = jdbc;
    }
}
```

```

@Override
public void run(ApplicationArguments args) {

    var query = "SELECT id, name, email FROM customer";
    jdbc.query(query, rs -> {
        logger.info("Customer [id={}, name={}, email={}]",
            rs.getLong(1), rs.getString(2), rs.getString(3));
    });
}
}

```

The `JdbcTemplate` is used to execute the query through the `query` method; this method takes a `String` and a `RowCallbackHandler`. The `JdbcTemplate` will execute the query and for each row call the `RowCallbackHandler`, which does the logging of the row. When running the application, the output is still the same but the code became cleaner.

The `JdbcTemplate` also has the, probably more familiar, `RowMapper` interface, which can be used to map a row from the `ResultSet` to a Java object. Let's create a `Customer` class and use a `RowMapper` to create `Customer` instances from the database.

```

package com.apress.springboot2recipes.jdbc;

import java.util.Objects;

public class Customer {

    private final long id;
    private final String name;
    private final String email;

    Customer(long id, String name, String email) {
        this.id = id;
        this.name = name;
        this.email = email;
    }

    public long getId() {
        return id;
    }
}

```

```
public String getName() {
    return name;
}

public String getEmail() {
    return email;
}

@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Customer customer = (Customer) o;
    return id == customer.id &&
        Objects.equals(name, customer.name) &&
        Objects.equals(email, customer.email);
}

@Override
public int hashCode() {
    return Objects.hash(id, name, email);
}

@Override
public String toString() {
    return "Customer [" +
        "id=" + id + ", name='" + name + '\'' +
        ", email='" + email + '\'' + ']';
}
}
```

Next we create a repository interface to define the contract and create a JDBC-based implementation.

```
package com.apress.springboot2recipes.jdbc;

import java.util.List;

public interface CustomerRepository {
```

```
List<Customer> findAll();
Customer findById(long id);
Customer save(Customer customer);
}
```

Next, the implementation uses the `JdbcTemplate` and a `RowMapper` to map the results to `Customer` objects.

```
package com.apress.springboot2recipes.jdbc;

import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.support.GeneratedKeyHolder;
import org.springframework.stereotype.Repository;

import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.List;

@Repository
class JdbcCustomerRepository implements CustomerRepository {

    private static final String ALL_QUERY =
        "SELECT id, name, email FROM customer";
    private static final String BY_ID_QUERY =
        "SELECT id, name, email FROM customer WHERE id=?";
    private static final String INSERT_QUERY =
        "INSERT INTO customer (name, email) VALUES (?,?)";
    private final JdbcTemplate jdbc;

    JdbcCustomerRepository(JdbcTemplate jdbc) {
        this.jdbc = jdbc;
    }

    @Override
    public List<Customer> findAll() {
        return jdbc.query(ALL_QUERY, (rs, rowNum) -> toCustomer(rs));
    }
}
```

```

@Override
public Customer findById(long id) {
    return jdbc.queryForObject(BY_ID_QUERY, (rs, rowNum) -> toCustomer(rs),
        id);
}

@Override
public Customer save(Customer customer) {
    var keyHolder = new GeneratedKeyHolder();
    jdbc.update(con -> {
        var ps = con.prepareStatement(INSERT_QUERY);
        ps.setString(1, customer.getName());
        ps.setString(2, customer.getEmail());
        return ps;
    }, keyHolder);
    return new Customer(keyHolder.getKey().longValue(),
        customer.getName(), customer.getEmail());
}

private Customer toCustomer(ResultSet rs) throws SQLException {
    var id = rs.getLong(1);
    var name = rs.getString(2);
    var email = rs.getString(3);
    return new Customer(id, name, email);
}
}

```

The `JdbcCustomerRepository` uses a `JdbcTemplate` and `RowMapper` (through a lambda expression) to convert the `ResultSet` into a `Customer`. The `CustomerListener` can now use the `CustomerRepository` to get all the `Customers` from the database and print them to the console.

```

@Component
class CustomerLister implements ApplicationRunner {

    private final Logger logger = LoggerFactory.getLogger(getClass());
    private final CustomerRepository customers;

```

```

CustomerLister(CustomerRepository customers) {
    this.customers = customers;
}

@Override
public void run(ApplicationArguments args) {
    customers.findAll()
        .forEach( customer -> logger.info("{}",
            customer));
}
}

```

As all JDBC code has been moved to the `JdbcCustomerRepository`, the class becomes very simple. It gets a `CustomerRepository` injected and, using the `findAll` method, it obtains the content of the database and for each customer prints a line.

Testing JDBC Code

When testing JDBC code one requires a database, often an embedded database like H2, Derby, or HSQLDB is used for testing. Spring Boot makes it very easy to write tests for JDBC code. JDBC-based tests can be annotated with `@JdbcTest` and Spring Boot will create a minimal application with only the JDBC-related beans like a `DataSource` and transaction manager.

Let's write a test for the `JdbcCustomerRepository` and use H2 as the embedded database. First, add H2 as a test dependency.

```

<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>test</scope>
</dependency>

```

Next create the `JdbcCustomerRepositoryTest`.

```

@RunWith(SpringRunner.class)
@JdbcTest(includeFilters =
    @ComponentScan.Filter(
        type= FilterType.REGEX,
        pattern = "com.apress.springboot2recipes.jdbc.*Repository"))

```

```
@TestPropertySource(properties = "spring.flyway.enabled=false")
public class JdbcCustomerRepositoryTest {

    @Autowired
    private JdbcCustomerRepository repository;
}
```

The `@RunWith(SpringRunner.class)` executes the test by a special JUnit runner to bootstrap the Spring Test Context framework. The `@JdbcTest` will replace the preconfigured `DataSource` with an embedded one (H2 in this case). As we also want to create instances of our repositories, we add an `includeFilters` and give it a regular expression to match our `JdbcCustomerRepository`.

Finally, there is the `@TestPropertySource(properties = "spring.flyway.enabled=false")`, which indicates that we want to disable Flyway. The application uses Flyway to manage the schema; however, those scripts are written for PostgreSQL and not H2. For testing, we want to disable Flyway and provide an H2-based `schema.sql` to create the schema.

Note This is one of the drawbacks of using a different database for testing (H2) than actually in the live system (PostgreSQL). You either need to maintain two sets of scripts for the schema or use the same database for testing as in the live system.

Create a `schema.sql` in `src/test/resources` and add the following DDL statement:

```
CREATE TABLE customer (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    email VARCHAR(255) NOT NULL,
    UNIQUE(name)
);
```

Now write a test that tests if records get inserted correctly.

```
@Test
public void insertNewCustomer() {
    assertThat(repository.findAll()).isEmpty();

    Customer customer = repository.save(new Customer(-1, "T. Testing",
    "t.testing@test123.tst"));

    assertThat(customer.getId()).isGreaterThan(-1L);
    assertThat(customer.getName()).isEqualTo("T. Testing");
    assertThat(customer.getEmail()).isEqualTo("t.testing@test123.tst");

    assertThat(repository.findById(customer.getId())).isEqualTo(customer);
}
```

This test first asserts that the database is empty—not really required but can be useful to detect if other tests pollute the database. Next a `Customer` is added to the database by calling the `save` method on the `JdbcCustomerRepository`. The resulting `Customer` is validated to have an id and if the name and email attributes are there. Finally, the `Customer` is retrieved again and compared to be the same.

Another test you could add is the `findAll` method. When inserting two records, calling `findAll` should result in two records being retrieved.

```
@Test
public void findAllCustomers() {
    assertThat(repository.findAll()).isEmpty();

    repository.save(new Customer(-1, "T. Testing1", "t.testing@test123.tst"));
    repository.save(new Customer(-1, "T. Testing2", "t.testing@test123.tst"));

    assertThat(repository.findAll()).hasSize(2);
}
```

Of course there could be more assertions, but saving the data is already verified in the other test method.

7-3 Use JPA

Problem

You want to use JPA in your Spring Boot application.

Solution

Spring Boot automatically detects the presence of Hibernate, and the needed JPA classes will use that information to configure the EntityManagerFactory.

How It Works

Spring Boot has out-of-the-box support for JPA through Hibernate.⁴ When Hibernate is detected, an EntityManagerFactory will be automatically configured using the earlier configured DataSource (see Recipe 7.1).

First you need to add hibernate-core and spring-orm as a dependency to your project. However, it is easier to add the spring-boot-starter-data-jpa dependency to your project (although this will also pull in spring-data-jpa as a dependency.)

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

This will add all the necessary dependencies to the classpath.

Using Plain JPA Repositories

For JPA to work you have to annotate the classes representing entities in your system. In your system we are going to store and retrieve the Customer from the database. We need to mark this as an entity using the @Entity annotation. A JPA entity class is required to have a default no-args constructor (although it can be package private) and a field marked with @id to mark the primary key.

⁴<https://www.hibernate.org>

```

@Entity
public class Customer {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;

    @Column(nullable = false)
    private final String name;

    @Column(nullable = false)
    private final String email;

    Customer() {
        this(null,null);
    }
    // Other code omitted
}

```

Next, create a JPA implementation of the `CustomerRepository` (see Recipe 7.2); to use JPA you have to get the `EntityManager`. This is done by declaring a field and annotating it with `@PersistenceContext`.

```

package com.apress.springboot2recipes.jpa;

import org.springframework.stereotype.Repository;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import java.util.List;

@Repository
class JpaCustomerRepository implements CustomerRepository {

    @PersistenceContext
    private EntityManager em;

    @Override
    public List<Customer> findAll() {
        var query = em.createQuery("SELECT c FROM Customer c", Customer.class);

```

```

    return query.getResultList();
}

@Override
public Customer findById(long id) {
    return em.find(Customer.class, id);
}

@Override
public Customer save(Customer customer) {
    em.persist(customer);
    return customer;
}
}
}

```

The following application class (similar to the one of Recipe 7.2) will read all Customers from the database and print them to the logging (Figure 7-4).

```

package com.apress.springboot2recipes.jpa;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.ApplicationArguments;
import org.springframework.boot.ApplicationRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.stereotype.Component;

@SpringBootApplication
public class JpaApplication {

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

@Component
class CustomerLister implements ApplicationRunner {

```

```

private final Logger logger = LoggerFactory.getLogger(getClass());
private final CustomerRepository customers;

CustomerLister(CustomerRepository customers) {
    this.customers = customers;
}

@Override
public void run(ApplicationArguments args) {

    customers.findAll()
        .forEach( customer -> logger.info("{}" , customer));
}
}

```

```

2018-09-18 20:22:00,958  INFO 99472 --- [           main] org.hibernate.type.BasicTypeRegistry : HHH000270: Type registration [java.util.UUID] overrides previous : org.h
2018-09-18 20:22:01,534  INFO 99472 --- [           main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
2018-09-18 20:22:02,029  INFO 99472 --- [           main] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
2018-09-18 20:22:02,096  INFO 99472 --- [           main] c.a.s.jpa.JpaApplication             : Started JpaApplication in 3.567 seconds (JVM running for 16.739)
2018-09-18 20:22:02,164  INFO 99472 --- [           main] o.h.h.i.QueryTranslatorFactoryInitiator : HHH000397: Using ASTQueryTranslatorFactory
2018-09-18 20:22:02,316  INFO 99472 --- [           main] c.a.s.jpa.CustomerLister            : Customer [id=1, name='Marten Deinum', email='marten.deinum@conspект.nl']
2018-09-18 20:22:02,326  INFO 99472 --- [           main] c.a.s.jpa.CustomerLister            : Customer [id=2, name='Josh Long', email='jlong@pivotall.com']
2018-09-18 20:22:02,326  INFO 99472 --- [           main] c.a.s.jpa.CustomerLister            : Customer [id=3, name='John Doe', email='john.doe@island.io']
2018-09-18 20:22:02,327  INFO 99472 --- [           main] c.a.s.jpa.CustomerLister            : Customer [id=4, name='Jane Doe', email='jane.doe@island.io']

```

Figure 7-4. JPA Customer output

There are some configuration options you can use to configure the EntityManagerFactory in your application; those properties can be found in the `spring.jpa` namespace.

Table 7-4. JPA Properties

Property	Description
<code>spring.jpa.database</code>	Target database to operate on, auto-detected by default.
<code>spring.jpa.database-platform</code>	Name of the target database to operate on, auto-detected by default. Can be used to specify a specific Hibernate Dialect to use.
<code>spring.jpa.generate-ddl</code>	Initialize the schema on startup, default is false.
<code>spring.jpa.show-sql</code>	Enable logging of SQL statements, default is false.

(continued)

Table 7-4. (continued)

Property	Description
spring.jpa.open-in-view	Register the OpenEntityManagerInViewInterceptor. Binds the EntityManager to the request processing thread. Default true.
spring.jpa.hibernate.ddl-auto	Shorthand for the hibernate.hbm2ddl.auto property. Default none and create-drop for embedded databases.
spring.jpa.hibernate.use-new-id-generator-mappings	Shorthand for the hibernate.id.new_generator_mappings property. When not explicitly set, defaults to true.
spring.jpa.hibernate.naming.implicit-strategy	FQN of the implicit naming strategy, default org.springframework.boot.orm.jpa.hibernate.SpringPhysicalNamingStrategy.
spring.jpa.hibernate.naming.physical-strategy	FQN of the physical naming strategy, default org.springframework.boot.orm.jpa.hibernate.SpringImplicitNamingStrategy.
spring.jpa.mapping-resources	Additional XML files containing entity mappings in XML instead of Java. Like the JPA specified orm.xml.
spring.jpa.properties.*	Additional properties to set on the JPA provider.

The `spring.jpa.properties` is useful if you want to configure advanced features of Hibernate like the fetch size `hibernate.jdbc.fetch_size` or batch size `hibernate.jdbc.batch_size` when doing batch processing.

```
spring.jpa.properties.hibernate.jdbc.fetch_size=250
spring.jpa.properties.hibernate.jdbc.batch_size=50
```

This will set the properties on the JPA provider.

Use Spring Data JPA Repositories

Instead of writing your own repositories, which can be a tedious and repetitive task, you can also let Spring Data JPA⁵ do the heavy lifting for you. Instead of writing your own implementation, you can extend the `CrudRepository` interface from Spring Data and have a repository available at runtime. This saves you from writing the data access code. Spring Boot will also autoconfigure Spring Data JPA when it detects it on the classpath.

```
public interface CustomerRepository extends CrudRepository<Customer, Long> { }
```

That is all you need; the `findAll`, `findById`, and `save` methods, among others, are provided out-of-the-box by Spring Data JPA. You can remove the `JpaCustomerRepository` implementation. Due to the `CrudRepository<Customer, Long>` Spring Data knows that it can query for `Customer` instances and that it has a `Long` as an id field.

Running the application should result in the same output as before (see Figure 7-4).

There is only a single property available for Spring Data JPA and that is to enable or disable it explicitly. It is by default enabled. Setting the `spring.data.jpa.repositories.enabled` to `false` will disable Spring Data JPA.

Including Entities from Different Packages

By default, Spring Boot will detect components, repositories, and entities starting from the package the `@SpringBootApplication` annotated class is in. But what if you have entities in a different package that still need to be included? For this you can use the `@EntityScan` annotation; it works like `@ComponentScan` but then for `@Entity` annotated beans.

```
package com.apress.springboot2recipes.order;

import javax.persistence.Entity;
import javax.persistence.Id;
import java.util.Objects;

@Entity
public class Order {
```

⁵<https://projects.spring.io/spring-data-jpa/>

```

@Id
private long id;
private String number;

public long getId() {
    return id;
}

public void setId(long id) {
    this.id = id;
}

public String getNumber() {
    return number;
}

public void setNumber(String number) {
    this.number = number;
}
// Other methods omitted
}

```

This Order class is in the `com.apress.springboot2recipes.order` package, which isn't covered by the `@SpringBootApplication` annotated class as that is in the `com.apress.springboot2.recipes.jpa` package. To have this entity detected, you can add the `@EntityScan` annotation with the package(s) to scan to your `@SpringBootApplication` annotated class (or a regular `@Configuration` class).

```

@SpringBootApplication
@EntityScan({
    "com.apress.springboot2recipes.order",
    "com.apress.springboot2recipes.jpa" })

```

With this addition the Order entity will now be detected and accessible by JPA.

Testing JPA Repositories

When testing JPA code, a database is required, often an embedded database like H2, Derby, or HSQLDB is used for testing. Spring Boot makes it very easy to write tests for JPA. JPA-based tests can be annotated with `@DataJpaTest`, and Spring Boot will create a minimal application with only the JPA-related beans like a `DataSource`, transaction manager, and if needed Spring Data JPA repositories.

Let's write a test for the `CustomerRepository` and use H2 as the embedded database. First, add H2 as a test dependency.

```
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
  <scope>test</scope>
</dependency>
```

Next, create the `CustomerRepositoryTest`.

```
@RunWith(SpringRunner.class)
@DataJpaTest
@TestPropertySource(properties = "spring.flyway.enabled=false")
public class CustomerRepositoryTest {

    @Autowired
    private CustomerRepository repository;

    @Autowired
    private TestEntityManager testEntityManager
}
```

The `@RunWith(SpringRunner.class)` executes the test by a special JUnit runner to bootstrap the Spring Test Context framework. The `@DataJpaTest` will replace the preconfigured `DataSource` with an embedded one (H2 in this case). It will also bootstrap the JPA components and, when detected, Spring Data JPA repositories.

Spring Boot also offers a `TestEntityManager`, which offers some convenience methods to easily store and find data for testing.

Finally there is the `@TestPropertySource (properties = "spring.flyway.enabled=false")`, which indicates that we want to disable Flyway. The application uses Flyway to manage the schema; however, those scripts are written for PostgreSQL and

not H2. For testing, you can let Hibernate manage the schema, which is also the default setting when using an embedded database with Spring Boot.

Now, write a test that checks if records get inserted correctly.

```
@Test
public void insertNewCustomer() {
    assertThat(repository.findAll()).isEmpty();

    Customer customer = repository.save(new Customer(-1, "T. Testing",
        "t.testing@test123.tst"));

    assertThat(customer.getId()).isGreaterThan(-1L);
    assertThat(customer.getName()).isEqualTo("T. Testing");
    assertThat(customer.getEmail()).isEqualTo("t.testing@test123.tst");

    assertThat(repository.findById(customer.getId())).isEqualTo(customer);
}
```

This test first asserts that the database is empty—not really required but can be useful to detect if other tests pollute the database. Next, a `Customer` is added to the database by calling the `save` method on the `CustomerRepository`. The resulting `Customer` is validated to have an id, and if the name and email attributes are there. Finally, the `Customer` is retrieved again and compared to be the same.

Another test you could add is the `findAll` method. When inserting two records, calling `findAll` should result in two records being retrieved.

```
@Test
public void findAllCustomers() {
    assertThat(repository.findAll()).isEmpty();

    repository.save(new Customer(-1, "T. Testing1", "t.testing@test123.tst"));
    repository.save(new Customer(-1, "T. Testing2", "t.testing@test123.tst"));

    assertThat(repository.findAll()).hasSize(2);
}
```

Of course there could be more assertions, but saving the data is already verified in the other test method.

7-4 Use Plain Hibernate Problem

You have some code that uses the plain Hibernate API, Session, and/or SessionFactory that you want to use with Spring Boot.

Solution

Use the EntityManager or EntityManagerFactory to obtain the plain Hibernate objects like Session or SessionFactory.

How It Works

When migrating older code to Spring Boot or when using a third-party library, it might not always be feasible to migrate everything to JPA. You want to use that code as is or with as little modification possible. There are two ways you can use the plain Hibernate API: either by getting the Session from the current EntityManager or configuring the SessionFactory. Which you want or can use depends on the influence you have on the code using the plain Hibernate API.

Using the EntityManager to Obtain the Session

To obtain the underlying Hibernate Session, the unwrap method can be used. This method has been added in JPA 2.0 to have a unified and convenient way to obtain the underlying JPA implementor classes, for those cases where you really need to access the native classes.

```
package com.apress.springboot2recipes.jpa;  
  
import org.hibernate.Session;  
import org.springframework.stereotype.Repository;  
  
import javax.persistence.EntityManager;  
import javax.persistence.PersistenceContext;  
import javax.transaction.Transactional;  
import java.util.List;
```

```

@Repository
@Transactional
class HibernateCustomerRepository implements CustomerRepository {

    @PersistenceContext
    private EntityManager em;

    private Session getSession() {
        return em.unwrap(Session.class);
    }

    @Override
    public List<Customer> findAll() {
        return getSession().createQuery("SELECT c FROM Customer c", Customer.
            class).getResultList();
    }

    @Override
    public Customer findById(long id) {
        return getSession().find(Customer.class, id);
    }

    @Override
    public Customer save(Customer customer) {
        getSession().persist(customer);
        return customer;
    }
}

```

The `HibernateCustomerRepository` gets the `EntityManager` injected, and to obtain the `Session` (see the `getSession` method) the `unwrap` method is used. Now you can use the `Hibernate Session` to perform queries, etc.

Using the SessionFactory

As of `Hibernate 5.2` the `Hibernate SessionFactory` extends `EntityManagerFactory` and as such can be used to create both a `Session` and `EntityManager`. `Spring 5.1` added support for this in the `LocalSessionFactoryBean` and this can be used to mix both native `Hibernate` code and `JPA` code.

To configure a SessionFactory bean, use the LocalSessionFactoryBean.

```
@Bean
public LocalSessionFactoryBean sessionFactory(DataSource dataSource) {

    Properties properties = new Properties();
    properties.setProperty(DIALECT,
        "org.hibernate.dialect.PostgreSQL95Dialect");

    LocalSessionFactoryBean sessionFactoryBean = new
    LocalSessionFactoryBean();
    sessionFactoryBean.setDataSource(dataSource);
    sessionFactoryBean.setPackagesToScan("com.apress.springboot2recipes.jpa");
    sessionFactoryBean.setHibernateProperties(properties);

    return sessionFactoryBean;
}
```

The SessionFactory requires a DataSource to obtain connections to the database, a dialect to create SQL, and it needs to know where the entities are. This is the minimal configuration needed for a SessionFactory. There is no need to add the HibernateTransactionManager, as the default configured JpaTransactionManager can take care of the transactions. However, as JPA has some limitations (especially in transaction propagation levels) you might want to add the HibernateTransactionManager, which doesn't suffer from these limitations.

The modified HibernateCustomerRepository uses the getCurrentSession method from the SessionFactory to obtain the Session.

```
@Repository
@Transactional
class HibernateCustomerRepository implements CustomerRepository {

    private final SessionFactory sf;

    HibernateCustomerRepository(SessionFactory sf) {
        this.sf=sf;
    }
}
```

```
private Session getSession() {  
    return sf.getCurrentSession();  
}  
// unmodified methods omitted  
}
```

Note Because the SessionFactory can also create the EntityManager, the CustomerRepository from Recipe 7-3 could be used as well!

7-5 Spring Data MongoDB

Problem

You want to use MongoDB in your Spring Boot application.

Solution

Add the Mongo Driver as a dependency and use the `spring.data.mongodb` properties to let Spring Boot set up the `MongoTemplate` to the correct MongoDB.

How It Works

Spring Boot automatically detects the presence of the MongoDB driver combined with the Spring Data MongoDB classes. If those are found, MongoDB as well as the `MongoTemplate` (among others) will be automatically set up for you.

Use the `MongoTemplate`

Now that the connection is set up you can use it, ideally through the `MongoTemplate` to make it easier to store and retrieve documents. First you need a document you want to store; let's create a `Customer` that we want to persist.

```
package com.apress.springboot2recipes.mongo;  
  
public class Customer {
```

```
private String id;  
  
private final String name;  
private final String email;  
  
Customer() {  
    this(null,null);  
}  
  
Customer(String name, String email) {  
    this.name = name;  
    this.email = email;  
}  
  
public String getId() {  
    return id;  
}  
  
public String getName() {  
    return name;  
}  
  
public String getEmail() {  
    return email;  
}  
// equals(), hashCode() and toString() omitted  
}
```

The `id` field will automatically be mapped to the `_id` document identifier of MongoDB. If you would like to use another field, you could use the `@Id` annotation from Spring Data to specify which field should be used.

Let's create a `CustomerRepository` so that instances can be saved and retrieved.

```
package com.apress.springboot2recipes.mongo;  
  
import java.util.List;  
  
public interface CustomerRepository {
```

```
List<Customer> findAll();
Customer findById(long id);
Customer save(Customer customer);
}
```

The MongoDB CustomerRepository is implemented by using a MongoTemplate.

```
package com.apress.springboot2recipes.mongo;

import org.springframework.data.mongodb.core.MongoTemplate;
import org.springframework.stereotype.Repository;

import java.util.List;

@Repository
class MongoCustomerRepository implements CustomerRepository {

    private final MongoTemplate mongoTemplate;

    MongoCustomerRepository(MongoTemplate mongoTemplate) {
        this.mongoTemplate = mongoTemplate;
    }

    @Override
    public List<Customer> findAll() {
        return mongoTemplate.findAll(Customer.class);
    }

    @Override
    public Customer findById(long id) {
        return mongoTemplate.findById(id, Customer.class);
    }

    @Override
    public Customer save(Customer customer) {
        mongoTemplate.save(customer);
        return customer;
    }
}
```

The MongoCustomerRepository uses the preconfigured MongoTemplate to store and retrieve customers in Mongo.

To use all the parts, first there needs to be some data in MongoDB; an ApplicationRunner can be used to insert some data.

```
@Component
@Order(1)
class DataInitializer implements ApplicationRunner {

    private final CustomerRepository customers;

    DataInitializer(CustomerRepository customers) {
        this.customers = customers;
    }

    @Override
    public void run(ApplicationArguments args) throws Exception {
        List.of(
            new Customer("Marten Deinum", "marten.deinum@conspect.nl"),
            new Customer("Josh Long", "jlong@pivotal.io"),
            new Customer("John Doe", "john.doe@island.io"),
            new Customer("Jane Doe", "jane.doe@island.io"))
            .forEach(customers::save);
    }
}
```

The DataInitializer will use the CustomerRepository to save some Customer instances into MongoDB. Notice the @Order we want this to execute first this can be enforced by explicitly ordering the bean.

Next create an ApplicationRunner to retrieve all the customers from MongoDB.

```
@Component
class CustomerLister implements ApplicationRunner {

    private final Logger logger = LoggerFactory.getLogger(getClass());
    private final CustomerRepository customers;

    CustomerLister(CustomerRepository customers) {
        this.customers = customers;
    }
```

```

    }

@Override
public void run(ApplicationArguments args) {
    customers.findAll().forEach( customer -> logger.info("{}", customer));
}
}
}

```

This listener will use the `CustomerRepository` to load all customers from the database and print a line in the logs.

Finally the application class to bootstrap all this (including the two aforementioned `ApplicationListeners`).

```

package com.apress.springboot2recipes.mongo;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.ApplicationArguments;
import org.springframework.boot.ApplicationRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.core.annotation.Order;
import org.springframework.stereotype.Component;

import java.util.Arrays;

@SpringBootApplication
public class MongoApplication {

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

@Component
@Order(1)
class DataInitializer implements ApplicationRunner {

```

```

private final CustomerRepository customers;

DataInitializer(CustomerRepository customers) {
    this.customers = customers;
}

@Override
public void run(ApplicationArguments args) throws Exception {
    List.of(
        new Customer("Marten Deinum", "marten.deinum@conspect.nl"),
        new Customer("Josh Long", "jlong@pivotal.io"),
        new Customer("John Doe", "john.doe@island.io"),
        new Customer("Jane Doe", "jane.doe@island.io"))
        .forEach(customers::save);
}
}

@Component
class CustomerLister implements ApplicationRunner {

    private final Logger logger = LoggerFactory.getLogger(getClass());
    private final CustomerRepository customers;

    CustomerLister(CustomerRepository customers) {
        this.customers = customers;
    }

    @Override
    public void run(ApplicationArguments args) {
        customers.findAll().forEach( customer -> logger.info("{}",
            customer));
    }
}

```

When running the application, it will automatically connect to the MongoDB instance, insert data into it, and finally retrieve it and print it to the logs.

You will need a MongoDB instance to store and retrieve documents. You can either use an embedded MongoDB (ideal for testing) or connect to an actual MongoDB instance.

Use Embedded MongoDB

An easy project to use is the Embedded MongoDB⁶; this will automatically start a MongoDB when Spring Boot starts. Spring Boot even has extensive support for this embedded MongoDB through properties in the `spring.mongodb.embedded` namespace (Table 7-5). The only thing needed for this to work is to add the necessary dependency.

```
<dependency>
<groupId>de.flapdoodle.embed</groupId>
<artifactId>de.flapdoodle.embed.mongo</artifactId>
</dependency>
```

Table 7-5. *Embedded MongoDB Properties*

Property	Description
<code>spring.mongodb.embedded.version</code>	The version of MongoDB to use. Default 3.2.2
<code>spring.mongodb.embedded.features</code>	List of features to enable, default only sync-delay.
<code>spring.mongodb.embedded.storage.database-dir</code>	Directory to store the data in
<code>spring.mongodb.embedded.storage.oplog-size</code>	Size of the log in megabytes.
<code>spring.mongodb.embedded.storage.repl-set-name-dir</code>	Name of the replica set, default null.

Now when running the `MongoApplication`, it will automatically bootstrap MongoDB as well and tear it down when the application stops.

Connect to an External MongoDB

Note The `bin` directory contains a `mongo.sh` script that will start a MongoDB instance using Docker.

⁶<https://flapdoodle-oss.github.io/de.flapdoodle.embed.mongo/>

When starting the `MongoApplication` without the Embedded MongoDB, it will by default try to connect to a MongoDB server on localhost and port 27017. If that is not where you want to connect to, use the `spring.data.mongodb` properties (Table 7-6) to configure the correct location.

Table 7-6. MongoDB Properties

Property	Description
<code>spring.data.mongodb.uri</code>	Mongo database URI, including credentials, settings etc. Default <code>mongodb://localhost/test</code>
<code>spring.data.mongodb.username</code>	Login user of the mongo server. Default none
<code>spring.data.mongodb.password</code>	Login password of the mongo server. Default none
<code>spring.data.mongodb.host</code>	Hostname (or IP address) if the MongoDB server. Default fallback to <code>localhost</code>
<code>spring.data.mongodb.port</code>	Port of the MongoDB server. Default fallback to 27017
<code>spring.data.mongodb.database</code>	Name of the MongoDB database/collection to use
<code>spring.data.mongodb.field-naming-strategy</code>	FQN of the <code>FieldNamingStrategy</code> used to map object fields to document fields. Defaults to <code>PropertyNameFieldNamingStrategy</code>
<code>spring.data.mongodb.authentication-database</code>	Name of MongoDB to use for authentication. Defaults to <code>spring.data.mongodb.database</code> value
<code>spring.data.mongodb.grid-fs-database</code>	Name of MongoDB to connect to. Defaults to <code>spring.data.mongodb.database</code> value

Warning The `spring.data.mongodb.username`, `spring.data.mongodb.password`, `spring.data.mongodb.host`, `spring.data.mongodb.port`, and `spring.data.mongodb.database` are only supported for MongoDB 2.0; for MongoDB 3.0 and up, you have to use the `spring.data.mongodb.uri` property!

Although you can use Spring Boot to configure the MongoClient, if you need more control you can always specify the MongoFactory or MongoClient yourself as a bean and Spring Boot will not autoconfigure the MongoClient. It will still detect Spring Data MongoDB classes and will enable repository support.

Use Spring Data MongoDB Repositories

Instead of writing your own implementation of the repository, you can also use Spring Data MongoDB to create them for you (just like as with Spring Data JPA). For this you need to extend one of the Repository interfaces from Spring Data. The easiest is to extend CrudRepository or MongoRepository.

```
public interface CustomerRepository extends MongoRepository<Customer, String>
{ }
```

This is all we need to get a fully functional repository, you can remove the MongoCustomerRepository implementation. This will give you save, findAll, findById, and many more methods.

The application will still run, insert some customers, and list them in the logs.

Reactive MongoDB Repositories

Instead of using regular blocking operations, MongoDB can also be used in a reactive fashion. For this, use the `spring-boot-starter-data-mongodb-reactive` instead of the `spring-boot-starter-data-mongodb`. This dependency will include the needed reactive libraries and reactive driver for MongoDB.

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

With the dependencies in order, you can make the CustomerRepository a reactive one. This is done by simply extending ReactiveRepository, ReactiveSortingRepository, or ReactiveMongoRepository depending on your needs.

```
public interface CustomerRepository
  extends ReactiveMongoRepository<Customer, String> { }
```

Now that `CustomerRepository` extends `ReactiveMongoRepository`, all the methods return either `Flux` (zero or more elements) or `Mono` (zero or one element).

Note The default implementations use Project Reactor⁷ as the reactive framework; however, you can also use RxJava.⁸ Then instead of a `Flux` and `Mono` you will use an `Observable` or `Single`. For this you need to extend the `RxJava2CrudRepository` or `RxJava2SortingRepository`.

The `DataInitializer` needs to be made reactive as well.

```
@Component
@Order(1)
class DataInitializer implements ApplicationRunner {

    private final CustomerRepository customers;

    DataInitializer(CustomerRepository customers) {
        this.customers = customers;
    }

    @Override
    public void run(ApplicationArguments args) throws Exception {
        customers.deleteAll()
            .thenMany(
                Flux.just(
                    new Customer("Marten Deinum", "marten.deinum@conspect.nl"),
                    new Customer("Josh Long", "jlong@pivotal.io"),
                    new Customer("John Doe", "john.doe@island.io"),
                    new Customer("Jane Doe", "jane.doe@island.io"))
            .flatMap(customers::save).subscribe(System.out::println);
    }
}
```

⁷<https://projectreactor.io>

⁸<https://github.com/ReactiveX/RxJava>

First it deletes everything, and then we create new `Customers` and add them to the repository. Finally we need to subscribe to this flow, else nothing will happen. Instead of subscribing you could also block, but blocking in a reactive application should be avoided.

Finally, also the `CustomerListener` needs to be made reactive.

```
@Component
class CustomerLister implements ApplicationRunner {

    private final Logger logger = LoggerFactory.getLogger(getClass());
    private final CustomerRepository customers;

    CustomerLister(CustomerRepository customers) {
        this.customers = customers;
    }

    @Override
    public void run(ApplicationArguments args) {
        customers.findAll().subscribe(customer -> logger.info("{}",
            customer));
    }
}
```

It will `findAll` the customers from the repository and for each will print a line in the log file.

When starting the application you will probably see nothing in the log files. Due to the reactive nature of the application, it finishes quickly and the `CustomerListener` doesn't have time to register itself and start listening. To prevent the shutdown of the application, add a `System.in.read()`; this will keep the application running until you press Enter. Generally when running an application this isn't needed, because the application is exposed as a service/web application, which keeps running automatically.

```
public static void main(String[] args) throws IOException {
    SpringApplication.run(ReactiveMongoApplication.class, args);
    System.in.read();
}
```

Now when running the application you will see that the customers will be retrieved from MongoDB and printed in the logs. This might not look like a huge difference, but the way in which the retrieval is done is totally different. This can be made clear with a small modification to the DataInitializer'; let's delay the elements with 250ms.

```

@Component
@Order(1)
class DataInitializer implements ApplicationRunner {

    private final CustomerRepository customers;

    DataInitializer(CustomerRepository customers) {
        this.customers = customers;
    }

    @Override
    public void run(ApplicationArguments args) throws Exception {
        customers.deleteAll().thenMany(
            Flux.just(
                new Customer("Marten Deinum", "marten.deinum@conspect.nl"),
                new Customer("Josh Long", "jlong@pivotal.io"),
                new Customer("John Doe", "john.doe@island.io"),
                new Customer("Jane Doe", "jane.doe@island.io")))
            .delayElements(Duration.ofMillis(250))
            .flatMap(customers::save)
            .subscribe(System.out::println);
    }
}

```

Now the insert of each individual element will be delayed with 250ms. When running the application you will see that the customer listener also takes some time for each element to appear.

Testing Mongo Repositories

When testing MongoDB code a running Mongo instance is used; for testing an embedded MongoDB is often used. Spring Boot makes it very easy to write tests for MongoDB, using the `@DataMongoTest`. Spring Boot will create a minimal application with only the MongoDB-related beans and start an embedded MongoDB (if detected).

Let's write a test for the `CustomerRepository` and use an embedded MongoDB. First, add the embedded MongoDB as a test dependency.

```
<dependency>
  <groupId>de.flapdoodle.embed</groupId>
  <artifactId>de.flapdoodle.embed.mongo</artifactId>
  <scope>test</scope>
</dependency>
```

Next create the `CustomerRepositoryTest`.

```
@RunWith(SpringRunner.class)
@DataMongoTest
public class CustomerRepositoryTest {

    @Autowired
    private CustomerRepository repository;

    @After
    public void cleanUp() {
        repository.deleteAll();
    }
}
```

The `@RunWith(SpringRunner.class)` executes the test by a special JUnit runner to bootstrap the Spring Test Context framework. The `@DataMongoTest` will replace the preconfigured MongoDB with an embedded one (if on the classpath). It will also bootstrap the MongoDB components and, when detected, Spring Data Mongo repositories.

After each test, we want to make sure that the embedded MongoDB doesn't contain any data. This can be achieved by adding an `@After` annotated method and calling `deleteAll` on the repository. This method will be called after each executed test method.

Now write a test that checks if records get inserted correctly.

```
@Test
public void insertNewCustomer() {
    assertThat(repository.findAll()).isEmpty();

    Customer customer = repository.save(new Customer(-1, "T. Testing",
    "t.testing@test123.tst"));

    assertThat(customer.getId()).isGreaterThan(-1L);
    assertThat(customer.getName()).isEqualTo("T. Testing");
    assertThat(customer.getEmail()).isEqualTo("t.testing@test123.tst");

    assertThat(repository.findById(customer.getId())).isEqualTo(customer);
}
```

This test first asserts that the database is empty—not really required but can be useful to detect if other tests pollute the database. Next, a `Customer` is added to the database by calling the `save` method on the `CustomerRepository`. The resulting `Customer` is validated to have an id, and if the name and email attributes are there. Finally, the `Customer` is retrieved again and compared to be the same.

Another test you could add is the `findAll` method. When inserting two records, calling `findAll` should result in two records being retrieved.

```
@Test
public void findAllCustomers() {
    assertThat(repository.findAll()).isEmpty();

    repository.save(new Customer(-1, "T. Testing1", "t.testing@test123.tst"));
    repository.save(new Customer(-1, "T. Testing2", "t.testing@test123.tst"));

    assertThat(repository.findAll()).hasSize(2);
}
```

Of course there could be more assertions, but saving the data is already verified in the other test method.

CHAPTER 8

Java Enterprise Services

In this chapter, you will learn about Spring's support for the most common Java enterprise services: Java Management Extensions (JMX), sending e-mail with JavaMail, background processing, and scheduling tasks.

JMX is part of JavaSE and is a technology for managing and monitoring system resources such as devices, applications, objects, and service-driven networks. These resources are represented as managed beans (MBeans). Spring supports JMX by exporting any Spring bean as model MBeans without programming against the JMX API. In addition, Spring can easily access remote MBeans.

JavaMail is the standard API and implementation for sending e-mail in Java. Spring further provides an abstract layer to send e-mail in an implementation-independent fashion.

8-1 Spring Asynchronous Processing Problem

You want to asynchronously invoke a method with a long-running method.

Solution

Spring has support to configure a TaskExecutor and the ability to asynchronously execute methods annotated with @Async. This can be done in a transparent way without the normal setup for doing asynchronous execution. Spring Boot, however, will not automatically detect the need for asynchronous method execution. This support has to be enabled with the @EnableAsync configuration annotation.

How It Works

Let's write a component that prints something in an asynchronous way to the console.

```
package com.apress.springbootrecipes.scheduling;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.scheduling.annotation.Async;
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Component;

@Component
public class HelloWorld {

    private static final Logger logger = LoggerFactory.
        getLogger(HelloWorld.class);

    @Async
    public void printMessage() throws InterruptedException {
        Thread.sleep(500);
        logger.info("Hello World, from Spring Boot 2!");
    }
}
```

The class will wait for 500 milliseconds before printing something to the logger. Notice the `@Async` annotation on the method; this indicates that the method can be executed in an asynchronous manner. However, this support has to be explicitly enabled for a Spring Boot application.

To enable asynchronous processing, the `@EnableAsync` configuration annotation is needed. The easiest solution is to add this annotation to your application class.

```
package com.apress.springbootrecipes.scheduling;

import org.springframework.boot.ApplicationRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.scheduling.annotation.EnableAsync;
import org.springframework.scheduling.annotation.EnableScheduling;
```

```

import java.io.IOException;

@SpringBootApplication
@EnableAsync
public class ThreadingApplication {

    public static void main(String[] args) throws IOException {
        SpringApplication.run(ThreadingApplication.class, args);

        System.out.println("Press [ENTER] to quit:");
        System.in.read();
    }

    @Bean
    public ApplicationRunner startupRunner(HelloWorld hello) {
        return (args) -> {hello.printMessage();};
    }
}

```

Spring Boot will by default create a TaskExecutor named applicationTaskExecutor. When adding @EnableAsync, it will automatically detect this instance and use it for asynchronous execution of methods, and also will enable the detection of @Async annotated methods.

The System.in.read is in there to prevent the application from shutting down, so that the background task can finish processing. When pressing the ENTER key the program will quit. Generally, when developing a web application you don't need things like this.

Configuring the TaskExecutor

Spring Boot will automatically configure a ThreadPoolTaskExecutor. This can be configured through the properties in the spring.task.execution namespace (Table 8-1).

Table 8-1. Spring Boot Task Executor Properties

Property	Description
spring.task.execution.pool.core-size	Number of core threads, default 8
spring.task.execution.pool.max-size	Maximum number of threads, default Integer.MAX_VALUE
spring.task.execution.pool.queue-capacity	Queue capacity. Default is unbounded and this by default ignores the max-size property.
spring.task.execution.pool.keep-alive	Limit of how long threads can be idle before being terminated
spring.task.execution.thread-name-prefix	Prefix to use for the names of newly created threads. Default is task-
spring.task.execution.pool.allow-core-thread-timeout	Are core threads allowed to time out, default true. This enables dynamically growing and shrinking the pool.

Adding the following to the application.properties will override some of the defaults.

```
spring.task.execution.pool.core-size=4
spring.task.execution.pool.max-size=16
spring.task.execution.pool.queue-capacity=125
spring.task.execution.thread-name-prefix=sbr-exec-
```

Now when rerunning the application, the name of the thread will start with sbr-exec. It will start four threads and will grow to a maximum size of 16. To enable pool resizing, it is required to give a fixed number to the capacity of the queue. Too large (or unbounded) will not increase the number of threads automatically.

Use the TaskExecutorBuilder to Create a TaskExecutor

If you need to construct a TaskExecutor, Spring Boot provides the TaskExecutorBuilder. This builder class makes it easier to construct a ThreadPoolTaskExecutor. It allows us to set the same properties as shown in Table 8-1.

```

@Bean
public TaskExecutor customTaskExecutor(TaskExecutorBuilder builder) {
    return builder.corePoolSize(4)
        .maxPoolSize(16)
        .queueCapacity(125)
        .threadNamePrefix("sbr-exec-").build();
}

```

If there are, for some reason, multiple TaskExecutor instances in your application, you would need to either mark one of them as `@Primary` to be used as the default TaskExecutor or use the AsyncConfigurer interface and implement the taskExecutor method to return the default TaskExecutor to use.

```

@SpringBootApplication
@EnableAsync
public class ThreadingApplication implements AsyncConfigurer {

    @Bean
    public ThreadPoolTaskExecutor taskExecutor() { ... }

    @Override
    public Executor getAsyncExecutor() {
        return taskExecutor();
    }
}

```

8-2 Spring Task Scheduling Problem

You want to schedule a method invocation in a consistent manner, using either a cron expression, an interval, or a rate.

Solution

Spring has support to configure TaskExecutor s and TaskScheduler s. This capability, coupled with the ability to schedule method execution using the `@Scheduled` annotation, makes Spring scheduling support work with a minimum of fuss: all you need are a

method, an annotation, and to have switched on the scanner for annotations. Spring Boot will not automatically detect the need for scheduling; you will have to enable it yourself using the `@EnableScheduling` annotation.

How It Works

Let's write a component that prints out a message in the logging every four seconds. Create a Java class and use the `@Scheduled` annotation on a method to indicate that this needs to be invoked. Using `fixedRate=4000` as an argument, it will run every four seconds. If you want to use a cron expression, you can set the `cron` attribute on the `@Scheduled` annotation instead.

```
package com.apress.springbootrecipes.scheduling;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Component;

@Component
public class HelloWorld {

    private static final Logger logger =
        LoggerFactory.getLogger(HelloWorld.class);

    @Scheduled(fixedRate = 4000)
    public void printMessage() {
        logger.info("Hello World, from Spring Boot 2!");
    }
}
```

The `@Component` will make sure that it will be detected by Spring Boot.

The next thing to do is enable scheduling for your application. The easiest solution is to annotate the application class with `@EnableScheduling`. Of course you can also place it on other `@Configuration` annotated classes.

```

package com.apress.springbootrecipes.scheduling;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.scheduling.annotation.EnableScheduling;

@SpringBootApplication
@EnableScheduling
public class SchedulingApplication {

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

```

The `@EnableScheduling` will enable the detection of `@Scheduled` annotated methods and will register a `TaskScheduler` to use for scheduling tasks. When a single `TaskScheduler` is detected in the application context, it will use that one rather than creating a new one.

Running the `SchedulingApplication` will give you an output in the logs about every four seconds.

Instead of using the `@Scheduled` annotation, you could also schedule a method using Java. This can be needed if you cannot place a `@Scheduled` annotation on the method you want to execute periodically or you just want to limit the amount of annotations. For this you can use the `SchedulingConfigurer`, which has a single callback method to configure additional tasks.

```

@SpringBootApplication
@EnableScheduling
public class SchedulingApplication implements SchedulingConfigurer {

    @Autowired
    private HelloWorld helloWorld;

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

```

```
@Override  
public void configureTasks(ScheduledTaskRegistrar taskRegistrar) {  
    taskRegistrar.addFixedRateTask(  
        () -> helloWorld.printMessage()  
        , 4000);  
}  
}
```

8-3 Sending E-mail

Spring Boot will automatically configure the ability to send mail when mail properties and the java mail library are detected on the classpath. In this recipe we will take a look at how to set the properties and how to send an e-mail using Spring Boot.

Problem

You want to send e-mail from a Spring Boot application.

Solution

Spring's e-mail support makes it easier to send e-mail by providing an abstract and implementation-independent API for sending e-mail. The core interface of Spring's e-mail support is `MailSender`. The `JavaMailSender` interface is a subinterface of `MailSender` that includes specialized JavaMail features such as Multipurpose Internet Mail Extensions (MIME message) support. To send an e-mail message with HTML content, inline images, or attachments, you have to send it as a MIME message. Spring Boot will automatically configure the `JavaMailSender` when the `javax.mail` classes are found on the classpath and when the appropriate `spring.mail` properties have been set.

How It Works

The first thing to do is add the `spring-boot-starter-mail` dependency to your list of dependencies. This will add the necessary `javax.mail` as well as the `spring-context` dependencies on the classpath.

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

Configure the JavaMailSender

To be able to send mail, you need to configure the appropriate `spring.mail` properties (Table 8-2). At least the `spring.mail.host` property is needed; the other properties are optional.

Table 8-2. *Spring Boot Mail Properties*

Property	Description
<code>spring.mail.host</code>	The SMTP server host
<code>spring.mail.port</code>	The SMTP sever port (default 25)
<code>spring.mail.username</code>	Username to use for connecting to the SMTP server
<code>spring.mail.password</code>	Password to use for connecting to the SMTP server
<code>spring.mail.protocol</code>	The protocol used by the SMTP server (default <code>smtp</code>)
<code>spring.mail.test-connection</code>	Test if the SMTP server is available at startup (default <code>false</code>)
<code>spring.mail.default-encoding</code>	Encoding used for MIME messages (default <code>UTF-8</code>)
<code>spring.mail.properties.*</code>	Additional properties to be set on the JavaMail Session
<code>spring.mail.jndi-name</code>	JNDI name of the JavaMail Session, can be used when deploying to a JEE server with preconfigured JavaMail Sessions in JNDI.

Next, you need to define at least the `spring.mail.host` property to be able to send mail.

```
spring.mail.host=localhost
spring.mail.port=3025
```

Note The code for this recipe uses GreenMail as an SMTP server; a configured instance can be run using the `smtp.sh` script from the `bin` directory. It will, by default, expose an SMTP server on port 3025.

Sending a Plain Text E-mail

With the added dependencies and `spring.mail` properties, Spring Boot will add a preconfigured `JavaMailSenderImpl` as a bean to the `ApplicationContext`. This bean can be autowired into components, by either using an `@Autowired` field or through constructors or, as shown here, as dependencies in `@Bean` annotated methods.

```
package com.apress.springbootrecipes.mailsender;

import org.springframework.boot.ApplicationRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.mail.javamail.JavaMailSender;
import org.springframework.mail.javamail.MimeMessageHelper;

import javax.mail.Message;

@SpringBootApplication
public class MailSenderApplication {

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

    @Bean
    public ApplicationRunner startupMailSender(JavaMailSender mailSender) {
        return (args) -> {
            mailSender.send((msg) -> {
                var helper = new MimeMessageHelper(msg);
                helper.setTo("recipient@some.where");
                helper.setFrom("spring-boot-2-recipes@apress.com");
                helper.setSubject("Status message");
            });
        };
    }
}
```

```
        helper.setText("All is well.");
    });
};

}
```

The MailSenderApplication will send an e-mail when the application finished starting. The startupMailSender is an ApplicationRunner (see Chapter 2), which takes the preconfigured JavaMailSender to send a mail message.

Using Thymeleaf for E-mail Templates

Spring Boot has some nice support for using Thymeleaf¹ as a templating solution; however, the default setup is mainly for using Thymeleaf for webpages. It is, however, very possible to use Thymeleaf for e-mail templates.

First, add the `spring-boot-starter-thymeleaf` as a dependency. This will pull in all the needed Thymeleaf dependencies and will automatically configure the Thymeleaf `TemplateEngine`, which we need to generate the HTML content.

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

By default, the Spring configured Thymeleaf TemplateEngine will resolve the HTML templates from the templates directory under `src/main/resources`. Add a file named `email.html` to this directory and make a nice looking e-mail message from it.

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
</head>
<body>
<p><strong th:text="${msg}">Some email content will be here.</strong></p>
```

¹<https://www.thymeleaf.org>

```
<p>
Kind Regards,
Your Application
</p>
</body>
</html>
```

The `th:text` is a Thymeleaf tag and will replace the content with the value of that attribute. Of Course we would need to pass in a value for that attribute from our mail sending/generating code.

```
package com.apress.springbootrecipes.mailsender;

import org.springframework.boot.ApplicationRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.context.i18n.LocaleContextHolder;
import org.springframework.mail.javamail.JavaMailSender;
import org.springframework.mail.javamail.MimeMessageHelper;
import org.thymeleaf.context.Context;
import org.thymeleaf.spring5.SpringTemplateEngine;

import javax.mail.Message;
import java.util.Collections;

@SpringBootApplication
public class MailSenderApplication {

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

    @Bean
    public ApplicationRunner startupMailSender(
        JavaMailSender mailSender,
        SpringTemplateEngine templateEngine) {
```

```
return (args) -> {
    mailSender.send((msg) -> {
        var helper = new MimeMessageHelper(msg);
        helper.setTo("recipient@some.where");
        helper.setFrom("spring-boot-2-recipes@apress.com");
        helper.setSubject("Status message");

        var context = new Context(
            LocaleContextHolder.getLocale(),
            Collections.singletonMap("msg", "All is well!"));
        var body = templateEngine.process("email.html", context);
        helper.setText(body, true);
    });
};
```

The code still is very similar to the code as it was, but with this difference now we also have a `SpringTemplateEnginge` at our disposable to generate the HTML content for our e-mail. We use the `process` method to select the template we want to render, `email.html`, and pass in a `Context` object. The `Context` object is used by Thymeleaf to resolve the attributes, in our case the `msg` one.

8-4 Register a JMX MBean

Problem

You want to register an object in your Spring Boot application as a JMX MBean, to have the ability to look at services that are running and manipulate their state at runtime. This will allow you to perform tasks like rerun batch jobs, invoke methods, and change configuration metadata.

Solution

Spring Boot by default enables the Spring JMX support and will detect the @ManagedResource annotated beans and register them with the JMX server.

How It Works

First, let's inspect the default JMX support that is enabled by Spring Boot. Let's create a simple Spring Boot application that will keep running and use JConsole to inspect the running application.

```
package com.apress.springbootrecipes.jmx;

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

import java.io.IOException;

@SpringBootApplication
public class JmxApplication {

    public static void main(String[] args) throws IOException {
        SpringApplication.run(JmxApplication.class, args);

        System.out.print("Press [ENTER] to quit:");
        System.in.read();
    }
}
```

When the application is running you can start jconsole and you will be prompted with a screen in which you can select the local process to connect to. Select the one that is running the JmxApplication (Figure 8-1).

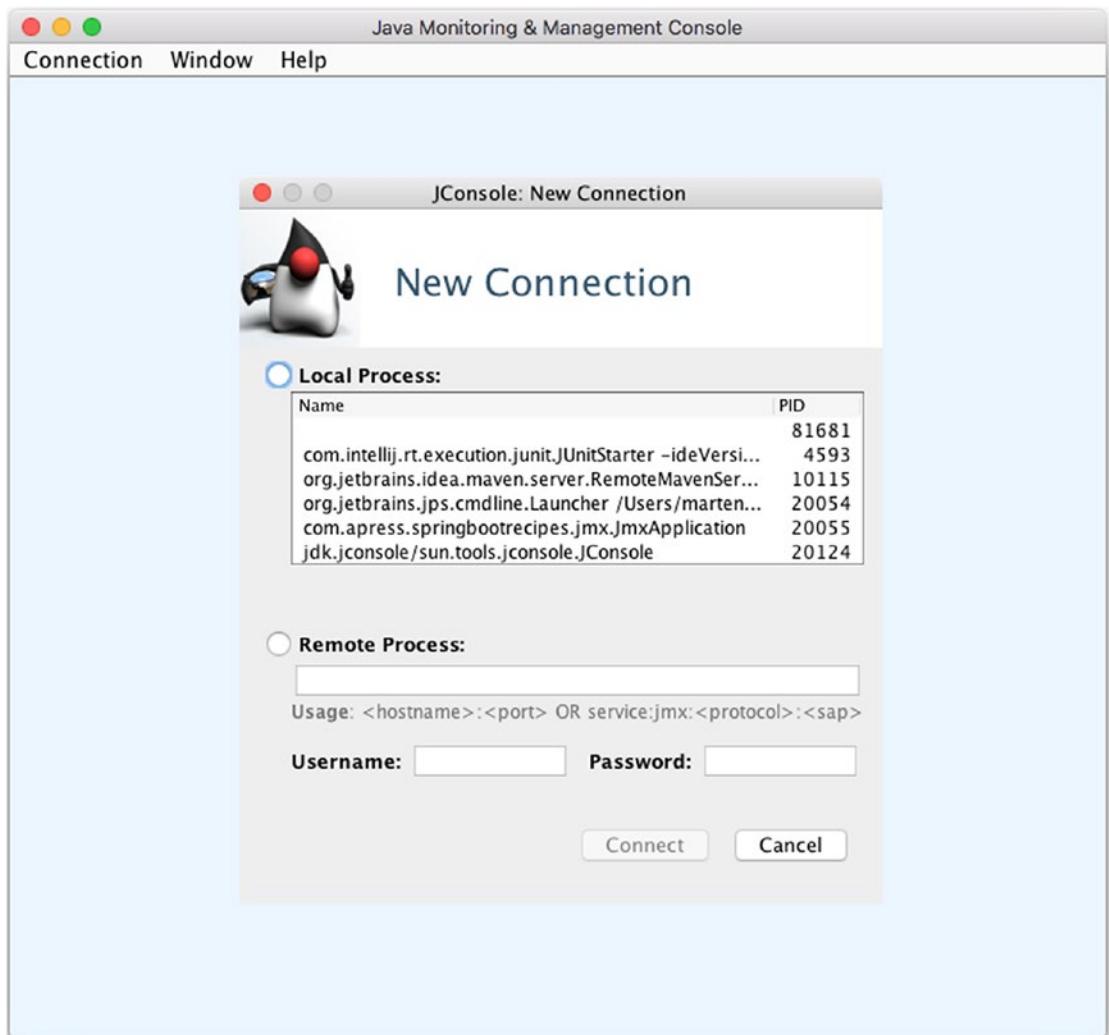


Figure 8-1. JConsole process selection

After selecting the process, go to the **MBeans** tab and open the `org.springframework.boot` menu and everything under it on the left side of the screen. There is a shutdown operation on the `SpringApplication` that you can invoke. When invoked, it will shut down the application (Figure 8-2).

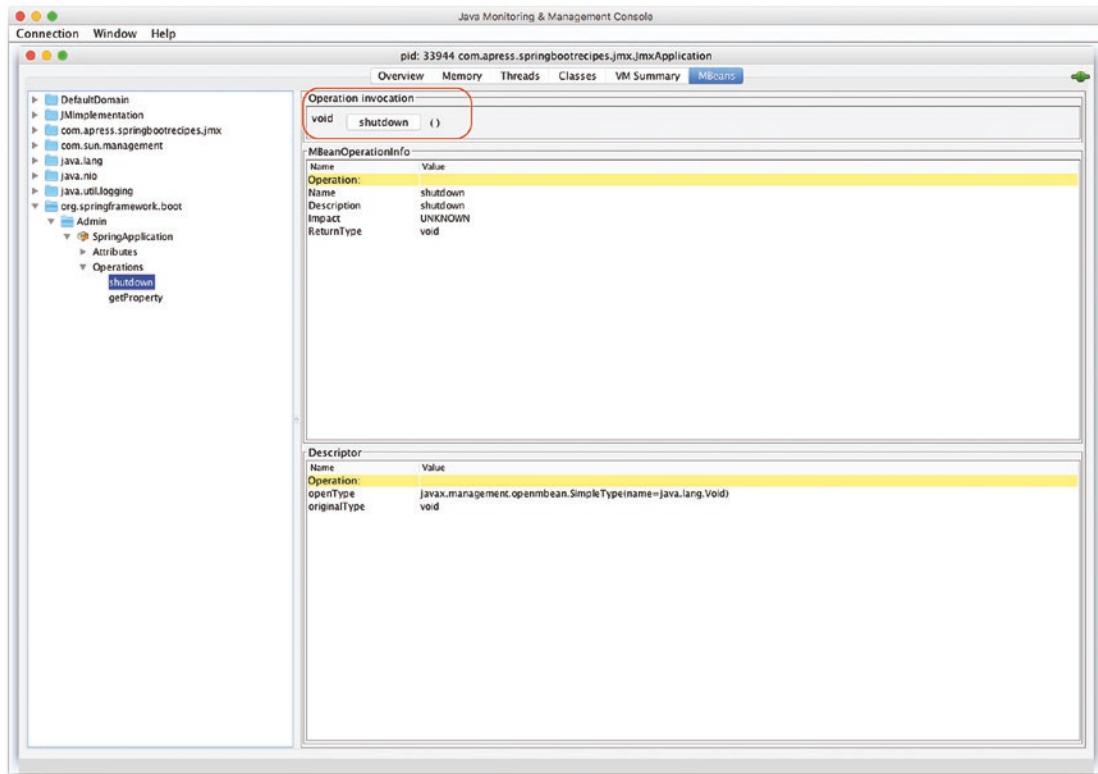


Figure 8-2. Invoking the shutdown method

To configure JMX, Spring Boot offers three properties (Table 8-3).

Table 8-3. Spring Boot JMX Properties

Property	Description
spring.jmx.enabled	Should JMX be enabled or not (default true)
spring.jmx.server	The bean name of the JMX MBeanServer to use (default mbeanServer). This is generally only needed if an MBeanServer has been manually registered in the application context.
spring.jmx.default-domain	The JMX domain name to use to register the beans (default is the package name)

As Spring Boot by default has JMX and the Spring support for JMX enabled, exposing a bean is pretty straightforward. For a bean to be exposed, it needs to have an `@ManagedResource` annotation and the operations to expose the `@ManagedOperation` annotation.

```
package com.apress.springbootrecipes.jmx;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.jmx.export.annotation.ManagedOperation;
import org.springframework.jmx.export.annotation.ManagedResource;
import org.springframework.scheduling.annotation.Async;
import org.springframework.stereotype.Component;

@Component
@ManagedResource
public class HelloWorld {

    private static final Logger logger =
        LoggerFactory.getLogger(HelloWorld.class);

    @ManagedOperation
    public void printMessage() {
        logger.info("Hello World, from Spring Boot 2!");
    }
}
```

When the application has been restarted and JConsole reconnected to the process running the `JmxApplication`, you will now notice a `com.apress.springbootrecipes.jmx` leaf in the menu on the left side. Open up all the nodes and in the Operations leaf you will find the `printMessage` operation (Figure 8-3).

CHAPTER 8 JAVA ENTERPRISE SERVICES

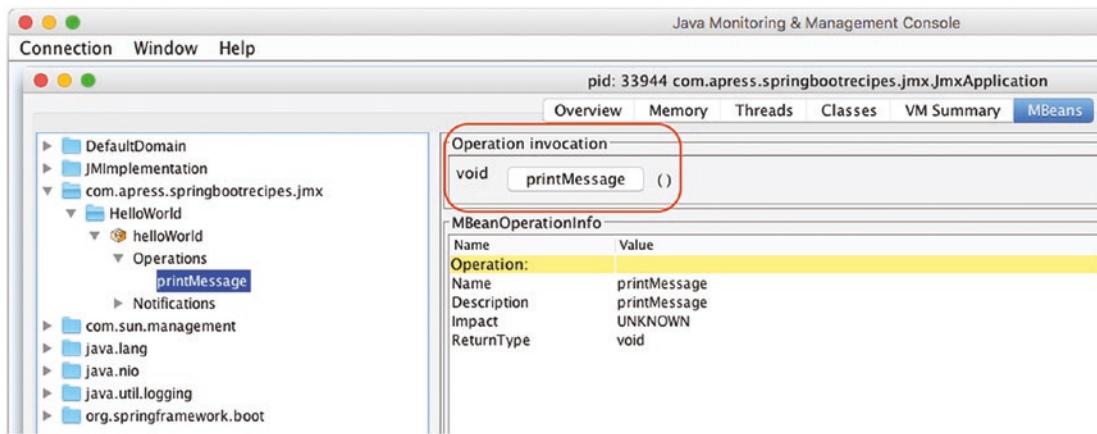


Figure 8-3. Invoking the printMessage method

When invoking the `printMessage` method, the console will print the message (Figure 8-4).

Figure 8-4. Console output

CHAPTER 9

Messaging

9-1 Configure JMS

Problem

You want to use JMS in a Spring Boot application and need to connect to the JMS broker.

Solution

Spring Boot supports auto-configuration for ActiveMQ¹ and Artemis.² Adding one of those JMS Providers together with setting some properties in, respectively, the `spring.activemq` and `spring.artemis` namespace will be all you need.

How It Works

By declaring a dependency of the JMS provider of your choice, Spring Boot will automatically configure the `ConnectionFactory` and strategy to look up destinations, the `DestinationResolver`, for your environment. This can also be done by using JNDI, and a final solution would be to do all the configuration yourself if you need more control over the `ConnectionFactory`.

Adding dependencies for the supported JMS providers is pretty easy, as Spring Boot has provides starter projects for those; for JNDI you need to include the JMS dependencies yourself.

¹<https://activemq.apache.org>

²<https://activemq.apache.org/artemis/>

Use ActiveMQ

When using ActiveMQ, the first thing to do is to include the `spring-boot-starter-activemq` and this will pull in all the needed JMS and ActiveMQ dependencies to get started. It will include the `spring-jms` dependency and the client libraries for ActiveMQ.

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

By default, Spring Boot will start an embedded broker if no explicit broker configuration is given. Configuration can be changed by using properties from the `spring.activemq` namespace (see Table 9-1).

Table 9-1. ActiveMQ Configuration Properties

Property	Description
<code>spring.activemq.broker-url</code>	URL of the broker to connect to, default <code>vm://localhost?broker.persistent=false</code> for an in-memory broker, else <code>tcp://localhost:61616</code>
<code>spring.activemq.user</code>	Username to use to connect to the broker, default empty
<code>spring.activemq.password</code>	Password to use to connect to the broker, default empty
<code>spring.activemq.in-memory</code>	Should an embedded broker be used, default true. Ignored when <code>spring.activemq.broker-url</code> has been explicitly set
<code>spring.activemq.non-blocking-redelivery</code>	Stop message delivery before redelivering rolled-back messages; when enabled message order will not be preserved!, default false
<code>spring.activemq.close-timeout</code>	Time to wait to consider a close to be effective, default 15 seconds
<code>spring.activemq.send-timeout</code>	Time to wait for a response from the broker, default 0 (unlimited)
<code>spring.activemq.packages.trust-all</code>	When using Java Serialization to send JMS messages should classes from all packages be trusted, default none (requires explicit setting the packages)
<code>spring.activemq.packages.trusted</code>	Comma-separated list of specific packages to trust

A simple application to list all the beans with jms in their name, this should include a bean named cachingJmsConnectionFactory.

```
@SpringBootApplication
public class JmsActiveMQApplication {

    private static final String MSG = "\tName: %10s, Type: %s\n";

    public static void main(String[] args) {
        var ctx = SpringApplication.run(JmsActiveMQApplication.class, args);

        System.out.println("# Beans: " + ctx.getBeanDefinitionCount());

        var names = ctx.getBeanDefinitionNames();
        Stream.of(names)
            .filter(name -> name.toLowerCase().contains("jms"))
            .forEach(name -> {
                Object bean = ctx.getBean(name);
                System.out.printf(MSG, name, bean.getClass().getSimpleName());
            });
    }
}
```

When running the preceding, it will print all the names and type of the beans containing jms in their name to the console. The output should be similar to that of Figure 9-1.



```
:: Spring Boot :: (v2.1.0.BUILD-SNAPSHOT)

2018-09-16 16:30:30.155 INFO 24235 --- [           main] c.a.s.demo.JmsActiveMQApplication      : Starting JmsActiveMQApplication on imac-van-marte
2018-09-16 16:30:30.158 INFO 24235 --- [           main] c.a.s.demo.JmsActiveMQApplication      : No active profile set, falling back to default pr
2018-09-16 16:30:31.034 INFO 24235 --- [           main] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExec
2018-09-16 16:30:31.077 INFO 24235 --- [           main] c.a.s.demo.JmsActiveMQApplication      : Started JmsActiveMQApplication in 1.246 seconds

# Beans: 54
Name:                                     cachingJmsConnectionFactory, Type: CachingConnectionFactory
Name:                                     jmsActiveMQApplication, Type: JmsActiveMQApplication$$Enh
Name:                                     jmsListenerContainerFactory, Type: DefaultJmsListenerContainernf
Name:                                     jmsListenerContainerFactoryConfigurer, Type: DefaultJmsListenerContainernf
Name:                                     jmsMessagingTemplate, Type: JmsMessagingTemplate
Name:                                     jmsTemplate, Type: JmsTemplate
```

Figure 9-1. ActiveMQ beans output

Use Artemis

When using Artemis, the first thing to do is to include `spring-boot-starter-artemis`. This will pull in all the needed JMS and Artemis dependencies to get started. It will include the `spring-jms` dependency and the libraries for Artemis.

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

Table 9-2. Artemis Configuration Properties

Property	Description
<code>spring.artemis.host</code>	Hostname for connecting to the Artemis broker, default localhost
<code>spring.artemis.port</code>	Port number for connecting to the Artemis broker, default 61616
<code>spring.artemis.user</code>	Port number for connecting to the Artemis broker, default empty
<code>spring.artemis.password</code>	Port number for connecting to the Artemis broker, default empty
<code>spring.artemis.mode</code>	Mode of operation either native or embedded, default is none leading to auto-detection of the mode. When the embedded classes are found, will run in embedded mode

A simple application to list all the beans with `jms` in their name, this should include a bean named `cachingJmsConnectionFactory`.

```
@SpringBootApplication
public class JmsActiveMQApplication {

    private static final String MSG = "\tName: %100s, Type: %s\n";

    public static void main(String[] args) {
        var ctx = SpringApplication.run(JmsActiveMQApplication.class, args);

        System.out.println("# Beans: " + ctx.getBeanDefinitionCount());

        var names = ctx.getBeanDefinitionNames();
```

```

Stream.of(names)
    .filter(name -> name.toLowerCase().contains("jms"))
    .sorted(Comparator.naturalOrder())
    .forEach(name -> {
        Object bean = ctx.getBean(name);
        System.out.printf(MSG, name, bean.getClass().
            getSimpleName());
    });
}
}

```

When run, the output will look like Figure 9-2.

```

:: Spring Boot :: (v2.1.0.BUILD-SNAPSHOT)

2018-09-16 16:33:30.901 INFO 24455 --- [           main] c.a.s.demo.JmsArtemisApplication      : Starting JmsArtemisApplication on imac-van-ma
ses started by marten in /Users/marten/Repositories/spring-boot-recipes/code/ch10/recipe_10_1_ii
2018-09-16 16:33:30.905 INFO 24455 --- [           main] c.a.s.demo.JmsArtemisApplication      : No active profile set, falling back to default
2018-09-16 16:33:31.855 INFO 24455 --- [           main] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskE
2018-09-16 16:33:31.920 INFO 24455 --- [           main] c.a.s.demo.JmsArtemisApplication      : Started JmsArtemisApplication in 1.322 seconds
# Beans: 50
Name:                                     cachingJmsConnectionFactory, Type: CachingConnectionFactory
Name:                                     jmsArtemisApplication, Type: JmsArtemisApplication$$EnhancerBySpringCGLIB$$3e
Name:                                     jmsListenerContainerFactory, Type: DefaultJmsListenerContain
Name:                                     jmsListenerContainerFactoryConfigurer, Type: DefaultJmsListenerContain
Name:                                     jmsMessagingTemplate, Type: JmsMessagingTemplate
Name:                                     jmsTemplate, Type: JmsTemplate

```

Figure 9-2. Artemis beans output

Note You might wonder when using Artemis if there is still an `ActiveMQConnectionFactory` in your configuration. Artemis is based on ActiveMQ, and as such shares classes with it.

Artemis can be used in embedded mode (just like ActiveMQ); it will then start an embedded broker. To configure it, there are several properties exposed in the `spring.artemis.embedded` namespace (see Table 9-3). The embedded mode requires `artemis-server` as an additional dependency.

```

<dependency>
  <groupId>org.apache.activemq</groupId>
  <artifactId>artemis-server</artifactId>
</dependency>

```

Table 9-3. Artemis Embedded Configuration Properties

Property	Description
spring.artemis.embedded.enabled	Should embedded mode be enabled, default true
spring.artemis.embedded.persistent	Are messages persisted, default false
spring.artemis.embedded.data-directory	Directory used to store the journal, only useful when persistent is true. Default is the Java Tempdir.
spring.artemis.embedded.queues	Comma-separated list of queues to create on startup
spring.artemis.embedded.topics	Comma-separated list of topics to create on startup
spring.artemis.embedded.cluster-password	Cluster password, default generated

JNDI

When deploying a Spring Boot application to a JEE container, there is a big chance that you want to use a preregistered ConnectionFactory from that container as well. To enable this, you would need a dependency on the `spring-jms` library and the `javax.jms-api` (the latter can then probably be marked, provided as well as that it will be supplied by your JEE container). You could use one of the starters and exclude the explicit ActiveMQ or Artemis dependencies; however, declaring only the needed dependencies is easier and clearer.

```

<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-jms</artifactId>
</dependency>
<dependency>
  <groupId>javax.jms</groupId>
  <artifactId>javax.jms-api</artifactId>
  <scope>provided</scope>
</dependency>
```

When JNDI is available, Spring Boot will first try to detect the `ConnectionFactory` in the JNDI register under one of the well-known names `java:/JmsXA` and `java:/XAConnectionFactory` or by the name specified in the `spring.jms.jndi-name` property. Furthermore, it will also automatically create a `JndiDestinationResolver` so that the queues and topics will also be detected in JNDI; by default a fallback to dynamic creation of destinations is allowed.

```
spring.jms.jndi-name=java:/jms/connectionFactory
```

With that in place and having built your WAR (see Recipe 11.2), you can now deploy your application to the JEE container and reuse the existing `ConnectionFactory`.

Manual Configuration

The final way of configuring JMS is to do manual configuration. For this you will at least need the `spring-jms` and `javax.jms-api` dependencies and probably some client libraries for the JMS broker you are using. Manual configuration can be needed when:

1. Spring Boot cannot auto-configure your `ConnectionFactory`.
2. Extensive setup of the `ConnectionFactory` is needed.
3. Multiple `ConnectionFactory` instances are needed.

To configure a `ConnectionFactory`, you can add a `@Bean` annotated method that constructs one.

```
@Bean
public ConnectionFactory connectionFactory() {
    var connectionFactory = new ActiveMQConnectionFactory("vm://localhost?
broker.persistent=false");
    connectionFactory.setClientId("someId");
    connectionFactory.setCloseTimeout(125);
    return connectionFactory;
}
```

This will create a `ConnectionFactory` for ActiveMQ; it will use an embedded nonpersistent broker and set the `clientId` and `closeTimeout`. When Spring Boot detects a preconfigured `ConnectionFactory`, it doesn't attempt to create one itself.

9-2 Send Messages Using JMS

Problem

You want to send messages to other systems over JMS.

Solution

Use the Spring Boot-provided `JmsTemplate` to send and (optionally) convert messages.

How It Works

When using Spring Boot, when it detects JMS and a single `ConnectionFactory` it will also automatically configure a `JmsTemplate`, which can be used to send and convert messages. Spring Boot exposes properties in the `spring.jms.template` namespace, which can be used to configure the `JmsTemplate`.

Send a Message with `JmsTemplate`

To send a message through JMS, you can use the `send` or `sendAndConvert` methods on the `JmsTemplate`. Let's write a component that places a message with the current date and time on a queue every second.

```
@Component
class MessageSender {

    private final JmsTemplate jms;

    MessageSender(JmsTemplate jms) {
        this.jms = jms;
    }

    @Scheduled(fixedRate = 1000)
    public void sendTime() {
        jms.convertAndSend("time-queue", "Current Date & Time is: " +
            LocalDateTime.now());
    }
}
```

The `JmsTemplate` is automatically injected through the constructor, and due to scheduling we will get a message containing the current date and time on a queue called `time-queue`. To run this code, you will need an `@SpringBootApplication` class with the `@EnableScheduling` annotation so that the `@Scheduled` will be processed.

```
@SpringBootApplication
@EnableScheduling
public class JmsSenderApplication {

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

Now when running this class it appears as if nothing much is happening, but the messages are filling up the queue. We can write a simple integration test to check if this code is working.

```
package com.apress.springbootrecipes.demo;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.test.context.junit4.SpringRunner;

import javax.jms.JMSEException;
import javax.jms.Message;
import javax.jms.TextMessage;

import static org.assertj.core.api.Assertions.assertThat;

@RunWith(SpringRunner.class)
@SpringBootTest
public class JmsSenderApplicationTest {

    @Autowired
    private JmsTemplate jms;
```

```
@Test  
public void shouldSendMessage() throws JMSException {  
  
    Message message = jms.receive("time-queue");  
  
    assertThat(message)  
        .isInstanceOf(TextMessage.class);  
    assertThat(((TextMessage) message).getText())  
        .startsWith("Current Date & Time is: ");  
}  
}
```

This JUnit test will bootstrap the application and starts sending messages. In the test, we use a `JmsTemplate` to receive a message and we do some assertions on it to see if messages are really sent and contain what we expected them to contain. The test will use the embedded JMS broker, as we haven't configured anything. When running the test, it should be green as a message will be sent and received.

Tip When writing tests for messaging, you might want to set the `receive-timeout` property of the `JmsTemplate`, as the default is to indefinitely wait for a message to come; however, after 500ms you might want to fail your test. You can do this by adding `spring.jms.template.receive-timeout=500ms` to your `application.properties`.

Configure the `JmsTemplate`

Spring Boot provides properties in the `spring.jms.template` namespace to configure the `JmsTemplate` (Table 9-4).

Table 9-4. *JmsTemplate Properties*

Property	Description
spring.jms.template.default-destination	Default destination to use for send and receive operations when a specific destination isn't specified
spring.jms.template.delivery-delay	Delivery delay for sending a message
spring.jms.template.delivery-mode	Delivery mode, persistent or non-persistent, when explicitly set sets qos-enabled to true
spring.jms.template.priority	Priority of the message when sending. Default none, when explicitly set sets qos-enabled to true
spring.jms.template.qos-enabled	Should QOS (Quality of Service) be enabled. When enabled the priority, delivery-mode, and time-to-live of a message will be set. Default false
spring.jms.template.receive-timeout	Timeout to use for receive calls. Default indefinite
spring.jms.template.time-to-live	Time-to-live of a JMS message, when set sets qos-enabled to true
spring.jms.pub-sub-domain	Default destination is a topic or queue. Default false meaning queue

Next to these properties the `JmsTemplate` will also be auto-configured with a `DestinationResolver` and `MessageConverter` if a unique instance of the beans can be found. If no unique instance can be found, the defaults will be used, `DynamicDestinationResolver` and `SimpleMessageConverter` (Table 9-5).

Table 9-5. *SimpleMessageConverter Class to JMS Message Converter*

Type	JMS Message Type
<code>java.lang.String</code>	<code>javax.jms.TextMessage</code>
<code>java.util.Map</code>	<code>javax.jms.MapMessage</code>
<code>java.io.Serializable</code>	<code>javax.jms.ObjectMessage</code>
<code>byte[]</code>	<code>javax.jms.BytesMessage</code>

Let's send an Order to the orders queue. Use Jackson to send JSON instead of using the Java Serialization mechanism.

```
public class Order {

    private String id;
    private BigDecimal amount;

    public Order() {
    }

    public Order(String id, BigDecimal amount) {
        this.id=id;
        this.amount = amount;
    }

    // Getters / Setters omitted for brevity

    @Override
    public String toString() {
        return String.format("Order [id='%s', amount=%4.2f]", id, amount);
    }
}
```

That is the simple order we are going to send.

Now we need a sender that constructs an Order and places it on the queue using a JmsTemplate.

```
@Component
class OrderSender {

    private final JmsTemplate jms;

    OrderSender(JmsTemplate jms) {
        this.jms = jms;
    }

    @Scheduled(fixedRate = 1000)
    public void sendTime() {
```

```

    var id = UUID.randomUUID().toString();
    var amount = ThreadLocalRandom.current().nextDouble(1000.00d);
    var order = new Order(id, BigDecimal.valueOf(amount));
    jms.convertAndSend("orders", order);
}
}

```

It isn't that different from the `MessageSender` written earlier, but now it creates an `Order` with some random generated data and sends it on the `orders` queue. When running this, it would actually fail. Converting fails, as `Order` doesn't implement `Serializable`, which is needed to convert the `Order` into an `ObjectMessage` (see Table 9-5). However, we wanted to use JSON to make this happen; a different `MessageConverter` is needed, the `MappingJackson2MessageConverter` to be exact. This uses Jackson to marshal and unmarshal objects to JSON. It should be added as a bean to the configuration.

First you will need to add the dependency on Jackson to your build configuration.

```

<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-core</artifactId>
</dependency>

```

Next you can configure the `MappingJackson2MessageConverter`.

```

@SpringBootApplication
@EnableScheduling
public class JmsSenderApplication {

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

    @Bean
    public MappingJackson2MessageConverter messageConverter() {
        var messageConverter = new MappingJackson2MessageConverter();
        messageConverter.setTypeIdPropertyName("content-type");
    }
}

```

```

        messageConverter.setTypeIdMappings(
            Collections.singletonMap("order", Order.class));
    return messageConverter;
}
}

```

The typeIdPropertyName is a required property and indicates the name of the property in which the actual type of the message is stored. Without any further configuration, the FQN of the class will be used. With the typeIdMappings you can specify which type to map to which class and vice versa. When not specified, the FQN of a class will be used as the type in the mapping. When sending an Order, the content-type header will contain the value order.

Tip It generally is a good idea to explicitly define type mappings. This way you don't explicitly bind two or more applications together on the Java level. They can use their own mapping for order to their own Order class.

With all this in place, we can write a test to see if orders are actually being sent.

```

@RunWith(SpringRunner.class)
@SpringBootTest
public class JmsSenderApplicationTest {

    @Autowired
    private JmsTemplate jms;

    @Test
    public void shouldReceiveOrderPlain() throws Exception {
        Message message = jms.receive("orders");
        assertThat(message)
            .isInstanceOf(BytesMessage.class);

        BytesMessage msg = (BytesMessage) message;
        ObjectMapper mapper = new ObjectMapper();
        byte[] content = new byte[(int) msg.getBodyLength()];
        msg.readBytes(content);
    }
}

```

```

Order order = mapper.readValue( content, Order.class);
assertThat(order).hasNoNullFieldsOrProperties();
}

@Test
public void shouldReceiveOrderWithConversion() throws Exception {

    Order order = (Order) jms.receiveAndConvert("orders");
    System.out.println(order);

    assertThat(order).hasNoNullFieldsOrProperties();
}
}

```

There are two test methods here: the plain one does manual conversion of the message to an `Order`, whereas the second uses the `receiveAndConvert` method to have the conversion done for you. This to show what the `MessageConverter` is doing and that it makes your code more readable. The `MappingJackson2MessageConverter` converts the `Order` into a `BytesMessage`. To use a `TextMessage`, you can set the `targetType` property to `TEXT`. You will then receive a `TextMessage` with a JSON as a `String` as the payload.

9-3 Receive Messages Using JMS

Problem

You want to read messages from a JMS destination so that you can handle them in your application.

Solution

Create a class and annotate methods with `@JmsListener` to bind it to a destination and handle incoming messages.

How It Works

You can create a POJO and annotate its methods with `@JmsListener`. Spring will detect this and create a JMS listener for it. Spring Boot exposes properties to configure the listeners under the `spring.jms.listener` namespace.

Receiving Messages

Let's create a service that listens to the message sent by the sender from Recipe 9.2.

```
@Component
class CurrentDateTimeService {

    @JmsListener(destination = "time-queue")
    public void handle(Message msg) throws JMSException {
        Assert.state(msg instanceof TextMessage, "Can only handle
TextMessage.");
        System.out.println("[RECEIVED] - " + ((TextMessage) msg).getText());
    }
}
```

It is a regular class and has a method annotated with `@JmsListener`, which requires at least a destination so that it knows where to retrieve messages from. It accepts a `javax.jms.Message`, which we validate to be a `TextMessage` and print the content to the console. The method signature of a `@JmsListener` annotated method is somewhat flexible, as it allows for several arguments either annotated or from a specific type (Table 9-6).

Table 9-6. Allowed Method Parameter Types

Type	Description
<code>java.lang.String</code>	Get message payload as String only for <code>TextMessage</code>
<code>java.util.Map</code>	Get message payload as Map only for <code>MapMessage</code>
<code>byte[]</code>	Get message payload as <code>byte[]</code> only for <code>BytesMessage</code>
Serializable object	Deserialize Object from <code>ObjectMessage</code>
<code>javax.jms.Message</code>	To get the actual JMS message
<code>javax.jms.Session</code>	To access the Session for access, sending a custom reply for instance
<code>@Header annotated element</code>	Extract a header from the JMS message
<code>@Headers annotated element</code>	Only usable on a <code>java.util.Map</code> to get all the JMS message headers

The listener could be simplified by simply using a `String` as a method argument instead of handling the `javax.jms.Message` ourselves.

```
@Component
class CurrentDateTimeService {

    @JmsListener(destination = "time-queue")
    public void handle(String msg) {
        System.out.println("[RECEIVED] - " + msg);
    }
}
```

Configure the Listener Container

Spring uses a `JmsListenerContainerFactory` to create the infrastructure needed to support the `@JmsListener` annotation. Spring Boot configures a default one, which can be configured using properties from the `spring.jms.listener` namespace. If that doesn't suffice, you can always configure your own instance and do all the configuration options manually. Spring Boot will then refrain from creating `JmsListenerContainerFactory` when it already detects one in the context.

Table 9-7. Listener Container Properties

Property	Description
<code>spring.jms.listener.acknowledge-mode</code>	Acknowledge mode of the container, default is automatic.
<code>spring.jms.listener.auto-startup</code>	Start the container automatically on startup. Default true
<code>spring.jms.listener.concurrency</code>	Minimum number of concurrent consumers. Default is none, leading to 1 concurrent consumer (the Spring default).
<code>spring.jms.listener.max-concurrency</code>	Maximum number of concurrent consumers. Default is none, leading to 1 concurrent consumer (the Spring default).
<code>spring.jms.pub-sub-domain</code>	Default destination is a topic. Default false meaning queue.

The default configured `JmsListenerContainerFactory` will also detect a single, unique `DestinationResolver` and `MessageConverter` and when found will use that; otherwise it will use the Spring defaults `DynamicDestinationResolver` and `SimpleMessageConverter` (see recipe 9.2 for more information).

Use Custom MessageConverter

What if you want to send objects as JSON to the next system over JMS? You can rely on Java Serialization, but that generally is frowned upon because that tightly couples systems. Using JSON or XML to transfer objects/messages is a better way. With Spring JMS it is a matter of configuring a different `MessageConverter` (see also Recipe 9.2 on the sending part).

```
@Bean
public MappingJackson2MessageConverter messageConverter() {

    var messageConverter = new MappingJackson2MessageConverter();
    messageConverter.setContentTypePropertyName("content-type");
    messageConverter.setTypeIdMappings(singletonMap("order", Order.class));
    return messageConverter;
}
```

The `MappingJackson2MessageConverter` by default requires a property name to put in the identifier for the content type. This will be read from a header in the JMS message (here we set it to `content-type`). Next we can, optionally, define a mapping between a type and the class. As we want to be able to map objects to the `Order` class, we specify that as the mapping for the content-type `order`.

```
@Component
class OrderService {

    @JmsListener(destination = "orders")
    public void handle(Order order) {
        System.out.println("[RECEIVED] - " + order);
    }
}
```

The listener receives the Order object, as Spring JMS will take care of receiving and converting the messages. If you combine this listener with the order sender from Recipe 9.2, you will see a steady stream of orders coming in.

Sending a Reply

Sometimes when receiving a message, you want to return an answer or trigger another part of the process. With Spring Messaging this is pretty easy: you can simply return what you want to send from your handler method. To determine where to send the response to, you can add an additional `@SendTo` annotation to specify the destination. Let's modify the sample to send an OrderConfirmation to the order-confirmations queue.

```
@Component
class OrderService {

    @JmsListener(destination = "orders")
    @SendTo("order-confirmations")
    public OrderConfirmation handle(Order order) {

        System.out.println("[RECEIVED] - " + order);
        return new OrderConfirmation(order.getId());
    }
}
```

The OrderService changed slightly; it now returns an OrderConfirmation after processing the order. With the `@SendTo` annotation, we specify which destination to put the result on.

Let's create another listener for the OrderConfirmation object so that we can see them coming in.

```
@Component
class OrderConfirmationService {

    @JmsListener(destination = "order-confirmations")
    public void handle(OrderConfirmation confirmation) {
        System.out.println("[RECEIVED] - " + confirmation);
    }
}
```

As last the OrderConfirmation class. This will be constructed from the incoming message.

```
public class OrderConfirmation {
    private String orderId;
    public OrderConfirmation() {}
    public OrderConfirmation(String orderId) {
        this.orderId = orderId;
    }
    public String getOrderId() {
        return orderId;
    }
    public void setOrderId(String orderId) {
        this.orderId = orderId;
    }
    @Override
    public String toString() {
        return String.format("OrderConfirmation [orderId='%s']", orderId);
    }
}
```

When running the application, you will see that first the order is received and next that the OrderConfirmation is being received.

9-4 Configure RabbitMQ

Problem

You want to use AMQP messaging in a Spring Boot application and need to connect to the RabbitMQ broker.

Solution

Configure the appropriate `spring.rabbitmq` properties (minimal `spring.rabbitmq.host`) to connect to the exchange and be able to send and receive messages.

How It Works

Spring Boot will automatically create a `ConnectionFactory` when it detects the RabbitMQ client library on the classpath. To get started, you need to add the `spring-boot-starter-amqp` dependencies; this will pull in all the required dependencies.

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

Now you can use the `spring.rabbitmq` properties to connect to the broker.

```
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=guest
spring.rabbitmq.password=guest
```

The preceding configuration is all that is needed to connect to a default instance of RabbitMQ and are also the defaults as used by Spring Boot.

Table 9-8. Common RabbitMQ Properties

Property	Description
<code>spring.rabbitmq.addresses</code>	Comma-separated list of addresses to which the client should connect
<code>spring.rabbitmq.connection-timeout</code>	Connection timeout. Default none, 0 never timeout
<code>spring.rabbitmq.host</code>	The RabbitMQ Host, default localhost
<code>spring.rabbitmq.port</code>	The RabbitMQ port, default 5672
<code>spring.rabbitmq.username</code>	Username to use for connecting, default guest
<code>spring.rabbitmq.password</code>	Password to use for connecting, default guest
<code>spring.rabbitmq.virtual-host</code>	Virtual host to use when connecting to the broker

9-5 Send Messages Using RabbitMQ

Problem

You want to send a message to a RabbitMQ broker so that the message can be delivered to the receiver.

Solution

Using the `RabbitTemplate`, you can send messages to an exchange and provide a routing key.

How It Works

Spring Boot automatically configures a `RabbitTemplate` when it finds a unique `ConnectionFactory`; this template can be used to send a message to a queue.

Configure the `RabbitTemplate`

Spring Boot will automatically configure a `RabbitTemplate` if it can find a unique `ConnectionFactory` and if no `RabbitTemplate` exists in the configuration. Spring Boot allows us to modify the configured `RabbitTemplate` through properties in the `spring.rabbitmq.template` namespace (Table 9-9).

Table 9-9. *RabbitTemplate Configuration Properties*

Property	Description
<code>spring.rabbitmq.template.exchange</code>	Name of the default exchange to use for send operations, default none
<code>spring.rabbitmq.template.routing-key</code>	Value of a default routing key to use for send operations, default none
<code>spring.rabbitmq.template.receive-timeout</code>	Timeout for receive operations. Default 0, no timeout
<code>spring.rabbitmq.template.reply-timeout</code>	Timeout for send-and-receive operations. Default 5 seconds

Spring Boot also makes it easy to configure retry logic with the `RabbitTemplate`; by default it is disabled. By putting `spring.rabbitmq.template.retry.enabled=true` in the `application.properties` it will be enabled. Now when sending fails, it will try an additional two times to send the message. To change the number of retries or the interval, you can use properties from the `spring.rabbitmq.template.retry` namespace (Table 9-10).

Table 9-10. *RabbitTemplate Retry Configuration*

Property	Description
<code>spring.rabbitmq.template.retry.enabled</code>	Publishing retries enabled, default false
<code>spring.rabbitmq.template.retry.max-attempts</code>	Number of attempts to deliver a message, default 3.
<code>spring.rabbitmq.template.retry.initial-interval</code>	Duration between the first and second publishing attempt, default 1 second
<code>spring.rabbitmq.template.retry.max-interval</code>	Number of attempts to deliver a message, default 10 seconds
<code>spring.rabbitmq.template.retry.multiplier</code>	Multiplier to apply to the previous interval, default 1.0

Sending a Simple Message

Sending a message with the `RabbitTemplate` can be done by the `convertAndSend` method. It takes, at least, the routing key and the object to send in the message.

```
@Component
class HelloWorldSender {

    private final RabbitTemplate rabbit;

    HelloWorldSender(RabbitTemplate rabbit) {
        this.rabbit = rabbit;
    }
}
```

```

@Scheduled(fixedRate = 500)
public void sendTime() {
    rabbit.convertAndSend("hello",
        "Hello World, from Spring Boot 2, over
        RabbitMQ!");
}
}

```

The `HelloWorldSender` will be injected with the `RabbitTemplate` through the constructor. Every 500ms a message will be sent on the default exchange with the `hello` routing key. As it is sent to the default exchange, a queue with the name `hello` will be automatically created. You can check in the RabbitMQ management console (default `http://localhost:15672`) the number of messages on the queue.

Write a test to verify the correct behavior of the application. As there isn't an embedded broker for RabbitMQ, you need to mock the `RabbitTemplate` with `@MockBean`. In the `@Test` method, validate the method call with the proper arguments.

```

@RunWith(SpringRunner.class)
@SpringBootTest
public class RabbitSenderApplicationTest {

    @MockBean
    private RabbitTemplate rabbitTemplate;

    @Test
    public void shouldSendAtLeastASingleMessage() {
        verify(rabbitTemplate, Mockito.atLeastOnce())
            .convertAndSend("hello",
                "Hello World, from Spring Boot 2, over RabbitMQ!");
    }
}

```

Tip There are ways of using Docker or an embedded process³ to bootstrap RabbitMQ. The source code includes a test with an embedded process.

³<https://github.com/AlejandroRivera/embedded-rabbitmq>

Sending an Object

To send a message to RabbitMQ, the message payload has to be converted into a `byte[]`. For a `String` that is pretty easy by calling `String.getBytes`. However, when sending an object this becomes more cumbersome. The default implementation will check if the object is `Serializable` and if so will use Java serialization to convert the object to a `byte[]`. Using Java serialization isn't the best solution, especially if you need to send messages to non-Java clients.

The `RabbitTemplate` uses a `MessageConverter` to delegate the message creation to. By default it uses the `SimpleMessageConverter`, which implements the strategy outlined above. There are, however, various implementations that use XML (`MarshallingMessageConverter`) or JSON (`Jackson2JsonMessageConverter`) for the actual payload (instead of Java serialization).

Spring Boot will automatically detect the configured `MessageConverter` and use it for both the `RabbitTemplate` as well as the listeners (see Recipe 9.5).

```
@Bean
public Jackson2JsonMessageConverter jsonMessageConverter() {
    return new Jackson2JsonMessageConverter();
}
```

This is enough to change the `SimpleMessageConverter` into a `Jackson2JsonMessageConverter`.

Let's create an `Order` and use the `RabbitTemplate` to send it to the `orders` exchange using a `new-order` routing key.

```
package com.apress.springbootrecipes.demo;

import java.math.BigDecimal;

public class Order {

    private String id;
    private BigDecimal amount;

    public Order() {
    }
}
```

```

public Order(String id, BigDecimal amount) {
    this.id=id;
    this.amount = amount;
}

public String getId() {
    return id;
}

public void setId(String id) {
    this.id = id;
}

public BigDecimal getAmount() {
    return amount;
}

public void setAmount(BigDecimal amount) {
    this.amount = amount;
}

@Override
public String toString() {
    return String.format("Order [id=%s, amount=%4.2f]", id, amount);
}
}

```

Now that we have an order, let's create a scheduled method that periodically sends a message with a random order.

```

@Component
class OrderSender {

    private final RabbitTemplate rabbit;

    OrderSender(RabbitTemplate rabbit) {
        this.rabbit = rabbit;
    }
}

```

```

@Scheduled(fixedRate = 256)
public void sendTime() {
    var id = UUID.randomUUID().toString();
    var amount = ThreadLocalRandom.current().nextDouble(1000.00d);
    var order = new Order(id, BigDecimal.valueOf(amount));
    rabbit.convertAndSend("orders", "new-order", order);
}
}

```

It will create an Order with a random amount of max 1000.00. It will then send it, using the convertAndSend method, to the orders exchange, using the new-order routing key.

The test will create a mock of the RabbitTemplate using @MockBean again and will test the method invocation.

```

@RunWith(SpringRunner.class)
@SpringBootTest
public class RabbitSenderApplicationTest {

    @MockBean
    private RabbitTemplate rabbitTemplate;

    @Test
    public void shouldSendAtLeastASingleMessage() {

        verify(rabbitTemplate, atLeastOnce())
            .convertAndSend(
                eq("orders"),
                eq("new-order"),
                any(Order.class));
    }
}

```

Write an Integration Test

Add a dependency to an embedded RabbitMQ server to easily start it from a test case.

```

<dependency>
    <groupId>io.arivera.oss</groupId>
    <artifactId>embedded-rabbitmq</artifactId>

```

```
<version>1.3.0</version>
<scope>test</scope>
</dependency>
```

Next, create a RabbitSenderApplicationIntegrationTestConfiguration that contains the additional configuration needed to run an integration test.

```
@TestConfiguration
public class RabbitSenderApplicationIntegrationTestConfiguration {

    @Bean(initMethod = "start", destroyMethod = "stop")
    public EmbeddedRabbitMq embeddedRabbitMq() {
        EmbeddedRabbitMqConfig config = new
            EmbeddedRabbitMqConfig.Builder()
                .rabbitMqServerInitializationTimeoutInMillis(10000).build();
        return new EmbeddedRabbitMq(config);
    }

    @Bean
    public Queue newOrderQueue() {
        return QueueBuilder.durable("new-order").build();
    }

    @Bean
    public Exchange ordersExchange() {
        return ExchangeBuilder.topicExchange("orders").durable(true).build();
    }

    @Bean
    public Binding newOrderQueueBinding(Queue queue, Exchange exchange) {
        return BindingBuilder.bind(queue).to(exchange)
            .with("new-order").noargs();
    }
}
```

The configuration includes the embedded RabbitMQ definition, a Queue and Exchange definition, and finally the Binding of the Queue to the Exchange with the `routingKey`.

The queue and bindings are needed to be able to receive the messages, else they would only reside on the exchange (or, depending on the configuration, be discarded).

The integration test will load the application and the additional configuration. It will use the RabbitTemplate to receive the message.

```
@RunWith(SpringRunner.class)
@SpringBootTest(classes = {
    RabbitSenderApplication.class,
    RabbitSenderApplicationIntegrationTestConfiguration.class })
public class RabbitSenderApplicationIntegrationTest {

    @Autowired
    private RabbitTemplate rabbitTemplate;

    @Test
    public void shouldSendAtLeastASingleMessage() {

        Message msg = rabbitTemplate.receive("new-order", 1500);

        assertThat(msg).isNotNull();
        assertThat(msg.getBody()).isNotEmpty();
        assertThat(msg.getMessageProperties().getReceivedExchange())
            .isEqualTo("orders");
        assertThat(msg.getMessageProperties().getReceivedRoutingKey())
            .isEqualTo("new-order");
        assertThat(msg.getMessageProperties().getContentType())
            .isEqualTo(MediaType.APPLICATION_JSON_VALUE);
    }
}
```

This test will load the application and additional configuration class. This is done by specifying the `classes` attribute on the `@SpringBootApplication` annotation. When the test starts it will receive a message from the `new-order` queue defined in the configuration. The received message will be used to do assertions on the message like the encoding (`application/json`), the routing key, etc.

9-6 Receive Messages Using RabbitMQ

Problem

You want to receive messages from a RabbitMQ.

Solution

Annotating a method with `@RabbitListener` will bind it to a queue and let it receive messages.

How It Works

A bean that has `@RabbitListener` annotated methods will be used as a message listener for incoming messages. A message listener container is constructed and the annotated method will receive the incoming message. The message listener container can be configured through properties in the `spring.rabbitmq.listener` namespace.

Table 9-11. Rabbit Listener Properties

Property	Description
<code>spring.rabbitmq.listener.type</code>	The listener container type <code>direct</code> or <code>simple</code> , default is <code>simple</code>
<code>spring.rabbitmq.listener.simple.acknowledge-mode</code>	Container acknowledge mode, default <code>none</code>
<code>spring.rabbitmq.listener.simple.prefetch</code>	Number of messages to be handled in a single request, default <code>none</code>
<code>spring.rabbitmq.listener.simple.default-requeue-rejected</code>	Should rejected deliveries be re-queued
<code>spring.rabbitmq.listener.simple.concurrency</code>	Minimum number of listener invoker threads
<code>spring.rabbitmq.listener.simple.max-concurrency</code>	Maximum number of listener invoker threads

(continued)

Table 9-11. (continued)

Property	Description
spring.rabbitmq.listener.simple.transaction-size	Number of messages processed in a single transaction. For best results should be smaller or equal to the prefetch size
spring.rabbitmq.listener.direct.acknowledge-mode	Container acknowledge mode, default none
spring.rabbitmq.listener.direct.prefetch	Number of messages to be handled in a single request, default none
spring.rabbitmq.listener.direct.default-requeue-rejected	Should rejected deliveries be re-queued
spring.rabbitmq.listener.direct.consumers-per-queue	Number of consumers per queue. Default is 1

Receiving a Simple Message

A component with an `@RabbitListener` annotation is all that is needed to start receiving messages from RabbitMQ.

```

@Component
class HelloWorldReceiver {

    @RabbitListener( queues = "hello")
    public void receive(String msg) {
        System.out.println("Received: " + msg);
    }
}

```

The preceding component will receive all the messages from the `hello` queue and print it to the console. This works fine for simple payloads or if the receiving object can be deserialized from the payload of the message. However, when sending objects or complex messages, one might prefer to use JSON or XML.

Receiving an Object

To receive a more complex object without relying on Java serialization, you need to configure a `MessageConverter` (see also Recipe 9.4). The configured `MessageConverter` will be used by the message listener container to convert incoming payloads into an object required by the `@RabbitListener` annotated method.

```
@Bean
public Jackson2JsonMessageConverter jsonMessageConverter() {
    return new Jackson2JsonMessageConverter();
}
```

To configure the `MessageConverter`, create an `@Bean` annotated method and construct the converter you want to use. Here the converter is a Jackson 2-based one, but there is also one for unmarshalling XML, the `MarshallingMessageConverter`.

```
@Component
class OrderService {

    @RabbitListener(bindings = @QueueBinding(
        exchange = @Exchange(name="orders", type = ExchangeTypes.TOPIC),
        value = @Queue(name = "incoming-orders"),
        key = "new-order"
    ))
    public void handle(Order order) {
        System.out.println("[RECEIVED] - " + order);
    }
}
```

The preceding listener will use the `orders` exchange (which is a topic exchange) and creates a binding for the `incoming-orders` using the `new-order` routing key. When started, the exchange and queue will automatically be created if they don't already exist. The incoming message is converted into an `Order` using the `Jackson2JsonMessageConverter`.

Receiving a Message and Sending a Reply

When receiving a message, it might be necessary to send a response back to the client or communicate with a different message; using an `@RabbitListener` on a nonvoid method is possible. It will create a result message and place it on an exchange with a routing-key; this needs to be specified in the `@SendTo` annotation.

```
@Component
class OrderService {

    @RabbitListener(bindings = @QueueBinding(
        exchange = @Exchange(name="orders", type = ExchangeTypes.TOPIC),
        value = @Queue(name = "incoming-orders"),
        key = "new-order"
    ))
    @SendTo("orders/order-confirmation")
    public OrderConfirmation handle(Order order) {
        System.out.println("[RECEIVED] - " + order);
        return new OrderConfirmation(order.getId());
    }
}
```

When receiving an `Order` after processing it, an `OrderConfirmation` will be sent; the `@SendTo` annotation (from the general Spring Messaging component) contains the exchange and routing key. The part before the / is the exchange, the part afterwards is the routing key, thus `<exchange>/<routing-key>` is the pattern used. An empty exchange or routing key value is possible (or both); in that case, the default configured exchange and routing key will be used. Here it will use the `orders` exchange and use `order-confirmation` as the routing key.

Another listener could be used to process the `OrderConfirmation` messages.

```
@Component
class OrderConfirmationService {

    @RabbitListener(bindings = @QueueBinding(
        exchange = @Exchange(name="orders", type = ExchangeTypes.TOPIC),
        value = @Queue(name = "order-confirmations"),
        key = "order-confirmation"
    ))
```

```
public void handle(OrderConfirmation confirmation) {  
    System.out.println("[RECEIVED] - " + confirmation);  
}  
}
```

It will create a queue `order-confirmations` using the `order-confirmation` routing-key and bind that on the `orders` exchange (just as the `OrderService` created earlier). When running the code, together with the sender from Recipe 9.4 you should receive `Order` instances and see that they will be confirmed as well.

CHAPTER 10

Spring Boot Actuator

When developing an application, you also want to be able to monitor the behavior of the application. Spring Boot makes it very easy to enable that by introducing Spring Boot Actuator. Spring Boot actuator exposes health and metrics from the application to interested parties. This can be over JMX, HTTP, or exported to an external system.

The health endpoints tell something about the health of your application and/or the system it is running on. It will detect if the database is up, report the diskspace, etc. The metrics endpoints expose usage and performance statistics like number of request, the longest request, the fastest, the utilization of your connection pool, etc.

All these metrics can be viewed either through JMX or HTTP when enabled, but can also be automatically exported to an external system like Graphite, InfluxDB, and many others.

10-1 Enable and Configure Spring Boot Actuator Problem

You want to enable health and metrics in your application so that you can monitor the status of the application.

Solution

Add a dependency for the `spring-boot-starter-actuator` to your project and the health and metrics will be enabled and exposed for your application. Additional configuration can be done through properties in the `management` namespace.

How It Works

When adding the `spring-boot-starter-actuator`, Spring Boot will automatically set up the health and metrics based on the beans in the application context. Which health and metrics are exposed depends on the beans and features that are enabled. When a `DataSource` is detected, metrics for that will be collected and exposed as well as health. Spring Boot does this for many components like Hibernate, RabbitMQ, Caches, etc.

To enable the actuator, add the dependency to your application (here we assume Recipe 3.3 sources as a starting point).

<dependency>

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

Now when starting the application, Spring Boot will have the actuator configured and it is accessible through JMX (Figure 10-1, and see Recipe 8.4 on how to use JConsole) and through the web (default under the `/actuator` path)(see Figure 10-2).

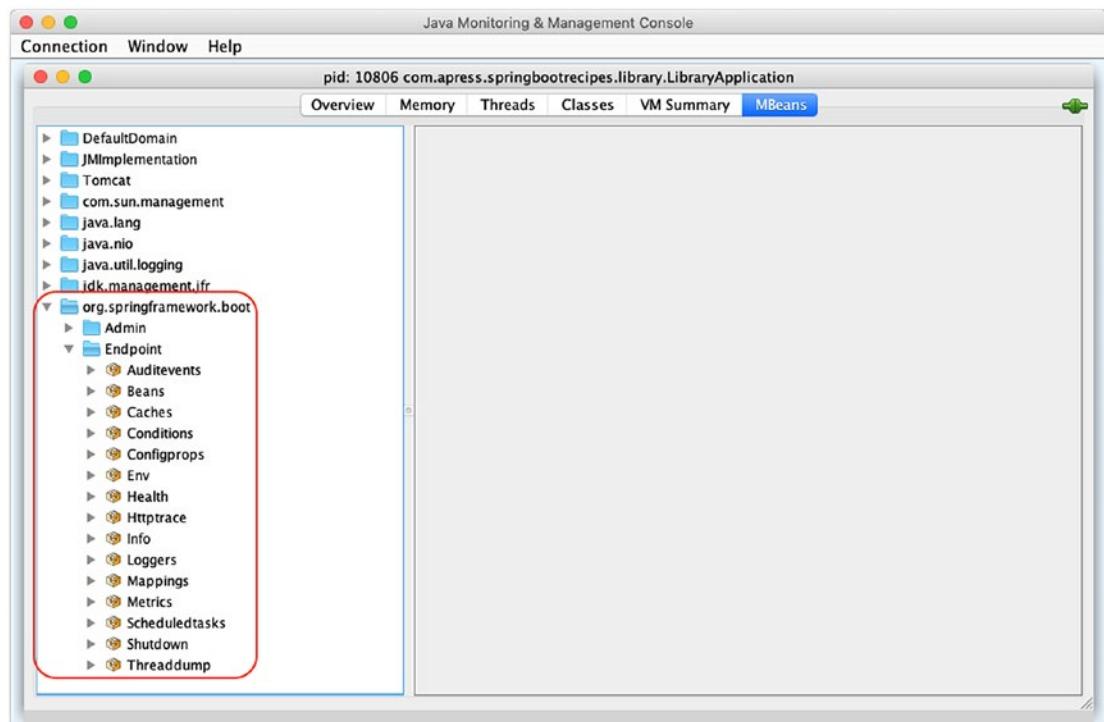
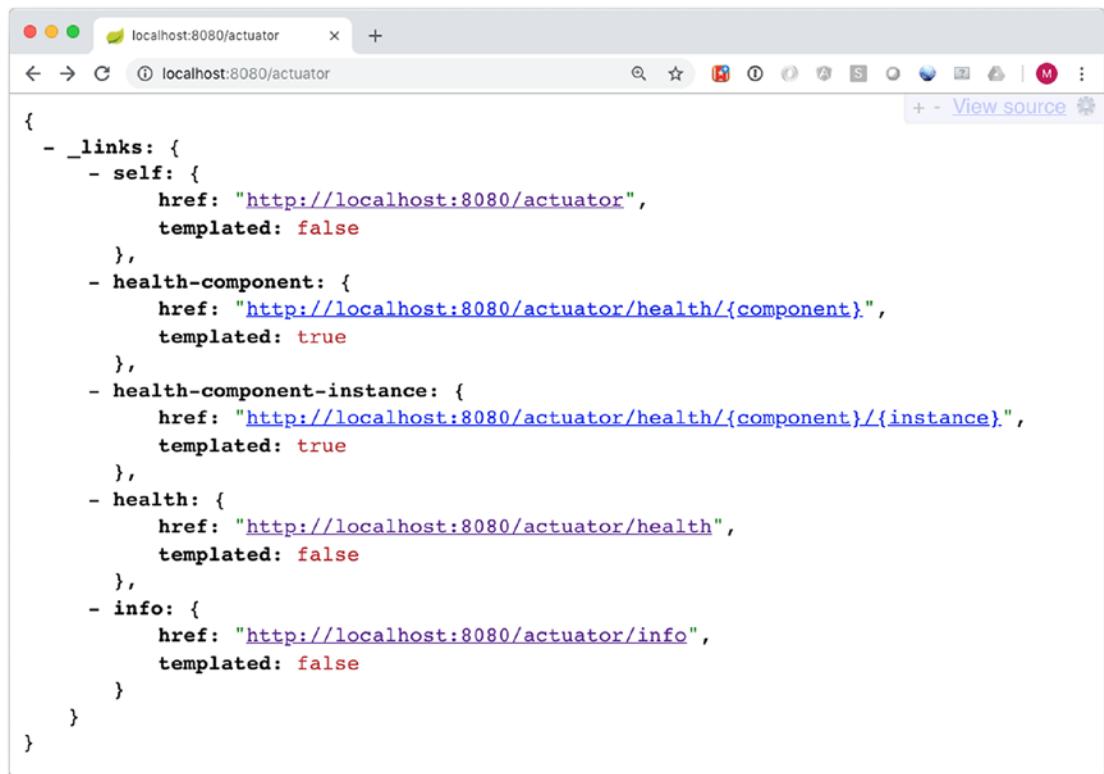


Figure 10-1. JMX exposed metrics



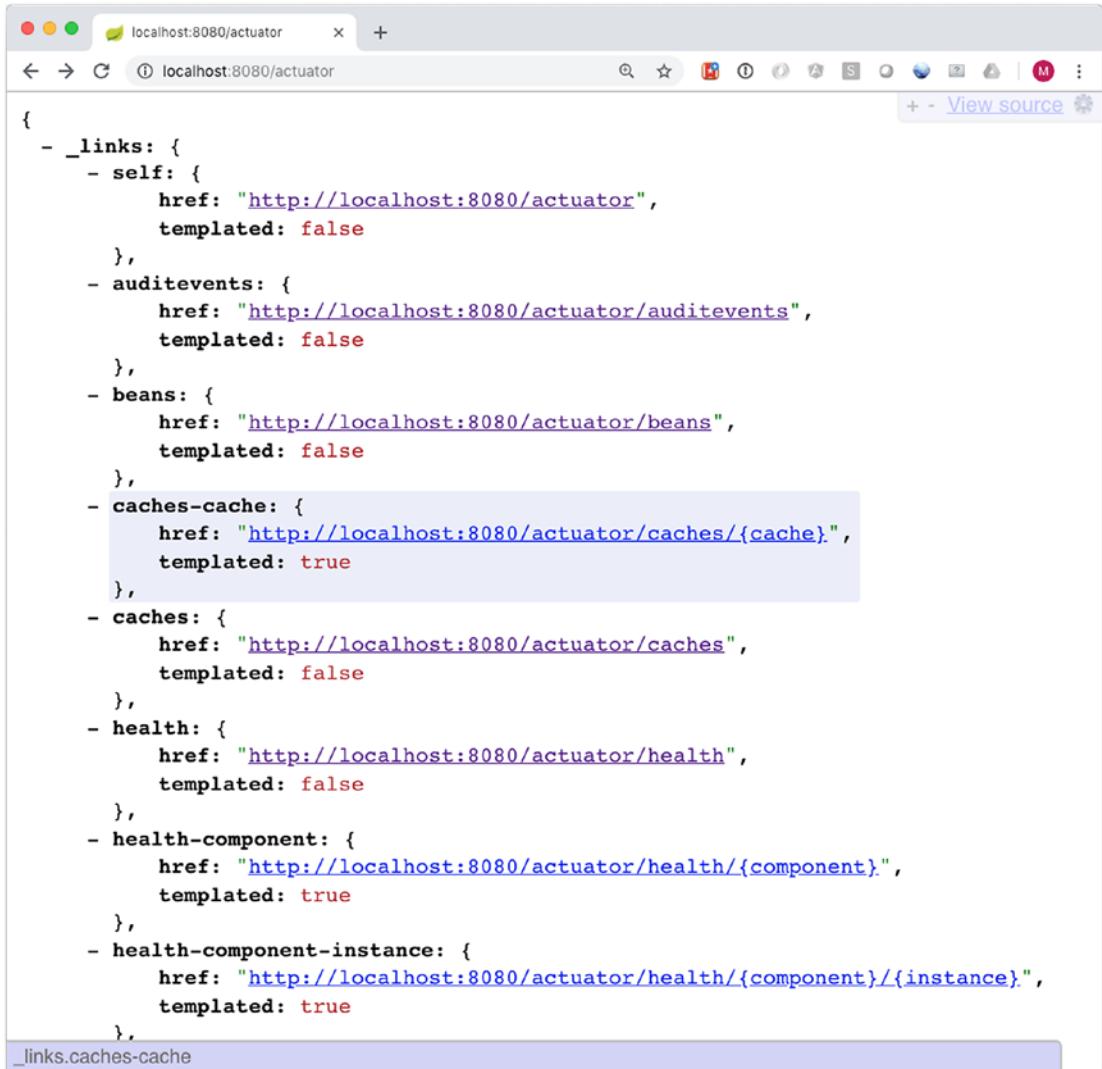
```
{
  "_links": {
    "self": {
      "href": "http://localhost:8080/actuator",
      "templated": false
    },
    "health-component": {
      "href": "http://localhost:8080/actuator/health/{component}",
      "templated": true
    },
    "health-component-instance": {
      "href": "http://localhost:8080/actuator/health/{component}/{instance}",
      "templated": true
    },
    "health": {
      "href": "http://localhost:8080/actuator/health",
      "templated": false
    },
    "info": {
      "href": "http://localhost:8080/actuator/info",
      "templated": false
    }
  }
}
```

Figure 10-2. HTTP exposed metrics

You will notice that JMX exposes a lot more endpoints than HTTP does. HTTP only exposes /actuator/health and /actuator/info. JMX exposed a lot more. This is done with security in mind: the /actuator is exposed publicly and as such you don't want everyone to see things. What to expose can be configured through the management.endpoints.web.exposure.include and management.endpoints.web.exposure.exclude properties. Using a * for the include will expose all endpoints to the web.

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

With the preceding configuration added to the application.properties, the same features will be exposed to the web as through JMX (Figure 10-3).



The screenshot shows a browser window with the URL `localhost:8080/actuator`. The page displays a JSON object representing the available management endpoints. The structure includes links to self, audit events, beans, caches, health, and health components. The 'caches' section is highlighted with a light blue background, and the 'caches-cache' link under it is also highlighted. The bottom status bar shows the path `_links.caches-cache`.

```
{  
  "_links": {  
    "self": {  
      "href": "http://localhost:8080/actuator",  
      "templated": false  
    },  
    "auditevents": {  
      "href": "http://localhost:8080/actuator/auditevents",  
      "templated": false  
    },  
    "beans": {  
      "href": "http://localhost:8080/actuator/beans",  
      "templated": false  
    },  
    "caches-cache": {  
      "href": "http://localhost:8080/actuator/caches/{cache}",  
      "templated": true  
    },  
    "caches": {  
      "href": "http://localhost:8080/actuator/caches",  
      "templated": false  
    },  
    "health": {  
      "href": "http://localhost:8080/actuator/health",  
      "templated": false  
    },  
    "health-component": {  
      "href": "http://localhost:8080/actuator/health/{component}",  
      "templated": true  
    },  
    "health-component-instance": {  
      "href": "http://localhost:8080/actuator/health/{component}/{instance}",  
      "templated": true  
    }  
  },  
  "caches-cache": {}  
}
```

Figure 10-3. HTTP exposed metrics (all)

Configure the Management Server

By default, the actuator is available on the same port and address (`http://localhost:8080`) as the regular application. It is, however, very easy (and common) to run the management endpoints on a different port. This can be configured through properties in the `management.server` namespace; most of them mimic the ones in the regular `server` namespace (Table 10-1).

Table 10-1. Management Server Properties

Property	Description
management.server.add-application-context-header	Add an X-Application-Context header to the response containing the application context name
management.server.port	The port to run the management server on, default same as server.port
management.server.address	The network address to bind to, default same as server.address (which is 0.0.0.0 all addresses)
management.server.servlet.context-path	The context path of the management server, default none resulting in /
management.server.ssl.*	SSL properties to configure SSL for the management server (see Recipe 3.8 on how to configure SSL)

Adding the following to the `application.properties` will run the management endpoints on a separate port and add the X-Application-Context header.

```
management.server.add-application-context-header=true
management.server.port=8090
```

When restarting the application, the management endpoints are now available on `http://localhost:8090/actuator`. Running the actuator on a different port has the benefit of hiding it from the public internet by blocking the port on the firewall and only allowing local access.

Note The `management.server` properties are only effective when using an embedded server; when deploying to an external server these properties don't apply anymore!

Configure Individual Management Endpoints

Individual endpoints can be configured through properties in the management.endpoint.<endpoint-name> namespace. Most of them have at least an enabled property and a cache.time-to-live property. The first will enable or disable the endpoint, the other one specifies how long to cache a result of the endpoint (Table 10-2).

Table 10-2. Endpoint Configuration Properties

Property	Description
management.endpoint.<endpoint-name>.enabled	Whether the specific endpoint is enabled, generally defaults to true and sometimes depends on availability of a feature (i.e., if Flyway isn't present, then enabling the flyway endpoint wouldn't have any effect)
management.endpoint.<endpoint-name>.cache.time-to-live	Time to cache a response, default is 0ms meaning no caching
management.endpoint.health.show-details	Whether to show details for the health endpoint, default is never, can be changed to always or when-authorized
management.endpoint.health.roles	Roles that are allowed to see the details (use together with when-authorized for show-details)

Adding management.endpoint.health.show-details=always to the application.properties will show more information about the health of the application. By default it will only show UP, but now you can see the different parts of the health information for your application (Figure 10-4).

```
{
  "status": "UP",
  "details": {
    "diskSpace": {
      "status": "UP",
      "details": {
        "total": 1120468197376,
        "free": 888696094720,
        "threshold": 10485760
      }
    }
  }
}
```

Figure 10-4. Extended health endpoint output

Securing Management Endpoints

When Spring Boot detects both Spring Boot Actuator and Spring Security, it will enable secure access to management endpoints automatically. When accessing the endpoint, a Basic Login prompt will be shown and asks for a username and password. Spring Boot will generate a default user, with the username user and a generated password (see Recipe 6.1) to use for login.

Adding the `spring-boot-starter-security` in addition to the `spring-boot-starter-actuator` is enough to have secured management endpoints.

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

This will enable security in your application and for the management endpoints. Now when accessing the endpoint `http://localhost:8090/actuator` a Basic login prompt will be shown. After entering the correct credentials you should still be able to see the results.

Configure Healthchecks

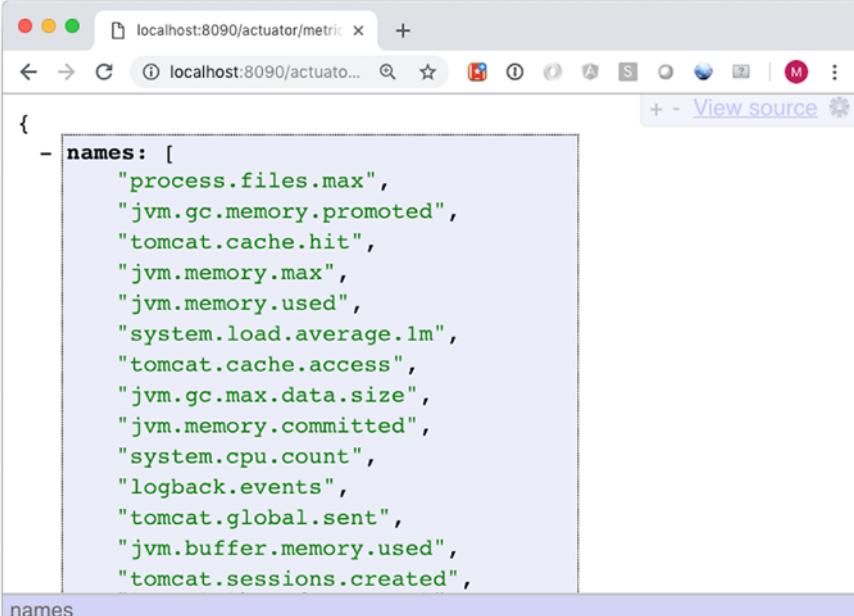
One of the features of Spring Boot Actuator is to do healthchecks. Those are exposed under `http://localhost:8090/actuator/health`; this produces a result if the application is UP or DOWN. The health endpoint calls all the available HealthIndicators in the system and reports those in the endpoint. Which HealthIndicators are present can be controlled by setting the `management.health.<health-indicator>.enabled` property. Setting a property to true for a not available feature (like trying to get information on the DataSource while one isn't available) won't work.

```
management.health.diskspace.enabled=false
```

This will disable the healthcheck for the diskspace, and it won't be part of the health checks anymore.

Configure Metrics

One of the features of Spring Boot Actuator is to expose metrics. Those are exposed under `http://localhost:8090/actuator/metrics`; this will produce a list of available metrics for your application (Figure 10-5).



```
{
  - names: [
    "process.files.max",
    "jvm.gc.memory.promoted",
    "tomcat.cache.hit",
    "jvm.memory.max",
    "jvm.memory.used",
    "system.load.average.1m",
    "tomcat.cache.access",
    "jvm.gc.max.data.size",
    "jvm.memory.committed",
    "system.cpu.count",
    "logback.events",
    "tomcat.global.sent",
    "jvm.buffer.memory.used",
    "tomcat.sessions.created",
  ]
}
```

Figure 10-5. List of currently available metrics

More information about a metric can be obtained by accessing `http://localhost:8090/actuator/metrics/{name-of-metric}`, for example, `http://localhost:8090/actuator/metrics/system.cpu.usage` will show the current CPU usage (Figure 10-6).

The screenshot shows a browser window with the URL `localhost:8090/actuator/metrics/system.cpu.usage`. The page displays a JSON object representing the metric. The JSON structure is as follows:

```
{
  "name": "system.cpu.usage",
  "description": "The \"recent cpu usage\" for the whole system",
  "baseUnit": null,
  "measurements": [
    {
      "statistic": "VALUE",
      "value": 0.4375
    }
  ],
  "availableTags": []
}
```

The word "measurements" is highlighted in blue at the bottom of the JSON object.

Figure 10-6. Detailed CPU metrics

Spring Boot uses micrometer.io¹ to record metrics. Metrics are enabled by default for the features detected by Spring Boot. So if a DataSource is detected, metrics will be enabled. To disable metrics, add them to the `management.metrics.enable` property. This is a map containing keys and values on which metrics to enable.

```
management.metrics.enable.system=false
management.metrics.enable.tomcat=false
```

The preceding configuration will disable `system` and `tomcat` metrics. When viewing the metrics at `http://localhost:8090/actuator/metrics`, they aren't part of the list anymore.

¹<https://micrometer.io>

10-2 Create Custom Health Checks and Metrics

Problem

Your application needs to expose certain metrics and have a health check that aren't available by default.

Solution

The health checks and metrics are pluggable, and beans of type `HealthIndicator` and `MetricBinder` are automatically registered to provide additional health checks and/or metrics. The task is to create a class implementing the desired interface and register an instance of that class as a bean in the context of having it contribute to the health and metrics.

How It Works

Assuming that a dependency on Spring Boot Actuator is already present, you can start writing an implementation right away. Assume you have an application using a `TaskScheduler` and you want to have some metrics and health on it. (Adding `@EnableScheduling` is enough to have Spring Boot create a default `TaskScheduler`.)

First, let's write the `HealthIndicator`. You can either directly implement the `HealthIndicator` interface or use the convenience `AbstractHealthIndicator` as a base class.

```
package com.apress.springboottrecipes.library.actuator;

import org.springframework.boot.actuate.health.AbstractHealthIndicator;
import org.springframework.boot.actuate.health.Health;
import org.springframework.scheduling.concurrent.ThreadPoolTaskScheduler;
import org.springframework.stereotype.Component;

@Component
class TaskSchedulerHealthIndicator extends AbstractHealthIndicator {
```

```

private final ThreadPoolTaskScheduler taskScheduler;

TaskSchedulerHealthIndicator(ThreadPoolTaskScheduler taskScheduler) {
    this.taskScheduler = taskScheduler;
}

@Override
protected void doHealthCheck(Health.Builder builder) throws Exception {

    int poolSize = taskScheduler.getPoolSize();
    int active = taskScheduler.getActiveCount();
    int free = poolSize - active;

    builder
        .withDetail("active", taskScheduler.getActiveCount())
        .withDetail("poolsize", taskScheduler.getPoolSize());

    if (poolSize > 0 && free <= 1) {
        builder.down();
    } else {
        builder.up();
    }
}
}

```

The TaskSchedulerHealthIndicator operates on the given ThreadPoolTaskExecutor. It reports the status as down if there is one or fewer threads available to schedule tasks. The condition on `poolSize > 0` is there because the creation of the underlying Executor is delayed until needed; until then the `poolSize` will report 0. The returned value includes the `poolsize` and `active` thread count just for information.

The TaskSchedulerMetrics implements the MeterBinder interface from micrometer.io. It exposes the active and pool-size to the metrics registry.

```

package com.apress.springbootrecipes.library.actuator;

import io.micrometer.core.instrument.FunctionCounter;
import io.micrometer.core.instrument.MeterRegistry;
import io.micrometer.core.instrument.binder.MeterBinder;
import org.springframework.scheduling.concurrent.ThreadPoolTaskScheduler;
import org.springframework.stereotype.Component;

```

```

@Component
class TaskSchedulerMetrics implements MeterBinder {

    private final ThreadPoolTaskScheduler taskScheduler;

    TaskSchedulerMetrics(ThreadPoolTaskScheduler taskScheduler) {
        this.taskScheduler = taskScheduler;
    }

    @Override
    public void bindTo(MeterRegistry registry) {
        FunctionCounter
            .builder("task.scheduler.active", taskScheduler,
                    ThreadPoolTaskScheduler::getActiveCount)
            .register(registry);

        FunctionCounter
            .builder("task.scheduler.pool-size", taskScheduler,
                    ThreadPoolTaskScheduler::getPoolSize)
            .register(registry);
    }
}

```

Now when placing an `@EnableScheduling` on the `LibraryApplication` and restarting the application, metrics and health will be reported for the `TaskScheduler` (Figure 10-7).



Figure 10-7. TaskScheduler health check

10-3 Export Metrics

Problem

You want to export the metrics to an external system, to create a dashboard to monitor the application.

Solution

Use one of the supported systems like Graphite and periodically push the metrics to that system. Include a micrometer.io registry dependency in your application (next to the `spring-boot-starter-actuator` dependency) and metrics will automatically be exported. By default, every minute the data will be pushed to the server.

How It Works

Exporting metrics is part of the Micrometer.io library and it supports a wide variety of services like Graphite, DataDog, Ganglia, or regular StatsD. This recipe uses Graphite, so a dependency to `micrometer-registry-graphite` needs to be added.

```
<dependency>
<groupId>io.micrometer</groupId>
<artifactId>micrometer-registry-graphite</artifactId>
</dependency>
```

In theory this could be enough to have metrics published to Graphite if Graphite (<https://graphiteapp.org>) is running on localhost and with the default ports. However, Spring Boot makes it easy to configure this by exposing some properties, generally in the `management.metrics.export.<registry-name>` namespace (Table 10-3).

Table 10-3. Common Metrics Export Properties

Property	Description
management.metrics.export.<registry-name>.enabled	Whether or not to enable metrics exporting. Default true when the library is detected on the classpath
management.metrics.export.<registry-name>.host	Host to send metrics to, mostly localhost or well-known url of service (like SignalFX, DataDog, etc.)
management.metrics.export.<registry-name>.port	Port to send metrics to, defaults to the well-known port of the desired service
management.metrics.export.<registry-name>.step	How often to send metrics, default 1 minute
management.metrics.export.<registry-name>.rate-units	Base time unit used to report rates, default seconds
management.metrics.export.<registry-name>.duration-units	Base time unit used to report durations, default milliseconds

To report the metrics every ten seconds instead of each minute, add the following to the application.properties:

```
management.metrics.export.graphite.step=10s
```

Note The bin directory contains a graphite.sh, which uses docker to start a Graphite instance.

Now the metrics will be published to Graphite every ten seconds. If you start the application and open Graphite on <http://localhost> (assuming you are running the aforementioned Docker container) you could, for example, create a graph of the CPU usage (Figure 10-8).

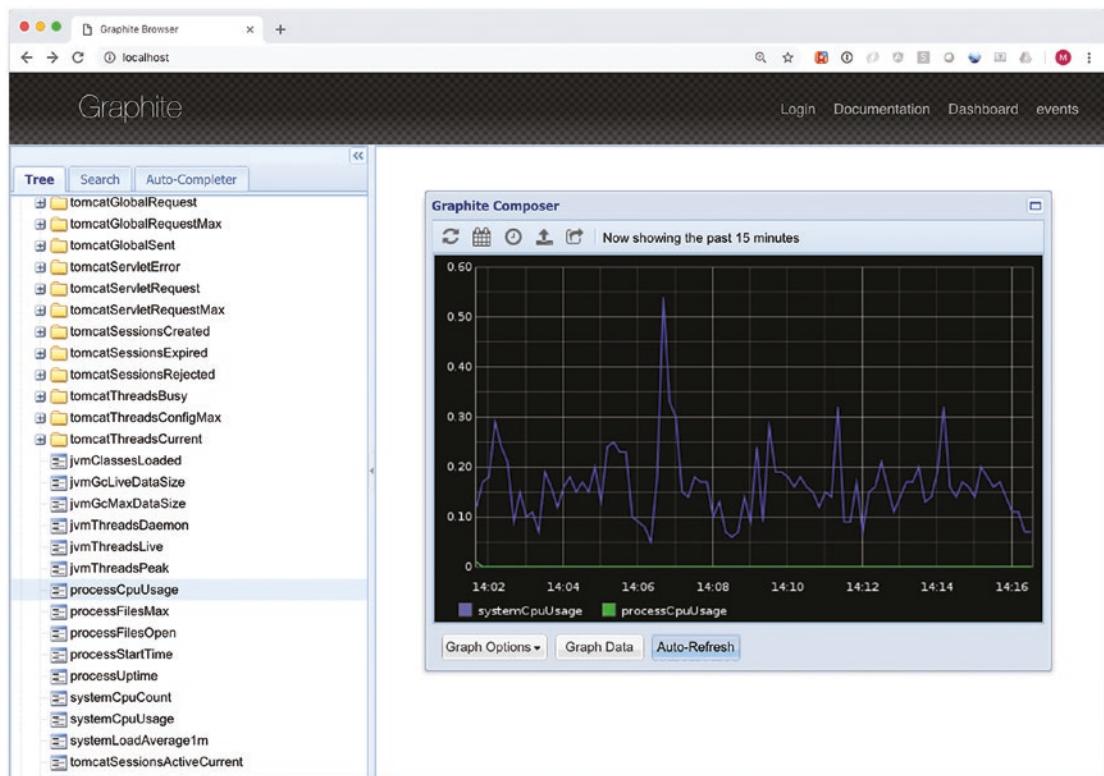


Figure 10-8. Graphite CPU graph

CHAPTER 11

Packaging

In this chapter you will take a look at different solutions for packaging a Spring Boot based application.

11-1 Create an Executable Archive

By default, Spring Boot creates a JAR or WAR and that can be run with `java -jar your-application.jar`. However, you might want to run the application as part of the startup of your server (currently tested and supported for Debian and Ubuntu-based systems). For this you can use the Maven or Gradle plugins to create an executable jar.

Problem

You want an executable JAR so that it can be installed as service on your environment.

Solution

The Spring Boot Maven and Gradle plugins both have the option to make the created artifact executable.¹ When doing so, the archive also becomes/behave like a Unix shell script to start/stop a service.

How It Works

To make an artifact executable, you would need to configure the Maven or Gradle plugin as such.

¹<https://docs.spring.io/spring-boot/docs/current/reference/html/deployment-install.html>

Making the Archive Executable

```
<plugin>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-maven-plugin</artifactId>
  <configuration>
    <executable>true</executable>
  </configuration>
</plugin>
```

In the Maven plugin you can set the executable property to true and the archive will be executable.

```
bootJar {
  launchScript()
}
```

Now after building your artifact, the artifact itself is executable and can be used to launch the application. Instead of `java -jar your-application.jar` you can now simply type `./your-application.jar` and it will start.

Note It might be necessary to make the archive executable; use `chmod +x your-application.jar` for that.

When making the archive executable, the archive will be prefixed with a bash script. You might think that doing so will break the Java archive. However, due to the way Java reads archives (bottom to top) and the way the shell reads the archives (top to bottom) this actually does work.

You can see the bash script using `head -n 290 your-application.jar`.

This is the header of the script the Java archive will be prefixed with. The description will be filled with the project description as given in Maven. You could override this by specifying the `initInfoDescription` next to the executable property.

Specifying Configuration

Normally when you launch a Spring Boot-based application, there are several options for providing additional configuration (see Recipe 2.2). However, when using the archive as a script (or as a service), some of them don't apply anymore. What you can do instead is make use of a `.conf` file next to your executable archive (it must be named `your-application.conf`) to contain additional configuration options for your application (See Table 11-1).

Table 11-1. Available Properties

Property	Description
MODE	The “mode” of operation. Default is auto, which will detect the mode; when launched from a symlink will behave like service. Change to run if you want to run the process in the foreground.
USE_START_STOP_DAEMON	Whether to use the start-stop-daemon command or not. By default will detect if the command is available
PID_FOLDER	Name of the folder to write the PID to, default /var/run
LOG_FOLDER	Name of the folder to write the logging to, default /var/log
CONF_FOLDER	Name of the folder to read .conf from, default the same directory as the jar file
LOG_FILENAME	Name of the logfile to write to, default <appname>.log.
APP_NAME	The name of the application. If the jar is run from a symlink, the script guesses the app name.
RUN_ARGS	The arguments to pass to the Spring Boot application
JAVA_HOME	By default will be discovered from the \$PATH but can be explicitly defined if needed
JAVA_OPTS	Options to pass to the JVM (like memory settings, GC settings, etc.)
JARFILE	The explicit location of the JAR file, in case the script is being used to launch a JAR that it is not actually embedded
DEBUG	If not empty, sets the -x flag on the shell process, making it easy to see the logic in the script
STOP_WAIT_TIME	The time to wait before forcing a shutdown (default is 60 seconds)

When using an embedded script (the default) the JARFILE and APP_NAME properties aren't configurable using a .conf file.

```
JAVA_OPTS=-Xmx1024m
DEBUG=true
```

With this, you will assign a maximum of 1GB of memory to your application and it will do some additional logging for the shell script.

11-2 Create a WAR for Deployment Problem

Instead of creating a JAR file, you want a WAR file for deployment to a Servlet Container or JEE Container.

Solution

Change the packaging of the application from JAR to WAR and let your Spring Boot application extend the `SpringBootServletInitializer` so that it can bootstrap itself as a regular application.

How It Works

To make Recipe 3.1 into a deployable WAR, three things need to be done:

1. Change the packaging from JAR to WAR.
2. Extend `SpringBootServletInitializer` to bootstrap the application on deployment.
3. Change the scope of the embedded server to provided.

In the `pom.xml` change the packaging from JAR to WAR.

<packaging>war</packaging>

To prevent the embedded container from adding its classes to the web application, the scope of the embedded container needs to be changed to `provided` instead of the default `compile`.

When using an embedded Tomcat (the default), add the following to the `pom.xml`.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-tomcat</artifactId>
  <scope>provided</scope>
</dependency>
```

When using a different embedded container, see Recipe 3.7, then change the scope of that container to provided by including `<scope>provided</scope>`. When doing so, the Spring Boot plugin will not add the libraries to the default WEB-INF/lib. However, they will still be part of the created WAR file, but in the special WEB-INF/lib-provided directory. Spring Boot knows this location, and as such you can also use the WAR to start the embedded container. Starting an embedded container is very nice for development; however, as it is a WAR file it is also deployable to a container like Tomcat or WebSphere.

To make sure that the application will start, you need your application to extend the `SpringBootServletInitializer`. This is a special class needed to bootstrap Spring Boot in a Servlet or JEE container.

```
package com.apress.springbootrecipes.helloworld;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.builder.SpringApplicationBuilder;
import org.springframework.boot.web.servlet.support.
SpringBootServletInitializer;

@SpringBootApplication
public class HelloWorldApplication extends SpringBootServletInitializer {

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

    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder
builder) {
        return builder.sources(HelloWorldApplication.class);
    }
}
```

When extending the `SpringBootServletInitializer`, you need to override the `configure` method. The `configure` method gets a `SpringApplicationBuilder`, which you can use to configure the application. One of the things to add is the main configuration class, just as with `SpringApplication.run`. That is what the line `builder.sources(HelloWorldApplication.class)` does. This will be used to bootstrap the application.

For completeness, the `HelloWorldController` as used in Recipe 3.7.

```
package com.apress.springbootrecipes.helloworld;

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

@RestController
public class HelloWorldController {

    @GetMapping
    public String hello() {
        return "Hello World, from Spring Boot 2!";
    }
}
```

Warning When deploying something to a Servlet or JEE Container, Spring Boot is no longer in control of the server; due to this configuration, options from the `server` and `management.servlet` namespaces don't apply anymore. So when a `server.port` has been defined, it will be ignored when deploying to an external server!.

When the application has been deployed, it is no longer available at the root URL / as it would be when using the embedded server. It will be available at the /<name-of-war> URL instead: generally, something like `http://<name-of-server>:8080/<name-of-war>` to access the application. When deployed to a standard Tomcat installation and looking at the Management GUI, it looks like Figure 11-1.

Applications		
Path	Version	Display Name
/	<i>None specified</i>	Welcome to Tomcat
/docs	<i>None specified</i>	Tomcat Documentation
/examples	<i>None specified</i>	Servlet and JSP Examples
/host-manager	<i>None specified</i>	Tomcat Host Manager Application
/manager	<i>None specified</i>	Tomcat Manager Application
/spring-mvc-as-war-2.0.0	<i>None specified</i>	

Figure 11-1. Tomcat management GUI

Clicking the /spring-mvc-as-war-2.0.0 link will open the application (Figure 11-2).



Figure 11-2. Result of deployed application

11-3 Reduce Archive Size Through the Thin Launcher Problem

Spring Boot, by default, generates a so-called fat JAR, a JAR with all the dependencies inside it. This has some obvious benefits, as the JAR is fully self-contained. However, the JAR size can grow considerably, and when using multiple applications you might want to reuse already downloaded dependencies to reduce the overall footprint.

Solution

When packaging the application, you can specify a custom layout. One such a custom layout is the [Thin Launcher].² This launcher will result in the dependencies being downloaded before starting the application, instead of the dependencies packaged inside the application.

How It Works

When using the Spring Boot plugin as is, all the needed libraries are included in the archive in the BOOT-INF/lib folder. However, one might want to reduce the size of the JAR and enable reuse of dependencies to create an even smaller overall footprint. For example, the archive generated with Recipe 14.1 is around 7.1M. Most of this comes from included dependencies.

It is possible to add a custom layout and launcher to the Spring Boot plugin. A layout defines where the sources are loaded from, and the launcher uses this information to load those dependencies. The Thin Launcher will download the artifacts from Maven Central and place them in a shared repository. So when multiple applications share the dependencies, they are only downloaded once.

To use the Thin Launcher, add a dependency to the Spring Boot plugin.

```
<plugin>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-maven-plugin</artifactId>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot.experimental</groupId>
      <artifactId>spring-boot-thin-layout</artifactId>
      <version>1.0.15.RELEASE</version>
    </dependency>
  </dependencies>
</plugin>
```

²<https://github.com/dsyer/spring-boot-thin-launcher>

This is enough to create a considerably smaller archive. When running `./mvn package` the resulting JAR is around 12K. The drawback is, however, that it needs to download the dependencies when the application is started, so starting the application might take longer depending on the number of dependencies.

What happens is that a class named `ThinJarWrapper` has been added and that will be the entrypoint of your application. Now your application contains a `pom.xml` and/or a `META-INF/thin.properties` to determine the dependencies. The `thinJarWrapper` will locate another JAR file (the “launcher”). The wrapper downloads the launcher (if it needs to), or else uses the cached version in your local Maven repository.

The launcher then takes over and reads the `pom.xml` (if present) and the `META-INF/thin.properties`, downloading the dependencies (and all transitives) as necessary. It will now create a custom class loader with all the downloaded dependencies on the classpath. Then it runs the application’s own main method with that class loader.

Note When downloaded, the dependencies from Maven will respect settings made in the `Maven settings.xml`, as it uses regular Maven tooling. When using a mirror like Nexus for Maven repositories, include a `thin.repo` in the `META-INF/thin.properties` to point to that mirror, else downloading the launcher will fail.

11-4 Using Docker

Using Docker³ to build and ship containers is common practice nowadays. When using Spring Boot, it is quite easy to wrap that into a Docker container.

Problem

You want to run your Spring Boot-based application inside a Docker container.

Solution

Create a `Dockerfile` and use one of the available Maven Docker plugins to build the Docker container.

³<https://www.docker.com>

Note In this recipe we choose to use the dockerfile-maven-plugin from Spotify, but there are others that work equally well.

How It Works

First you will need a Dockerfile containing the information needed to build the container. When the container has been built, you can launch it using the `docker run` command.

Update Build Script to Produce a Docker Container

To create a Docker container, first create a Dockerfile. The Dockerfile is the file containing the information on how to build the container. Place this file in the root of the project.

```
FROM openjdk:11-jre-slim
VOLUME /tmp
ARG JAR_FILE
COPY ${JAR_FILE} app.jar
ENTRYPOINT ["java","-Djava.security.egd=file:/dev/.urandom","-jar","/app.jar"]
```

With the preceding Dockerfile, we use the openjdk-provided container to build our own. We are going to add our application to it using the ADD command, and finally we need to tell the container what to start during startup; for this you can use the ENTRYPOINT command.

Caution Using publicly available containers as a starting point might seem like a good idea, but you have to be aware of the possible security implications this can have. As you don't control that specific container, you cannot guarantee what is built into (or not) the container. For real-life situations you might want to build your own base containers.

The ARG with the JAR_FILE argument is telling the build that there is a variable JAR_FILE available that can be used in the build script. We will provide the value of that variable through the plugin configuration.

The ENTRYPPOINT simply specifies that it will run `java -jar /app.jar`. You could also combine this with Recipe 14.1 and install an executable JAR as script, making the entrypoint a little smaller.

Now that there is a Dockerfile, you need to add the `dockerfile-maven-plugin` to your build section of the `pom.xml`.

```
<plugin>
  <groupId>com.spotify</groupId>
  <artifactId>dockerfile-maven-plugin</artifactId>
  <version>1.4.4</version>
  <configuration>
    <repository>spring-boot-recipes/${project.name}</repository>
    <tag>${project.version}</tag>
    <buildArgs>
      <JAR_FILE>target/${project.build.finalName}.jar</JAR_FILE>
    </buildArgs>
  </configuration>
  <dependencies>
    <dependency>
      <groupId>javax.activation</groupId>
      <artifactId>javax.activation-api</artifactId>
      <version>1.2.0</version>
    </dependency>
    <dependency>
      <groupId>org.codehaus.plexus</groupId>
      <artifactId>plexus-archiver</artifactId>
      <version>3.6.0</version>
    </dependency>
  </dependencies>
</plugin>
```

We specify a name of the image through the `repository` property in the plugin; that way we know where to find the container later on. Generally you also want to place a tag on the container, and we choose to use the `${project.version}` as the tag.

The `buildArgs` are for passing arguments to the `docker build` command so that they can be made available when processing the Dockerfile. As we have specified a `JAR_FILE`, we also declare that in the configuration and point it to the location of the generated artifact.

Note The dependency on the `javax.activation-api` is there because that isn't part of the default set of available classes of Java 9; if you are on Java 8 you don't need this. The dependency for `plexus-archiver` is there because the default version with Maven doesn't work on Java 9+.

Building and Starting the Container

Now that everything is in place, you can use `mvn clean build dockerfile:build` to generate a Docker container.

To launch the container, you can run it with something like this:

```
docker run -d spring-boot-recipes/dockerize:2.0.0-SNAPSHOT
```

This will start the container in the background and it will start to print messages to the console. To see the logs, do `docker logs <name-of-container> --follow` and you will see a message printed every two seconds (Figure 11-3).

```
code/ch14/recipe_14_4 master ✘
└> docker logs blissful_hypatia --follow

          _/\_ 
         (   ) 
        / \ \ 
       /   \ 
      /     \ 
     /       \ 
    /         \ 
   /           \ 
  /             \ 
 :: Spring Boot :: (v2.1.0.BUILD-SNAPSHOT)

2018-09-18 18:24:10.754  INFO 1 --- [           main] c.a.s.helloworld.HelloWorldApplication : Starting HelloWorldApplication
2018-09-18 18:24:10.759  INFO 1 --- [           main] c.a.s.helloworld.HelloWorldApplication : No active profile set, falling
2018-09-18 18:24:11.681  INFO 1 --- [           main] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService "main"
2018-09-18 18:24:11.700  INFO 1 --- [           main] o.s.s.c.ThreadPoolTaskScheduler : Initializing ExecutorService "
Hello World, from Spring Boot 2!
2018-09-18 18:24:11.742  INFO 1 --- [           main] c.a.s.helloworld.HelloWorldApplication : Started HelloWorldApplication
Hello World, from Spring Boot 2!
Hello World, from Spring Boot 2!
Hello World, from Spring Boot 2!
```

Figure 11-3. Docker container logging

Passing Properties to the Spring Boot Application

When running a Spring Boot application in a Docker container, you probably want to be able to change properties based on the environment the application gets deployed on. You can use Spring Profiles for this (see also recipes in Chapter 2) but you still need to provide a variable specifying which profile to use. Normally you would start the application like `java -jar your-application.jar --spring.profiles.active=profile1,profile`. However, as you have a Docker container, that isn't possible.

Luckily with Docker you can use the `-e` switch to pass in environment variables and, as Spring Boot also takes into consideration the variables in the local environment,⁴ that will work.

For the simple application we have here, we can pass the `audience` property (by default it will say `World` and the message becomes `Hello World, from Spring Boot 2!`). Let's change it to Docker. To do this you need to add `-e AUDIENCE='Docker'` to the run command.

```
docker run -d -e AUDIENCE='Docker' spring-boot-recipes/dockerize:2.0.0-SNAPSHOT
```

When looking at the logging of the newly started container, the message has changed to `Hello Docker, from Spring Boot 2!` (Figure 11-4).

```
' _   _| .__| |__| | \_, | / / / /  
=====|_|=====|_/_/=/\_/_/_/  
:: Spring Boot :: (v2.1.0.BUILD-SNAPSHOT)  
  
2018-09-18 18:27:00.941 INFO 1 --- [           main] c.a.s.helloworld.HelloWorldApplication : Starting HelloWorldApplication : Starting HelloWorldApplication  
2018-09-18 18:27:00.948 INFO 1 --- [           main] c.a.s.helloworld.HelloWorldApplication : No active profile set, falling back to default profiles: SPRING_PROFILE=, SPRING_CONFIG=null : No active profile set, falling back to default profiles: SPRING_PROFILE=, SPRING_CONFIG=null  
2018-09-18 18:27:01.863 INFO 1 --- [           main] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'taskExecutor' : Initializing ExecutorService 'taskExecutor'  
2018-09-18 18:27:01.882 INFO 1 --- [           main] o.s.s.c.ThreadPoolTaskScheduler : Initializing ExecutorService 'taskScheduler' : Initializing ExecutorService 'taskScheduler'  
2018-09-18 18:27:01.916 INFO 1 --- [           main] c.a.s.helloworld.HelloWorldApplication : Started HelloWorldApplication : Started HelloWorldApplication  
Hello Docker, from Spring Boot 2!  
Hello Docker, from Spring Boot 2!
```

Figure 11-4. Docker container output

Note When setting variables to a Unix based system, you generally want to use all uppercase characters in the name and replace any `.` with a `_`. So when passing in `spring.profiles.active` that would become `SPRING_PROFILES_ACTIVE`.

⁴<https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-external-config.html>

Index

A

Access control decision

ADMIN role, 181

annotations and expressions

@EnableGlobalMethod

Security, 183–184

@PostAuthorize, 183

@PreAuthorize, 183–184

built-in expressions, 180, 181

SpEL, 180

Spring Beans

@ syntax, 182

AccessChecker, 183

@accessChecker.hasLocal

Access, 182

Access data

DataSource (*see* DataSource)

Flyway, database

database migrations, 202

database schema, 202

Flyway properties, 203

flyway_schema_history, 203

Spring Boot, database

ApplicationRunner, 201

DataSource initialize

properties, 199–200

data.sql, 199–200

Derby, 199

initialization, 202

schema.sql, 199–200

using JdbcTemplate (*see* JdbcTemplate)

using JPA (*see* JPA)

using plain Hibernate (*see* Plain
Hibernate)

Actuator

configuration (*see* Configuration,
actuator)

exporting metrics

CPU usage, 299, 304

Graphite, 303–305

micrometer.io, 303

properties, 304

health checks

@EnableScheduling, 302

HealthIndicator, 300–301

MeterBinder interface, 301

MetricBinder, 300

poolSize, 301

TaskScheduler, 300, 301–302

TaskSchedulerMetrics, 300, 301–302

ThreadPoolTaskExecutor, 301

anonymous() method, 172

Applications, Spring Boot

annotations, testing, 159

authenticated result, 158

authentication, 156

BookControllerTest, 159

INDEX

Applications, Spring Boot (*cont.*)

- HTTP headers, 156
- integration test, 160, 162–163
- LibraryApplication, 157
- password generation, 156
- RequestPostProcessor, 159
- ResponseEntity, 162
- setBasicAuth method, 162–163
- spring-boot-starter-security, 155
- TestRestTemplate, 161
- unauthenticated result, 157
- user, 156
- @WebMvcTest, 158–159
- WebTestClient, 161–162
- @With* annotations, 160
- withBasicAuth, 162
- @WithMockUser, 159–160
- @WithUserDetails, 160
- Web applications (*see* Web applications, logging)
- WebFlux application (*see* WebFlux application, security)

Asynchronous request

- async controllers testing, 89–90
- asyncDispatch, 90
- async processing, 90
- Callable<String>, 87–88
- CompletableFuture<String>, 88–89
- controller, 87
- DispatcherServlet, 88
- HTTP request, 86
- return types, 86
- Spring MockMvc, 89
- supplyAsync method, 89
- throughput, 86
- WebMvcConfigurer, 90
- @WebMvcTest, 89

B

@Bean method

- ApplicationRunner interface, 20–21
- @Autowired, 20
- calculator, application, 17
- calculator factory method, 20
- calculator operations, 18–19
- factory method, 17
- operation.apply method, 18
- operation.handles method, 18
- @SpringBootApplication, 19, 20

C

Configuration, actuator

- DataSource, 292
- HTTP metrics, 293–294
- JMX metrics, 292–293
- management server
 - CPU metrics, 299
 - endpoint properties, 296
 - healthcheck, 298
 - HealthIndicators, 298
 - individual endpoints, 296–297
 - micrometer.io, 299
 - port, 294
 - properties, 295
 - public internet, 295
 - secure management
 - endpoints, 297
 - tomcat metrics, 299
 - X-Application-Context
 - header, 295
 - metrics, 298
 - spring-boot-starter-actuator, 291
- @Controller annotated class, 122
- @CookieValue, 122

D

DataSource

- configuration, 194
- DatabaseMetaData, 196
- Derby, 194
- embedded database, 194–195
- HikariCP connection pool
 - settings, 197–198
- JDBC driver, 196
- JDBC support, 194
- JNDI, 197
- leak detection, 199
- PostgreSQL, 196–197
- Spring Boot, 194
- TableLister, 195–197

Docker

- container
 - ADD command, 317
 - buildArgs, 319
 - creation, 317
 - Dockerfile, 317
 - ENTRYPOINT command, 317–318
 - JAR_FILE, 317, 319
 - launch, 319
 - logs, 319
- Maven Docker plugin, 316
- passing properties
 - e switch, 320
 - output, 320
 - Spring Profiles, 320
- DriverManagerDataSource, 193
- DynamicDestinationResolver, 274

E, F

EchoHandlerIntegration
 Test, 108, 110, 151

E-mail, sending

- @Autowired field, 248
- JavaMailSender, 246
- JavaMailSenderImpl, 248
- mail properties, 247
- MailSenderApplication, 249
- MIME message, 246
- spring.mail.host
 - property, 247
- Thymeleaf TemplateEngine
 - HTML templates, 249
 - SpringTemplateEnginge, 251
 - th:text, 250
- @EnableWebFluxSecurity, 185

Error handling

- adding attributes, 63–64
- properties, 59
- working, 59
 - custom error page, 61
 - stacktrace, custom
 - error page, 62–63
 - whitelabel error page, 60

G

- Gradle, 6
 - building, 7
 - creating DemoApplication class, 8–9
- Graphite, 303
- GreenMail, 248

H

- HikariCP, 193–194
- httpie, 94, 133
- HTTP request, 85

- I**
- inMemoryAuthentication() method, 175
 - Integration test, WebSockets
 - assertions, received message, 153
 - callbacks, 152
 - @ClientEndpoint, 152–153
 - closeAndWait, 108, 152
 - @Configuration class, 151
 - connectToServer method, 110
 - ContainerProvider, 153
 - EchoHandlerIntegrationTest, 151
 - @LocalServerPort, 108, 150
 - Netty, 151
 - received messages assertions, 110
 - @RunWith, 150
 - sendTextAndWait, 108, 152
 - SimpleTestClientEndpoint, 153
 - @SpringBootTest, 108, 150
 - WebSocket client, 151
 - Internationalization (I18N) process
 - properties, 66
 - working, 65–68
- J, K**
- Java enterprise services
 - Spring asynchronous process
 - @Async, 239
 - AsyncConfigurer, 243
 - @EnableAsync, 239–240
 - System.in.read, 241
 - TaskExecutor, 239, 241–242
 - TaskExecutorBuilder, 242–243
 - TaskExecutor properties, 242
 - Spring Boot, JMX MBean
 - (*see* JMX MBean)
 - Spring Boot, sending e-mail
 - (*see* E-mail, sending)
- Spring task scheduling
 - @Component, 244
 - @EnableScheduling, 244–245
 - @Scheduled annotation, 243
 - SchedulingApplication, 245
 - SchedulingConfigurer, 245
 - TaskScheduler, 245
 - Java keytool, 80
 - Java Management Extensions (JMX), 239
 - Java WebSocket API, 108
 - JdbcTemplate
 - customer objects, 207
 - customer class creation, 205, 206
 - CustomerLister, 204
 - customer objects, 207–208
 - CustomerRepository, 208
 - single candidate DataSouce, 204
 - findAll method, 209
 - query method, 205
 - repository interface, 206
 - RowCallbackHandler, 205
 - RowMapper, 205, 208
 - testing JDBC code
 - embedded database, 209
 - findAll method, 211
 - Flyway, 210
 - H2, 209
 - includeFilters, 210
 - JdbcCustomerRepository, 209
 - @JdbcTest, 209
 - JUnit runner, 210
 - schema.sql, 210
 - Jetty container, 79
 - JMS
 - ActiveMQ
 - beans, 259
 - cachingJmsConnectionFactory, 259
 - configuration properties, 258

Artemis
 artemis-server, 261
 beans, 260–261
 cachingJmsConnection
 Factory, 260–261
 configuration properties, 260
 embedded configuration
 properties, 262
 JNDI, 262–263
 spring-jms, 260
 ConnectionFactory, 257
 manual configuration, 263
 receiving messages (*see* Receiving messages, JMS)
 sending messages (*see* JmsTemplate)
 spring.jms.template, 264
 JmsListenerContainerFactory, 273
 JmsTemplate
 configuration
 DestinationResolver, 267
 Jackson, 269
 JSON, 268
 MappingJackson2Message
 Converter, 269
 MessageConverter, 267
 ObjectMessage, 269
 Order, 268
 properties, 267
 receiveAndConvert method, 271
 test methods, 271
 test orders, 270–271
 typeIdPropertyName, 270
 @EnableScheduling, 265
 integration test, 265–266
 JUnit test, 266
 sendAndConvert methods, 264
 time-queue, 265

JMX MBean
 JConsole, 252
 JmxApplication, 252–253
 JMX properties, 254
 @ManagedOperation, 255
 @ManagedResource, 252, 255
 printMessage
 operation, 255–256
 shutdown operation, 253–254

JPA
 EntityManagerFactory, 212
 @EntityScan, 217–218
 Order entity, 218
 repositories
 read customers, 214–215
 @Entity, 212
 EntityManager, 213
 @id, 212
 JPA properties, 215–216
 @PersistenceContext, 213–214
 read customers, 215
 spring.jpa.properties, 216
 Spring Data
 CrudRepository, 217
 findAll method, 217
 findById method, 217
 spring.data.jpa.repositories, 217
 testing repositories
 CustomerRepository, 219
 @DataJpaTest, 219
 DataSource, 219
 findAll method, 220
 Flyway, 219
 H2, 219
 TestEntityManager, 219
 @TestPropertySource, 219
 transaction manager, 219

INDEX

L

LibrarySecurityConfig, 168

M

Maven

 DemoApplication class, 3, 5

 pom.xml, 2–3

 @SpringBootApplication
 annotation, 5

MBEans, 239

@MessageMapping annotated
 method, 113, 116

Message/messaging

 receiving

 using JMS, 271

 using RabbitMQ, 286

 sending

 using JMS, 264

 using RabbitMQ, 278

MockMvc-based test, 51

MongoRepository, 232

Multipurpose Internet Mail Extensions
 (MIME message), 246

N, O

NamedParameterJdbcTemplate, 204

Netty, 121, 124, 145

NoOpServerSecurityContext
 Repository, 188

P, Q

Packaging

 archive size reduction

 Docker (*see* Docker)

 Thin Launcher (*see* Thin Launcher)

executable archive

 APP_NAME, 310

 .conf file, 309

 configuration properties, 310

 Gradle plugin, 307

 initInfoDescription, 309

 JAR, 307

 JARFILE, 310

 Java, 309

 Maven plugin, 308

 Spring Boot Maven, 307

WAR file, deployment

 bootstrap, 312

 Recipe 3.1, 311

 result, 314

 Servlet/JEE container, 312–313

 SpringApplicationBuilder, 312

 SpringBootServlet

 Initializer, 311–312

 Tomcat, 311, 313

 Tomcat management GUI, 314

 WEB-INF/lib, 312

Plain Hibernate

 API, 221

 DataSource, 223

 EntityManager, 221–222

 getCurrentSession method, 223

 HibernateCustomerRepository, 222

 JpaTransactionManager, 223

 LocalSessionFactoryBean, 223

 SessionFactory, 221–222

Project reactor, 85

R

Rabbit listener

 @Bean annotated method, 288

 Jackson2JsonMessageConverter, 288

JSON, [287](#)
 MarshallingMessageConverter, [288](#)
 MessageConverter, [288](#)
 properties, [286–287](#)
 @RabbitListener annotated
 methods, [286](#)
 receiving messages, [287](#)
 sending reply
 Order instances, [290](#)
 OrderConfirmation, [289](#)
 @SendTo annotation, [289](#)
 XML, [287](#)

@RabbitListener annotated method, [288](#)

RabbitMQ
 AMQP messaging, [276](#)
 ConnectionFactory, [277](#)
 properties, [277](#)
 receiving messages (*see* Rabbit
 listener)
 sending messages (*see* RabbitTemplate)

RabbitTemplate
 configuration properties, [278](#)
 ConnectionFactory, [278](#)
 convertAndSend method, [279](#)
 HelloWorldSender method, [280](#)
 integration test, [283, 285](#)
 @MockBean, [280](#)
 retry configuration, [279](#)
 sending object
 convertAndSend method, [283](#)
 Jackson2JsonMessage
 Converter, [281](#)
 Java serialization, [281](#)
 MessageConverter, [281](#)
 @MockBean, [283](#)
 scheduled method, [282](#)
 SimpleMessageConverter, [281](#)
 send messages, [278](#)

String.getBytes, [281](#)
 @Test method, [280](#)
 ReactiveAuthenticationManager, [188](#)
 Reactive Rest services
 @AutoConfigureWebTestClient, [137](#)
 @DirtiesContext, [137](#)
 integration test, [135–137](#)
 JSON streaming, [134–135](#)
 Order class, [129–131](#)
 OrderController, [131–133](#)
 OrderGenerator, [129, 131](#)
 OrderService, [128–129](#)
 orders result, [133](#)
 @RestController, [128](#)
 Server Sent Events, [135](#)
 @SpringBootTest, [137](#)
 WebTestClient, [135](#)

Receiving messages, JMS
 @JmsListener, [271](#)
 listener container, [273–274](#)

MappingJackson2Message
 Converter, [274](#)
 MessageConverter, [274–275](#)
 method parameter types, [272](#)
 OrderConfirmation, [275–276](#)
 POJO, [271](#)
 @SendTo, [275](#)
 String, [273](#)

rememberMe() method, [172](#)

@RequestAttribute, [122](#)
 Request handling method, [46, 56](#)
 @RequestHeader, [122](#)
 @RequestParam, [122](#)
 @ResponseBody annotation, [40](#)

Response writers
 events, multiple results
 event() factory-method, [98](#)
 HttpMessageConverter, [98](#)

INDEX

Response writers (*cont.*)

 Server-Sent-Events, 96–97

 SseEmitter, 97–99

 text/event-stream, 98

ResponseBodyEmitter, 91

response, multiple results

 complete() method, 94

 httpie, 94

 HttpMessageConverter, 91

 @MockBean, 95, 96

 MockMvc, 96

 Order, 91

 OrderController, 91, 93, 94

 OrderService, 92, 93

 OrderService.findAll

 method, 94

 ResponseBodyEmitter, 94–95

 @RunWith class, 95

 @WebMvcTest class, 95

@RestController annotation, 40

RowCallbackHandler, 205

RSA algorithm, 80

S

SecurityMockMvcRequestPost

 Processors, 159

@SendTo annotation, 289

ServerHttpRequest/ServerHttp

 Response, 122

ServerSecurityContext

 Repository, 188

Server-Sent-Events, 96, 100

SimpleDataSource, 193

Simple/Streaming Text Oriented Message

 Protocol (STOMP)

 defined, 111

 handler unit test, 116

integration testing

 CompletableFuture, 119

 EchoHandlerIntegrationTest
 class, 117

 transfer result, 118

 @RunWith, 117

 STOMP client, 118

 StompSession, 119

 test case, 118–119

 WebSocketStompClient, 117

MessageMapping

 channels and routing, 113

 EchoHandler, 111–112

 @EnableWebSocketMessage
 Broker, 112–113

 method arguments and

 annotations, 116

 STOMP messages, 113

 webstomp-client, 114–115

WebSocket protocol, 111

Spring Boot

bean class

 @Component, 16

 @ComponentScan, 15

 HelloWorldApplication, 16

 @PostConstruct annotated
 method, 16

@Bean method (*see* @Bean method)

configuration

 @Configuration classes, 35

 @Import, 35

 @ImportResource, 34

 Java, 35

 @SpringBootApplication, 34

 XML, 35

logging configuration

 DEBUG logging, 33

 Java temp directory, 33

Logback, provider, 33–34
 logfile, 33
 logging.file, 33
 logging.file.max-history, 33
 logging.file.max-size, 33
 logging.path, 33
 log providers, 32
 SLF4J, 32
 packaging (*see* Packaging)
 properties
 application.properties, 24
 CalculatorApplication, 23
 command line arguments, 22, 25
 command line parameters, 26
 IllegalArgumentException, 24
 override, 22
 profiles, 25
 profile-specific, 22, 26
 @PropertySource, 25
 resources, 22
 @Value, 23–24
 testing
 bootstrapping, 27
 calculator, 28–29
 JUnit rule, 30
 @MockBean, 30–31
 Mockito framework, 27, 28, 32
 Mockito.mock, 29
 ReflectionTestUtils class, 31
 @Rule, 30
 @SpringBootApplication, 29
 @SpringBootTest, 29
 @Test, 28
 unit test, 27–28
 Spring Boot actuator, *see* Actuator
 Spring-boot-starter-security, 155
 Spring-boot-starter-test, 12
 Spring-boot-starter-websocket, 100
 Spring Data MongoDB
 embedded MongoDB, 230
 external connection
 port 27017, 231
 localhost, 231
 MongoClient, 232
 MongoDbFactory, 232
 MongoDB properties, 231
 MongoTemplate, 224
 ApplicationListeners, 228, 229
 ApplicationRunner, 227
 create customer, 224–225
 CustomerRepository, 225
 DataInitializer, 227
 embedded MongoDB, 229
 implementation, 226
 reactive repositories
 CustomerListener, 234
 DataInitializer, 233
 elements delay, 235
 reactive driver, 232
 reactive libraries, 232
 ReactiveMongoRepository, 233
 System.in.read(), 234
 testing
 CustomerRepository, 237
 @DataMongoTest, 236
 deleteAll method, 236
 embedded MongoDB, 236
 findAll method, 237
 JUnit, 236
 Spring Expression Language (SpEL), 180
 Spring Initializr, 10
 ApplicationContext, 13
 building JAR, 13
 import project, 12
 spring-boot-maven-plugin, 12
 with values, 11

INDEX

- spring.jpa.properties, 216
- Spring Messaging abstraction, 111
- Spring MockMvc testing
 - framework, 27
- Spring MVC
 - configure SSL
 - key-store, 79–80
 - namespace, 79
 - support HTTPS, 81–84
 - embedded server
 - properties, 73–77
 - runtime container, 78
 - filters, 39
 - REST Resources
 - testing @RestController, 49–51
 - working, 42–46, 48–49
 - server properties, 73–74
 - testing, 40, 42
 - working
 - HelloWorldApplication, 38–39
 - HelloWorldController, 40
 - locale resolver, 68–72
 - spring-boot-starter-web, 37
- Spring security
 - authenticating users (*see* User authentication)
 - control decisions access (*see* Access control decision)
- Spring Boot, 155
- Spring Boot application (*see* Applications, Spring Boot)
- SseEventBuilder, 98
- StompSessionHandlerAdapter, 117
- Thin Launcher
 - dependencies reuse, 315
 - drawback, 316
 - layout, 315
 - libraries, 315
 - Maven Central, 315
 - META-INF/thin.properties, 316
 - pom.xml, 316
 - thinJarWrapper, 316
- ThreadPoolTaskExecutor, 241
- Thymeleaf, 52
 - controller, 140
 - Flux, 143
 - Flux<Order>, 140–141
 - index.html, 139
 - list.html, 141–142
 - OrderApplication, 142
 - OrderController, 140
 - order list, 143
 - OrderService, 141
 - properties, 53, 138–139
 - ReactiveDataDriverContext Variable, 143–144
- SpringWebFluxTemplateEngine, 138
- ThymeleafReactiveViewResolver, 138
- WebFlux application, 137
- working, 52
 - adding controller, 55–57
 - adding details page, 57, 59
 - adding index page, 53
- Thymeleaf Spring Dialect, 138
- Tomcat server, 73, 151

T

- TaskSchedulerHealthIndicator, 301
- TestStompFrameHandler, 119

U, V

- User authentication
 - AuthenticationProviders, 173
 - BCrypt, 174

- database
 - authoritiesByUsernameQuery(), [178](#)
 - authorities table, [176](#)
 - jdbcAuthentication()
 - configuration, [177](#)
 - MEMBER_ROLE table, [178](#)
 - MEMBER table, [177–178](#)
 - spring.datasource properties, [177](#)
 - SQL statements, [175](#)
 - usersByUsernameQuery(), [178](#)
 - users table, [176](#)
 - encrypted passwords, [173](#)
 - hash function, [173](#)
 - in-memory, [174–175](#)
 - password encryption, [179](#)
 - SCrypt, [174](#)
 - User.withDefaultPasswordEncoder()
 - method, [175](#)
-
- ## **W, X, Y, Z**
- Web applications, logging
 - anonymous login, [172](#)
 - anonymous user, [163](#)
 - exception handling, [164](#)
 - form-based login
 - addViewControllers methods, [168](#)
 - authentication-failure-url, [169](#)
 - configuration, [167](#)
 - CSRF protection, [166](#)
 - error message, [169](#)
 - error request, [169](#)
 - formLogin method, [165](#)
 - login.html file, [166, 167](#)
 - secure URL, [168](#)
 - SPRING_SECURITY_LAST_EXCEPTION, [169–170](#)
 - view resolver, [168](#)
 - HTTP auto-config, [164](#)
 - HTTP Basic authentication, [163, 165](#)
 - httpBasic() method, [165](#)
 - HttpServletRequest, [165](#)
 - logout service
 - caches, [171–172](#)
 - logout() configuration, [170](#)
 - logoutSuccessUrl configuration, [171](#)
 - POST requests, [170](#)
 - remember-me login, [164](#)
 - remember-me support, [172](#)
 - security configuration, [164](#)
 - WebSecurityConfigurerAdapter, [164](#)
 - WebFlux
 - application, set up, [124](#)
 - @Controller/@RestController, [122](#)
 - controller, unit test
 - MockMvc, [126](#)
 - StepVerifier, [126](#)
 - @WebFluxTest, [125](#)
 - WebTestClient, [127](#)
 - creating controller, [125](#)
 - @GetMapping, [125](#)
 - handler method, [122–123](#)
 - HelloWorldApplication, [123–124](#)
 - HTTP GET handler method, [125](#)
 - HttpHandler, [121](#)
 - Mono<Void>, [121](#)
 - integration test
 - @SpringBootTest, [127](#)
 - WebTestClient, [127–128](#)
 - Map and Model objects, [123](#)
 - ReactorHttpHandler
 - Adapter, [121](#)
 - @RequestMapping, [122](#)
 - ServletHttpHandler
 - Adapter, [121](#)
 - ViewResolver interface, [123](#)

INDEX

WebFlux application, security
access control decisions
 access() expression, 191
 built-in expressions, 190
 ROLE_ADMIN, 191
@EnableWebFluxSecurity, 185
HTTP headers, 185
JAR files, 185
logging, 188
login form, 189
login page, 186
ReactiveAuthenticationManager, 188
spring-boot-starter-security, 185
URL access
 csrf(), 187
 SecurityWebFilterChain, 187
 ServerHttpSecurity, 187
 user authentication, 188–189
 UserDetailsRepository, 188–189
WebMvcConfigurer, 168
@WebMvcTest, 41
WebSocket JavaScript object, 105
WebSocketConfigurer, 102
WebSocketHandlerRegistry, 103
WebSocketMessageBroker
 Configurer, 112–113
WebSockets
 bidirectional communication, 100
 configuration, 100
 EchoHandler, 145
 @EnableWebSocket, 100
 full duplex communication, 100
 HTML & JavaScript
 app.js, 147–148
 callbacks, 148
 echo function, 149
 index.html, 149–150
 WebSocket connection, 148
 WebSocket service, 150
HTTP, 100
integration test (*see* Integration test, WebSockets)
javax.websocket-api, 144
reactive application, 144
SimpleUrlHandlerMapping, 145
STOMP (*see* Simple/Streaming Text Oriented Message Protocol (STOMP))
WebSocketHandler (*see* WebSocketHandler)
WebSocketHandler
 Adapter, 145
 WebSocketService, 145–147
WebSocketHandler, 145
 app.js, 103–104
 client output, 106
 @Configuration class, 102
 echo function, 105
 EchoHandler, 101
 echo WebSocket service, 106
 index.html, 105–106
 methods, 101
 Mockito, 107
 onclose callbacks, 105
 onmessage callbacks, 105
 onopen callbacks, 105
 registerWebSocketHandlers
 method, 102–103
 unit test, 107
WebSocketStompClient, 119