



Pro Spring Boot 2

An Authoritative Guide to Building
Microservices, Web and Enterprise
Applications, and Best Practices

—
Second Edition

—
Felipe Gutierrez

Apress®

www.allitebooks.com

Pro Spring Boot 2

An Authoritative Guide to Building
Microservices, Web and Enterprise
Applications, and Best Practices

Second Edition

Felipe Gutierrez

Apress®

Pro Spring Boot 2: An Authoritative Guide to Building Microservices, Web and Enterprise Applications, and Best Practices

Felipe Gutierrez
Albuquerque, NM, USA

ISBN-13 (pbk): 978-1-4842-3675-8

ISBN-13 (electronic): 978-1-4842-3676-5

<https://doi.org/10.1007/978-1-4842-3676-5>

Library of Congress Control Number: 2016941344

Copyright © 2019 by Felipe Gutierrez

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 Spahr
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 <http://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/9781484236758. For more detailed information, please visit <http://www.apress.com/source-code>.

Printed on acid-free paper

To my wife, Norma Castaneda

Table of Contents

About the Author	xiii
About the Technical Reviewer	xv
Acknowledgments	xvii
Chapter 1: Spring Framework 5	1
A Little History.....	1
Design Principles and Patterns	2
Spring Framework 5	3
A Simple Spring Web Application	4
Using Maven for Creating a Project.....	4
Adding Dependencies.....	5
Spring Web Configuration.....	9
Classes	17
Running the App	23
Using Java Config.....	26
Summary.....	30
Chapter 2: Introduction to Spring Boot.....	31
Spring Boot	31
Spring Boot to the Rescue.....	33
Spring Boot CLI.....	34
Spring Boot Application Model.....	36
My First Spring Boot Application	37
Why Spring Boot?.....	42
Spring Boot Features.....	43
Summary.....	44

TABLE OF CONTENTS

- Chapter 3: Spring Boot Internals and Features 45**
 - Auto-Configuration 45
 - Disable a Specific Auto-Configuration 48
 - @EnableAutoConfiguration and @Enable<Technology> annotations 50
 - Spring Boot Features 54
 - SpringApplication Class 57
 - Custom Banner 58
 - SpringApplicationBuilder 63
 - Application Arguments 66
 - ApplicationRunner and CommandLineRunner 68
 - Application Configuration 71
 - Configuration Properties Examples 73
 - Custom Properties Prefix 81
 - Summary 85
- Chapter 4: Web Applications with Spring Boot 87**
 - Spring MVC 87
 - Spring Boot MVC Auto-Configuration 88
 - Spring Boot Web: ToDo App 90
 - ToDo App 91
 - Running: ToDo App 104
 - Testing: ToDo App 105
 - Spring Boot Web: Overriding Defaults 110
 - Server Overriding 110
 - JSON Date Format 112
 - Content-Type: JSON/XML 112
 - Spring MVC: Overriding Defaults 114
 - Using a Different Application Container 114
 - Spring Boot Web: Client 115
 - ToDo Client App 116
 - Summary 124

Chapter 5: Data Access with Spring Boot	127
SQL Databases	127
Spring Data	128
Spring JDBC	129
JDBC with Spring Boot	130
ToDo App with JDBC	131
Spring Data JPA	139
Spring Data JPA with Spring Boot	140
ToDo App with Spring Data JPA	141
Spring Data REST	150
Spring Data REST with Spring Boot	150
ToDo App with Spring Data JPA and Spring Data REST	151
No SQL Databases	158
Spring Data MongoDB	158
Spring Data MongoDB with Spring Boot	159
ToDo App with Spring Data MongoDB	161
ToDo App with Spring Data MongoDB REST	165
Spring Data Redis	165
Spring Data Redis with Spring Boot	166
ToDo App with Spring Data Redis	166
More Data Features with Spring Boot	170
Multiple Data Sources	170
Summary	171
Chapter 6: WebFlux and Reactive Data with Spring Boot	173
Reactive Systems	173
The Reactive Manifesto	173
Project Reactor	175
ToDo App with Reactor	175

TABLE OF CONTENTS

- WebFlux 184
 - WebClient 186
- WebFlux and Spring Boot Auto-configuration 187
 - Using WebFlux with Spring Boot 188
- Reactive Data 196
 - MongoDB Reactive Streams 196
- Summary..... 206
- Chapter 7: Testing with Spring Boot..... 207**
 - Spring Testing Framework 207
 - Spring Boot Testing Framework..... 209
 - Testing Web Endpoints 210
 - Mocking Beans 211
 - Spring Boot Testing Slices 212
 - Summary..... 218
- Chapter 8: Security with Spring Boot 219**
 - Spring Security 219
 - Security with Spring Boot 220
 - ToDo App with Basic Security 220
 - Overriding Simple Security 227
 - Overriding the Default Login Page..... 229
 - Custom Login Page..... 232
 - Using Security with JDBC 240
 - Directory App with JDBC Security 241
 - Using the Directory App within the ToDo App 250
 - WebFlux Security 257
 - ToDo App with OAuth2..... 257
 - Creating the ToDo App in GitHub..... 261
 - Summary..... 268

Chapter 9: Messaging with Spring Boot	269
What Is Messaging?	269
JMS with Spring Boot	270
ToDo App with JMS	270
Using JMS Pub/Sub	281
Remote ActiveMQ	282
RabbitMQ with Spring Boot	282
Installing RabbitMQ	283
RabbitMQ/AMQP: Exchanges, Bindings, and Queues	283
ToDo App with RabbitMQ	285
Remote RabbitMQ	297
Redis Messaging with Spring Boot	298
Installing Redis	298
ToDo App with Redis	298
Remote Redis	306
WebSockets with Spring Boot	307
ToDo App with WebSockets	307
Summary	318
Chapter 10: Spring Boot Actuator	319
Spring Boot Actuator	319
ToDo App with Actuator	320
/actuator	325
/actuator/conditions	326
/actuator/beans	327
/actuator/configprops	328
/actuator/threaddump	329
/actuator/env	330
/actuator/health	331
/actuator/info	333
/actuator/loggers	333

TABLE OF CONTENTS

- [/actuator/loggers/{name}](#) 334
- [/actuator/metrics](#)..... 334
- [/actuator/mappings](#)..... 336
- [/actuator/shutdown](#) 337
- [/actuator/httptrace](#) 339
- [Changing the Endpoint ID](#) 340
- [Actuator CORS Support](#) 341
- [Changing the Management Endpoints Path](#) 341
- [Securing Endpoints](#) 342
- [Configuring Endpoints](#) 342
- [Implementing Custom Actuator Endpoints](#)..... 343
 - [ToDo App with Custom Actuator Endpoints](#) 343
- [Spring Boot Actuator Health](#)..... 353
 - [ToDo App with Custom HealthIndicator](#) 358
- [Spring Boot Actuator Metrics](#)..... 363
 - [ToDo App with Micrometer: Prometheus and Grafana](#)..... 363
 - [General Stats for Spring Boot with Grafana](#)..... 378
- [Summary](#)..... 381
- [Chapter 11: Spring Integration and Spring Cloud Stream with Spring Boot](#) 383**
 - [Spring Integration Primer](#)..... 384
 - [Programming Spring Integration](#) 386
 - [Using XML](#)..... 392
 - [Using Annotations](#)..... 395
 - [Using JavaConfig](#)..... 397
 - [ToDo with File Integration](#)..... 398
 - [Spring Cloud Stream](#) 405
 - [Spring Cloud](#) 405
 - [Spring Cloud Stream](#) 407
 - [Spring Cloud Stream App Starters](#)..... 430
 - [Summary](#)..... 431

Chapter 12: Spring Boot in the Cloud	433
The Cloud and Cloud-Native Architecture	433
Twelve-Factor Applications	434
Microservices.....	436
Preparing the ToDo App as a Microservice	437
Pivotal Cloud Foundry	438
PAS: Pivotal Application Service	439
PAS Features	440
Using PWS/PAS.....	441
Cloud Foundry CLI: Command-Line Interface	444
Log in to PWS/PAS Using the CLI Tool.....	444
Deploying the ToDo App into PAS.....	445
Creating Services	448
Cleaning Up	453
Summary.....	454
Chapter 13: Extending Spring Boot.....	455
Creating a spring-boot-starter	455
todo-client-spring-boot-starter	457
todo-client-spring-boot-autoconfigure.....	459
Creating an @Enable* Feature.....	467
ToDo REST API Service.....	471
Installing and Testing	474
Task Project.....	474
Running the Task App	477
Summary.....	479
Appendix A: Spring Boot CLI.....	481
Spring Boot CLI	481
The run Command.....	483
The test Command.....	485

TABLE OF CONTENTS

The grab Command..... 488

The jar Command..... 489

The war Command..... 491

The install Command 492

The uninstall Command 493

The init Command..... 494

The shell Command 498

The help Command 499

Summary..... 500

Index..... 501

About the Author



Felipe Gutierrez is a solutions software architect, with bachelor's and master's degrees in computer science from Instituto Tecnológico y de Estudios Superiores de Monterrey Campus Ciudad de México. Gutierrez has over 20 years of IT experience and has developed programs for companies in multiple vertical industries, such as government, retail, healthcare, education, and banking. He is currently working as a platform and solutions architect for Pivotal, specializing in cloud foundry PAS and PKS, Spring Framework, Spring Cloud Native Applications, Groovy, and RabbitMQ, among other technologies. He has also worked as a solutions architect for big companies like Nokia, Apple, Redbox, and Qualcomm. Gutierrez is the author of *Spring Boot Messaging* (Apress, 2017) and *Introducing Spring Framework* (Apress, 2014).

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, including *Pro Spring, 4th Edition* (Apress, 2014), *Practical Spring LDAP* (Apress, 2013), *Pro JPA 2, Second Edition* (Apress, 2013), and *Pro Spring Security* (Apress, 2013). You can read his detailed tutorials about Spring technologies and contact him through his blog at www.manueljordanelera.blogspot.com, and follow him on his Twitter account, `@dr_pompeii`.

Acknowledgments

I would like to express all my gratitude to the Apress team: Steve Anglin for accepting my proposal, Mark Powers for keeping me on track and for his patience with me, and the rest of the Apress team involved in this project. Thanks to everybody for making this possible.

Thanks to my technical reviewer, Manuel Jordan, for all the details and effort in his reviews, and the entire Spring Boot team for creating this amazing technology.

Thanks to my parents, Rocio Cruz and Felipe Gutierrez, for all their love and support; to my brother, Edgar Gerardo Gutierrez, and my sister-in-law, Auristella Sanchez, and specially to my girls, who also keep me on track—Norma, Laura “Lau”, Nayely my “Flaca”, and Ximena my “Gallito”—I love you girls. And to my baby, Rodrigo!

—Felipe Gutierrez

CHAPTER 1

Spring Framework 5

Welcome to the first chapter of the book, where I give you an introduction to the Spring Framework, a little bit of history, and how it has evolved since its inception. This chapter is for developers that are new to the Spring Framework. If you are an experienced Spring Framework developer, you can skip this chapter.

Maybe you are thinking, “I want to learn Spring Boot. Why do I need to know about Spring Framework?” Well, let me tell you that Spring Boot *is* Spring. Spring Boot has a different mechanism for running Spring applications; to understand how Spring Boot really works and does its job, it is necessary to know more about the Spring Framework.

A Little History

The Spring Framework was created in 2003 by Rod Johnson, author of *J2EE Development without EJB* (Wrox Publishing, 2004). The Spring Framework was the response to all the complexity that the J2EE specifications had at that time. Today, it has improved, but you need to have a whole infrastructure to run certain aspects of the J2EE ecosystem.

We can say Spring is a complementary technology to Java EE. The Spring Framework integrates several technologies, such as Servlet API, WebSocket API, concurrency utilities, JSON Binding API, bean validation, JPA, JMS, and JTA/JCA.

The Spring Framework supports the *dependency injection* and *common annotation* specifications that make development easier.

This chapter shows that the Spring Framework version 5.x requires a Java EE 7 level (Servlet 3.1+ and JPA 2.1) as the minimum. Spring still works with Tomcat 8 and 9, WebSphere 8, and JBoss EAP 7. Also, I show you the new addition to the Spring Framework 5— reactive support!

Nowadays, Spring is one of the most used and recognized frameworks in the Java community, not only because it works, but because it continues to innovate with other amazing projects, including Spring Boot, Spring Security, Spring Data, Spring Cloud, Spring Batch, and Spring Integration, among others.

Design Principles and Patterns

To know Spring Boot, you need to learn about a framework; it's important to know not only what it does but also which principles it follows. The following are some of the principles of the Spring Framework.

- *Provide choice at every level.* Spring lets you defer design decisions as late as possible. For example, you can switch persistence providers through configuration without changing your code. The same is true for many other infrastructure concerns and integration with third-party APIs. And you will see, this even happens when you deploy your application to the cloud.
- *Accommodate diverse perspectives.* Spring embraces flexibility and is not opinionated about how things should be done. It supports a wide range of application needs with different perspectives.
- *Maintain strong backward compatibility.* Spring's evolution has been carefully managed to force few breaking changes between versions. Spring supports a carefully chosen range of JDK versions and third-party libraries to facilitate maintenance of applications and libraries that depend on Spring.
- *Care about API design.* The Spring team puts a lot of thought and time into making APIs that are intuitive and that hold up across many versions and many years.
- *Set high standards for code quality.* The Spring Framework puts a strong emphasis on meaningful, current, and accurate Javadocs. It is one of very few projects that can claim clean code structure with no circular dependencies between packages.

So, what do you need to run a Spring application? Spring works with Plain Old Java Objects (POJOs), making it easy to extend. Spring is not invasive and makes your application enterprise ready; but you need to help Spring by adding a configuration to wire up all dependencies and inject what's needed to create Spring *beans* to execute your application (see Figure 1-1).

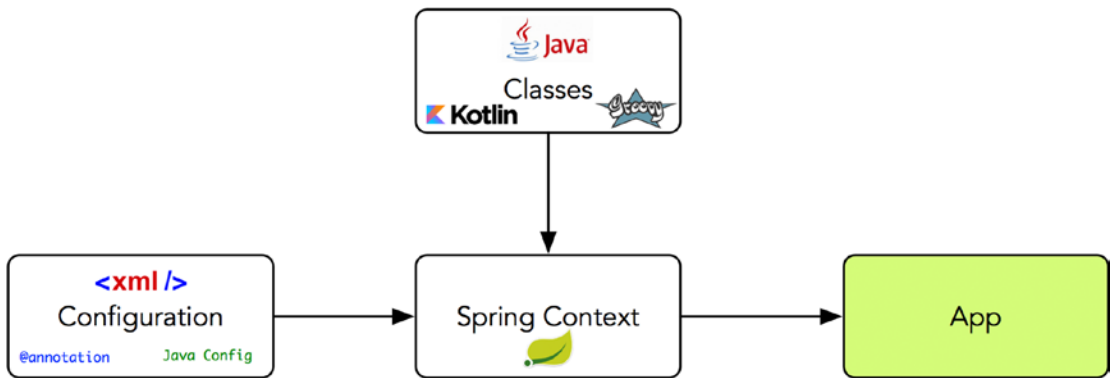


Figure 1-1. Spring context

Figure 1-1 shows the Spring context that creates all the Spring beans—thanks to the configuration that references your classes, which makes your application run. You find out more in the next sections, in which you create a complete REST API app.

Spring Framework 5

Spring makes it easy to create Java enterprise applications because it provides everything that a developer needs to embrace the Java language in an enterprise environment. It offers excellent support of Groovy and Kotlin as alternative languages on the JVM (Java virtual machine).

Spring Framework 5 requires JDK 8+ and provides out-of-the-box-support for Java Development Kit (JDK) 9, 10, and 11. The Spring team has the same long-term maintenance support for 11 and version 17, which correlates with the JDK team. This new version came out in 2017 with a new way to do functional programming with Reactive Streams.

Spring Web MVC was built to serve the Servlet API and Servlet containers. This was OK until there was more demand for services, which detected a particular problem: there was some blocking on each request; and with high demand, it was necessary to do something else. The result: the reactive stack, a web framework. The *Spring WebFlux* module was introduced in version 5, with a fully non-blocking stack that supports Reactive Streams back pressure and runs on servers such as Netty, Undertow, and Servlet 3.1+ containers. This was part of the answer for a non-blocking stack that handles concurrency with a small number of threads that can scale with less hardware.

The WebFlux module depends on another Spring project: *Project Reactor*. Reactor is the reactive library of choice for Spring WebFlux. It provides the Mono and Flux API types to work on data sequences of 0..1 and 0..N through a rich set of operators aligned with the *ReactiveX* vocabulary of operators. Reactor is a Reactive Streams library, and therefore, all of its operators support non-blocking back pressure. Reactor has a strong focus on server-side Java. It is developed in close collaboration with Spring.

I don't want to get into much of the Spring features because I can show them with a simple web application. What do you think? All of these cool WebFlux features are reviewed in its own chapter.

A Simple Spring Web Application

Let's start by creating a Spring web application— a ToDo app that offers a REST API that can do a CRUD (create, read, update, and delete). To create a new Spring app, you need to have Maven installed. In the following chapters, you can choose either Maven or Gradle.

Using Maven for Creating a Project

Let's start by using the following command from Maven to create the ToDo Spring project.

```
$ mvn archetype:generate -DgroupId=com.apress.todo  
-DartifactId=todo -Dversion=0.0.1-SNAPSHOT -DinteractiveMode=false  
-DarchetypeArtifactId=maven-archetype-webapp
```

This command is generating the basic template and structure for web applications. Normally, it generates the webapp and resources folders but not the java folder, which you need to create manually.

```

todo
├── pom.xml
└── src
    ├── main
    │   ├── resources
    │   └── webapp
    │       ├── WEB-INF
    │       │   └── web.xml
    │       └── index.jsp

```

You can import the code in your favorite IDE; this will make it easier to identify any problems.

Adding Dependencies

Open `pom.xml` and replace all the content with Listing 1-1.

Listing 1-1. `todo/pom.xml`

```

<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/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.apress.todo</groupId>
  <artifactId>todo</artifactId>
  <packaging>war</packaging>
  <version>0.0.1-SNAPSHOT</version>
  <name>todo Webapp</name>

  <properties>
    <!-- Generic properties -->
    <java.version>1.8</java.version>

```

```

<!-- Web -->
<jsp.version>2.2</jsp.version>
<jstl.version>1.2</jstl.version>
<servlet.version>3.1.0</servlet.version>
<bootstrap.version>3.3.7</bootstrap.version>
<jackson.version>2.9.2</jackson.version>
<webjars.version>0.32</webjars.version>

<!-- Spring -->
<spring-framework.version>5.0.3.RELEASE</spring-framework.version>

<!-- JPA -->
<spring-data-jpa>1.11.4.RELEASE</spring-data-jpa>
<hibernate-jpa.version>1.0.0.Final</hibernate-jpa.version>
<hibernate.version>4.3.11.Final</hibernate.version>

<!-- Drivers -->
<h2.version>1.4.197</h2.version>

<!-- Logs -->
<slf4j.version>1.7.25</slf4j.version>
<logback.version>1.2.3</logback.version>
</properties>

<dependencies>
  <!-- Spring MVC -->
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>${spring-framework.version}</version>
  </dependency>

  <!-- Spring Data JPA -->
  <dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-jpa</artifactId>
    <version>${spring-data-jpa}</version>
  </dependency>

```

```

<dependency>
  <groupId>org.hibernate.javax.persistence</groupId>
  <artifactId>hibernate-jpa-2.1-api</artifactId>
  <version>${hibernate-jpa.version}</version>
</dependency>
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>${hibernate.version}</version>
</dependency>
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-entitymanager</artifactId>
  <version>${hibernate.version}</version>
</dependency>

<!-- Logs -->
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>jcl-over-slf4j</artifactId>
  <version>${slf4j.version}</version>
</dependency>
<dependency>
  <groupId>ch.qos.logback</groupId>
  <artifactId>logback-classic</artifactId>
  <version>${logback.version}</version>
</dependency>

<!-- Drivers -->
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
  <version>${h2.version}</version>
  <scope>runtime</scope>
</dependency>

```

```

<!-- Java EE Web dependencies -->
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>jstl</artifactId>
  <version>${jstl.version}</version>
</dependency>
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>javax.servlet-api</artifactId>
  <version>${servlet.version}</version>
  <scope>provided</scope>
</dependency>
<dependency>
  <groupId>javax.servlet.jsp</groupId>
  <artifactId>jsp-api</artifactId>
  <version>${jsp.version}</version>
  <scope>provided</scope>
</dependency>

<!-- Web UI -->
<dependency>
  <groupId>org.webjars</groupId>
  <artifactId>webjars-locator</artifactId>
  <version>${webjars.version}</version>
</dependency>

<dependency>
  <groupId>org.webjars</groupId>
  <artifactId>bootstrap</artifactId>
  <version>${bootstrap.version}</version>
</dependency>

<!-- Web - JSON/XML Response -->
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
  <version>${jackson.version}</version>
</dependency>

```

```

<dependency>
  <groupId>com.fasterxml.jackson.datatype</groupId>
  <artifactId>jackson-datatype-joda</artifactId>
  <version>${jackson.version}</version>
</dependency>

<dependency>
  <groupId>com.fasterxml.jackson.dataformat</groupId>
  <artifactId>jackson-dataformat-xml</artifactId>
  <version>${jackson.version}</version>
</dependency>
</dependencies>

<build>
  <finalName>todo</finalName>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <configuration>
        <source>1.8</source>
        <target>1.8</target>
      </configuration>
    </plugin>
  </plugins>
</build>
</project>

```

Listing 1-1 shows the `pom.xml` file and all the dependencies that you need to create a simple Spring web app.

Spring Web Configuration

Next, let's start with the Spring configuration. Spring needs the developer to decide where the classes are and how they interact with each other, as well some extra configuration for web applications.

Let's start by modifying the `web.xml` file, shown in Listing 1-2.

Listing 1-2. todo/src/main/webapp/WEB-INF/web.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    version="2.5">
  <display-name>ToDo Web Application</display-name>
  <servlet>
    <servlet-name>dispatcherServlet</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>dispatcherServlet</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>
</web-app>

```

It is necessary to set up the `DispatcherServlet`, which is the main entry point for any Spring web app. This class wires up everything based on the context configuration. As you can see, it is a very trivial configuration.

Next, let's configure the Spring context by creating a `dispatcherServlet-servlet.xml` file. There is a naming convention; if the servlet is named `todo` in the `web.xml` file, then the Spring context file should be named `todo-servlet.xml`. In this case, the servlet was named `dispatcherServlet`, so it looks for a `dispatcherServlet-servlet.xml` file (see Listing 1-3).

Listing 1-3. todo/src/main/webapp/WEB-INF/dispatcherServlet-servlet.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"

```

```

xmlns:mvc="http://www.springframework.org/schema/mvc"
xmlns:jpa="http://www.springframework.org/schema/data/jpa"
xmlns:jdbc="http://www.springframework.org/schema/jdbc"
xmlns:tx="http://www.springframework.org/schema/tx"
xsi:schemaLocation="http://www.springframework.org/schema/jdbc
http://www.springframework.org/schema/jdbc/spring-jdbc-4.3.xsd
http://www.springframework.org/schema/mvc http://www.
springframework.org/schema/mvc/spring-mvc-4.3.xsd
http://www.springframework.org/schema/beans http://www.
springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context http://www.
springframework.org/schema/context/spring-context-4.3.xsd
http://www.springframework.org/schema/data/jpa http://www.
springframework.org/schema/data/jpa/spring-jpa-1.8.xsd
http://www.springframework.org/schema/tx http://www.
springframework.org/schema/tx/spring-tx-4.3.xsd">

```

```
<context:component-scan base-package="com.apress.todo" />
```

```
<mvc:annotation-driven>
```

```
  <mvc:message-converters>
```

```
    <bean class="org.springframework.http.converter.json.
      MappingJackson2HttpMessageConverter">
```

```
      <property name="objectMapper" ref="jsonMapper"/>
```

```
    </bean>
```

```
    <bean class="org.springframework.http.converter.xml.
      MappingJackson2XmlHttpMessageConverter">
```

```
      <property name="objectMapper" ref="xmlMapper"/>
```

```
    </bean>
```

```
  </mvc:message-converters>
```

```
</mvc:annotation-driven>
```

```
<bean id="jsonMapper" class="org.springframework.http.converter.json.
  Jackson2ObjectMapperFactoryBean">
```

```
  <property name="simpleDateFormat" value="yyyy-MM-dd HH:mm:ss" />
```

```
</bean>
```

```

<bean id="xmlMapper" parent="jsonMapper">
  <property name="createXmlMapper" value="true"/>
</bean>

<mvc:resources mapping="/webjars/**" location="classpath:META-INF/
resources/webjars/" />

<jpa:repositories base-package="com.apress.todo.repository" />

<jdbc:embedded-database id="dataSource" type="H2">
  <jdbc:script location="classpath:META-INF/sql/schema.sql" />
  <jdbc:script location="classpath:META-INF/sql/data.sql" />
</jdbc:embedded-database>

<bean id="jpaVendorAdapter"

  class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
  <property name="showSql" value="true" />
</bean>

<bean id="entityManagerFactory"

  class="org.springframework.orm.jpa.
  LocalContainerEntityManagerFactoryBean">
  <property name="dataSource" ref="dataSource" />
  <property name="jpaVendorAdapter" ref="jpaVendorAdapter"/>
</bean>

<bean id="transactionManager"

  class="org.springframework.orm.jpa.JpaTransactionManager">
  <property name="entityManagerFactory" ref="entityManagerFactory"/>
</bean>

<tx:annotation-driven transaction-manager="transactionManager" />

<bean

  class="org.springframework.web.servlet.view.
  InternalResourceViewResolver">
  <property name="prefix" value="/WEB-INF/views/" />
  <property name="suffix" value=".jsp" />
</bean>

```

```

<bean id="h2WebServer" class="org.h2.tools.Server" factory-
method="createWebServer"
    init-method="start" destroy-method="stop">
    <constructor-arg value="-web,-webAllowOthers,-webDaemon,
    -webPort,8082" />
</bean>
</beans>

```

Listing 1-3 shows the Spring web configuration. Take a look at all the XML namespaces it uses. This can be helpful because if you use an IDE with code completion, it gives you the components and their attributes for every entry. Let's analyze it.

- `<context:component-scan/>`. This tag tells the *Spring container* that it needs to scan all the classes; it looks for annotations, including `@Service` and `@Configuration`. This helps Spring to wire up all the Spring beans, so that your application can run. In this case, it scans for the marked classes at the `com.apress.todo.*` package level and all the subpackages.
- `<mvc:annotation-driven/>`. This tag tells the Spring container that this is a web app, and that it needs to look for every `@Controller` and `@RestController` class and their methods that have `@RequestMapping` or other Spring MVC annotations, so it can create the necessary MVC beans for accepting requests from the user.
- `<mvc:message-converters/>`. This tag informs the MVC beans about what to use for message conversion when there is a request. For example, if there is a request that has the HTTP header `Accept: application/xml`, it responds as XML, the same way as when it has `application/json`.
- `jsonMapper` and `xmlMapper` beans. The classes are Spring beans that help format the data and create the right mapper.
- `<mvc:resources/>`. This tag tells the Spring MVC which resources to use, and where to find them. In this case, this app is using *WebJars* (declared in the `pom.xml` file).

- `<jpa:repositories/>`. This tag tells the Spring container and the Spring Data module where the interfaces that extend the `CrudRepository` interface are located. In this case, it looks for them in the `com.apress.todo.repository.*` package level.
- `<jdbc:embedded-database/>`. Because this app is using JPA and the H2 driver for an in-memory database, this tag is just a declaration to use a utility that can execute SQL script at startup; and in this case, it creates the `todo` table and inserts some records.
- `jpaVendorAdapter` bean. This bean declaration is needed for using the JPA implementation; in this case, it is Hibernate (a dependency used in the `pom.xml` file). In other words, the Hibernate framework is used as an implementation of Java Persistence API (JPA).
- `EntityManagerFactory` bean. For every JPA implementation, it is necessary to create an Entity Manager that holds all the sessions and executes all the SQL statements on the app's behalf.
- `TransactionManager` bean. The app needs to have a transaction, because we don't want to have duplicates or bad data, right? We need to apply and be compliant with ACID (Atomicity, Consistency, Isolation, Durability), so we need transactions.
- `<tx:annotation-driven/>`. This annotation sets up all the transactions based on the previous declarations.
- `viewResolver` bean. It is necessary to state which kind of a view engine the web app will use because there are a lot of options, such as Java Server Faces, JSP, and so forth.
- `h2WebServer` bean. This bean sets up the H2 engine so that it can be accessed within the application.

As you can see, this part requires a little bit of knowledge on how to wire up Spring. If you want to understand more, I recommend several books from the Apress, including *Pro Spring 5*, by I. Cosmina, et al.

I want to show you what you need to do to run a simpler REST API; and believe me, if you think that this is too much, try to do the same with Java EE with all the features this app has (MVC, JPA, SQL initialization, JSP, transactions).

Let's review the SQL scripts that execute at startup. Create the two files in the resources/META-INF/sql folder (see Listing 1-4 and Listing 1-5).

Listing 1-4. todo/src/main/resources/META-INF/sql/schema.sql

```
create table todo (
  id varchar(36) not null,
  description varchar(255) not null,
  created timestamp,
  modified timestamp,
  completed boolean,
  primary key (id)
);
```

As you can see, it's very straightforward for a SQL table creation.

Listing 1-5. todo/src/main/resources/META-INF/sql/data.sql

```
insert into todo values ('7fd921cfd2b64dc7b995633e8209f385', 'Buy
Milk', '2018-09-23 15:00:01', '2018-09-23 15:00:01', false);
insert into todo values ('5820a4c2abe74f409da89217bf905a0c', 'Read a
Book', '2018-09-02 16:00:01', '2018-09-02 16:00:01', false);
insert into todo values ('a44b6db26aef49e39d1b68622f55c347', 'Go to Spring
One 2018', '2018-09-18 12:00:00', '2018-09-18 12:00:00', false);
```

And, of course, some SQL statements and ToDo's.

Is important to know that JPA requires a persistence unit where you can configure things, such as which managed classes are part of the persistence unit, how classes are mapped to database tables, datasource connections, and so forth. So, it is necessary create one. You can create the persistence.xml file in the resources/META-INF/ folder (see Listing 1-6).

Listing 1-6. todo/src/main/resources/META-INF/persistence.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
```

```

http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd" version="1.0">
  <persistence-unit name="todo">
    <description>My Persistence Unit</description>
  </persistence-unit>
</persistence>

```

It is not necessary declare mapped classes or connections here because the Spring Data module takes care of it; you only need to declare a persistence-unit name.

Next, it is important to have to logging for the app, not only for debugging but you can use it to learn what's going on in your app. Create the `logback.xml` file in the resources folder (see Listing 1-7).

Listing 1-7. `todo/src/main/resources/logback.xml`

```

<?xml version="1.0" encoding="UTF-8"?>
<configuration xmlns="http://ch.qos.logback/xml/ns/logback"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ch.qos.logback/xml/ns/logback
http://ch.qos.logback/xml/ns/logback/logback.xsd">

  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
    <layout class="ch.qos.logback.classic.PatternLayout">
      <Pattern>
        %d{yyyy-MM-dd HH:mm:ss} %-5level %logger{36} - %msg%n
      </Pattern>
    </layout>
  </appender>

  <logger name="org.springframework" level="info" additivity="false">
    <appender-ref ref="STDOUT" />
  </logger>

  <logger name="org.springframework.jdbc" level="debug" additivity="false">
    <appender-ref ref="STDOUT" />
  </logger>

  <logger name="com.apress.todo" level="debug" additivity="false">
    <appender-ref ref="STDOUT" />
  </logger>

```

```

<root level="error">
  <appender-ref ref="STDOUT" />
</root>

</configuration>

```

Again, nothing fancy here. Note that the logging level for the `com.apress.todo` is set to `DEBUG`.

Classes

Next, it's time to create the actual code for the `ToDo` REST API. Let's start by creating the domain model: the `ToDo` domain class. Create the classes in the `src/main/java` folder. Remember that the Maven tool didn't create this structure; we need to create it manually (see Listing 1-8).

Listing 1-8. `todo/src/main/java/com/apress/todo/domain/ToDo.java`

```

package com.apress.todo.domain;

import org.hibernate.annotations.GenericGenerator;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import java.sql.Timestamp;

@Entity
public class ToDo {

    @Id
    @GeneratedValue(generator = "system-uuid")
    @GenericGenerator(name = "system-uuid", strategy = "uuid")
    private String id;
    private String description;

    private Timestamp created;
    private Timestamp modified;

    private boolean completed;

```



```
public Todo() {  
}  
  
public String getId() {  
    return id;  
}  
  
public void setId(String id) {  
    this.id = id;  
}  
  
public String getDescription() {  
    return description;  
}  
  
public void setDescription(String description) {  
    this.description = description;  
}  
  
public Timestamp getCreated() {  
    return created;  
}  
  
public void setCreated(Timestamp created) {  
    this.created = created;  
}  
  
public Timestamp getModified() {  
    return modified;  
}  
  
public void setModified(Timestamp modified) {  
    this.modified = modified;  
}  
  
public boolean isCompleted() {  
    return completed;  
}
```

```

    public void setCompleted(boolean completed) {
        this.completed = completed;
    }
}

```

As you can see, it is just a regular Java class, but because this app persists data (in this case, the `ToDo`'s) it is necessary to mark the class with the `@Entity` annotation and declare the *primary key* with an `@Id` annotation. This class also uses extra annotation to generate a 36-random-character GUID for the primary key.

Next, let's create a repository that has all the CRUD actions. Here the app uses the power of the Spring Data module, which hides all the boilerplate mapping classes with tables and keep sessions, and even does transactions. The Spring Data implements all the CRUD; in other words, you don't need to worry how to save, update, delete, and find records.

Create the `ToDoRepository` interface that extends from `CrudRepository` interface (see Listing 1-9).

Listing 1-9. `todo/src/main/java/com/apress/todo/repository/ToDoRepository.java`

```

package com.apress.todo.repository;

import com.apress.todo.domain.ToDo;
import org.springframework.data.repository.CrudRepository;

public interface ToDoRepository extends CrudRepository<ToDo,String> {
}

```

Listing 1-9 shows an interface. This `ToDoRepository` interface extends from `CrudRepository<T,K>`, which is a generic interface. The `CrudRepository` is asking for a domain class and the primary key type; in this case, the domain class is the `ToDo` class and the primary key type is a `String` (the one marked with the `@Id` annotation).

In the XML configuration, you used the `<jpa:repositories/>` tag. That tag is pointing to the `ToDoRepository` package, which means that Spring Data keeps record, and it wires up everything related to interfaces that extend `CrudRepository` interfaces.

Next, let's create the web controller that accepts requests from users. Create the `ToDoController` class (see Listing 1-10).

Listing 1-10. todo/src/main/java/com/apress/todo/controller/ToDoController.

java

```
package com.apress.todo.controller;

import com.apress.todo.domain.ToDo;
import com.apress.todo.repository.ToDoRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpStatus;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestHeader;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;

import javax.servlet.http.HttpServletRequest;

@Controller
@RequestMapping("/")
public class ToDoController {

    private ToDoRepository repository;

    @Autowired
    public ToDoController(ToDoRepository repository) {
        this.repository = repository;
    }

    @GetMapping
    public ModelAndView index(ModelAndView modelAndView, HttpServletRequest
request) {
        modelAndView.setViewName("index");
        return modelAndView;
    }
}
```

```

@RequestMapping(value = "/toDos", method = { RequestMethod.GET },
    produces = {
        MediaType.APPLICATION_JSON_UTF8_VALUE, MediaType.APPLICATION_
            XML_VALUE, MediaType.TEXT_XML_VALUE})
public ResponseEntity<Iterable<ToDo>> getToDos(@RequestHeader
    HttpHeaders headers) {
    return new ResponseEntity<Iterable<ToDo>>(this.repository.findAll(),
        headers, HttpStatus.OK);
}
}

```

Listing 1-10 shows the web controller. Take time to review it. Here we need an entire book to describe all the Spring MVC modules and every feature.

The important thing here is that the class is marked with the `@Controller` annotation. Remember the `<mv:annotation-driven/>` tag? This tag finds each class marked `@Controller` and registers the controllers with all the methods that have the `@GetMapping`, `@RequestMapping`, and `@PostMapping` annotations to accept requests based on the path defined. In this case, only the `/` and the `/toDos` paths are defined.

This class uses a constructor that has `ToDoRepository` as the parameter. This is injected by the Spring container thanks to the `@Autowired` annotation. This annotation can be omitted if you are using the Spring 4.3 version; by default, Spring container identifies that a constructor needs dependencies and it injects them automatically. It is like saying, “Hey, Spring container, I need the `ToDoRepository` bean to be injected because I will use it.” This is how Spring uses the dependency injection (there is also method injection, field injection, and setter injection).

`@GetMapping` (`@RequestMapping` does the same, by default) responds to the `/` path and the name of a view; in this case, it returns the index name that corresponds to the `WEB-INF/view/index.jsp` JSP page. `@RequestMapping` is another way to do the same (`@GetMapping`) but this time it is declaring the `/toDos` path. This method response depends on the kind of header the requester sends, such as `application/json` or `application/xml`. It is using `ResponseEntity` as response; it uses the repository instance to call the `findAll` methods that return all the `ToDo`’s from the database, because the configuration declared in the JSON and XML mappers that the engine takes care of that conversion.

Again, take the time to analyze what’s is happening. And after you run the app you can play around with all these annotations.

Next, let's create the view, which is the JSP that is called when the / path is requested. Create `index.jsp` in the `WEB-INF/views` folder (see Listing 1-11).

Listing 1-11. `todo/src/main/webapp/WEB-INF/views/index.jsp`

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<!doctype html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Simple Directory Web App</title>
  <link rel="stylesheet" type="text/css"
        href="webjars/bootstrap/3.3.7/css/bootstrap.min.css">
  <link rel="stylesheet" type="text/css"
        href="webjars/bootstrap/3.3.7/css/bootstrap-theme.min.css">
</head>
<body>
<div class="container theme-showcase" role="main">
  <div class="jumbotron">
    <h1>ToDo Application</h1>
    <p>A simple Rest API Spring MVC application</p>
  </div>

  <div class="page-header">
    <h1>API</h1>
    <a href="toDos">Current ToDos</a>
  </div>

</div>
</body>
</html>
```

I think the only thing to notice here is the use of the resources, like the WebJars. The app is using Bootstrap CSS. But where are these resources coming from? First, the `org.webjars:bootstrap` dependency in `pom.xml` is declared. Second, the `<mvc:resources/>` tag was used in the configuration to state where to find these resources.

Running the App

You have finished all the configuration and code needed to run the app. Now, it is time for the application server. To run this application, follow these steps.

1. Open a terminal and go to your root project (todo/). Execute the next maven command.

```
$ mvn clean package
```

This command packages your app in a WAR file (web archive) that is ready to be deployed in an application server. The file is in the `target/` folder, and it is named `todo.war`.

2. Download the Tomcat application server. (You don't need heavy application servers to run this app; a light Tomcat will do). You can download it from <https://tomcat.apache.org/download-90.cgi>.
3. Unzip and install it in any directory.
4. Copy the `target/todo.war` in the `<tomcat-installation>/webapps/` folder.
5. Run your tomcat. Go to a browser, and click the `http://localhost:8080/todo` URL (see Figure 1-2).

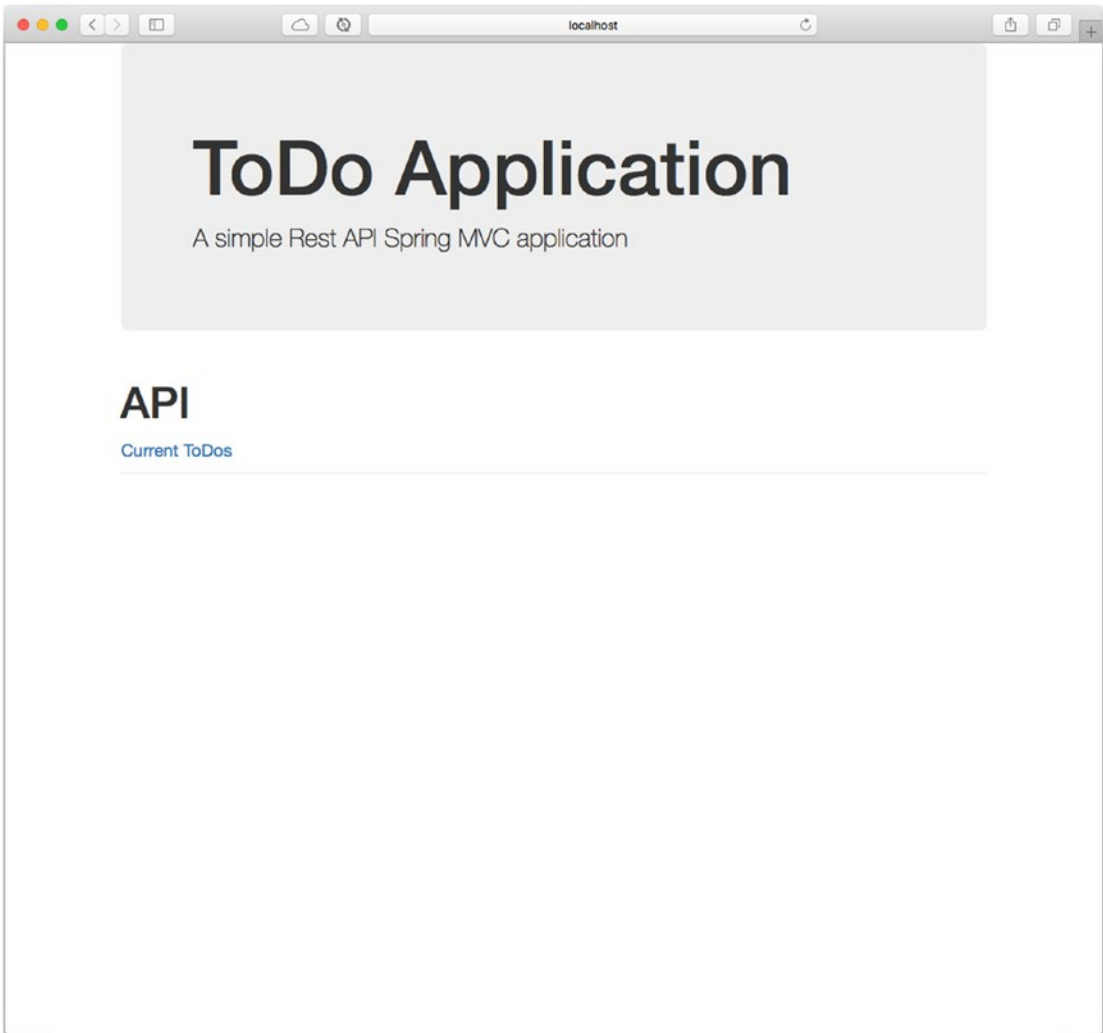


Figure 1-2. *http://localhost:8080/todo/*

If you click the link, you should have an XML response (see Figure 1-3).

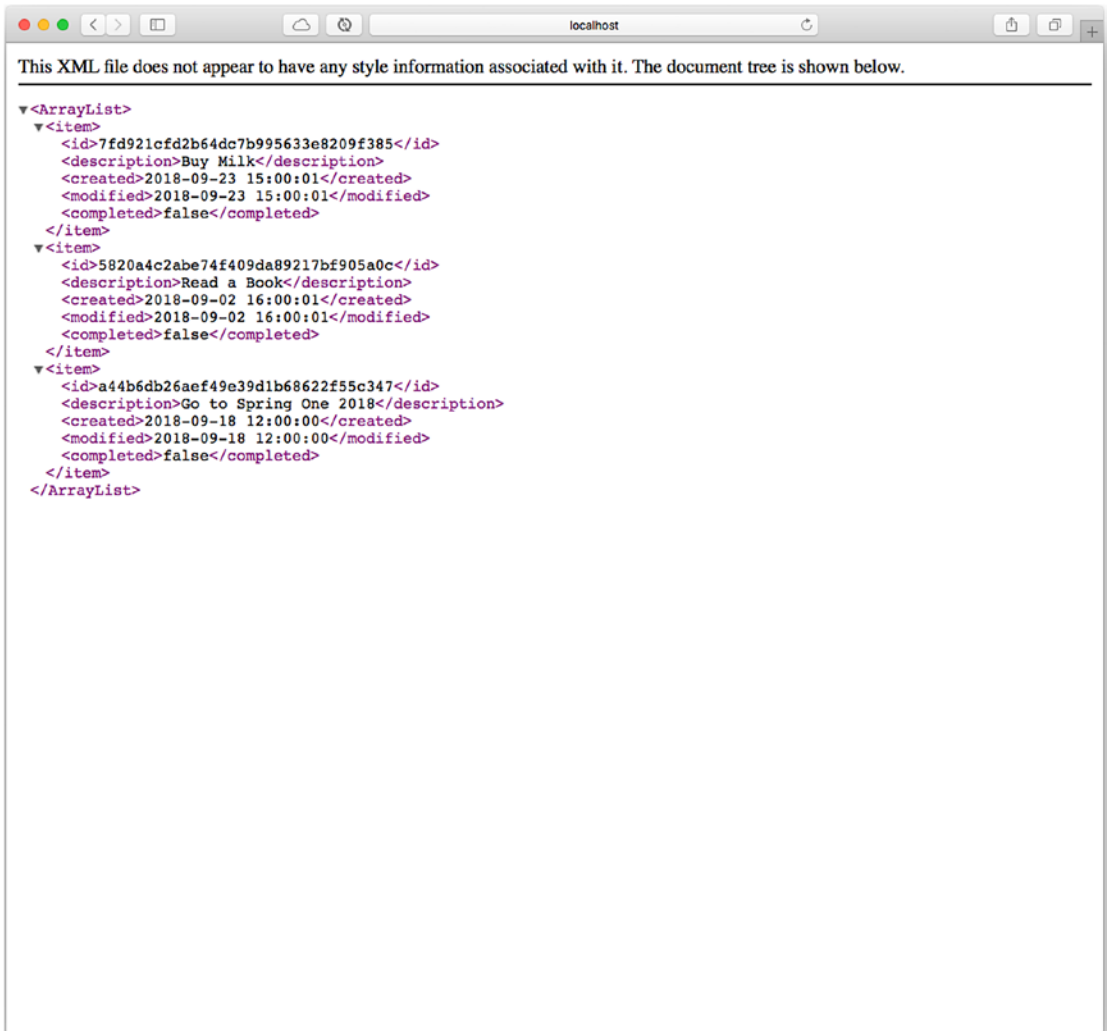


Figure 1-3. `http://localhost:8080/todo/todos`

How can you get the JSON response? Open a terminal and execute the following command.

```

$ curl -H "Accept: application/json" localhost:8080/todo/todos
[ {
  "id" : "7fd921cfd2b64dc7b995633e8209f385",
  "description" : "Buy Milk",
  "created" : "2018-09-23 15:00:01",

```



```

    "modified" : "2018-09-23 15:00:01",
    "completed" : false
  }, {
    "id" : "5820a4c2abe74f409da89217bf905a0c",
    "description" : "Read a Book",
    "created" : "2018-09-02 16:00:01",
    "modified" : "2018-09-02 16:00:01",
    "completed" : false
  }, {
    "id" : "a44b6db26aef49e39d1b68622f55c347",
    "description" : "Go to Spring One 2018",
    "created" : "2018-09-18 12:00:00",
    "modified" : "2018-09-18 12:00:00",
    "completed" : false
  } ]

```

You can test with `application/xml` instead and see the same result as the browser. Congratulations! You just have created your first Spring MVC REST API app.

Using Java Config

You may be thinking that XML is too verbose for creating a configuration. Well, sometimes it can be, but Spring has another way to configure the Spring container, which is through annotation and Java config classes.

If you want to try this, you can create the `ToDoConfig` class and add the code shown in Listing 1-12.

Listing 1-12. `todo/src/main/java/com/apress/todo/config/ToDoConfig.java`

```

package com.apress.todo.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.jpa.repository.config.
EnableJpaRepositories;
import org.springframework.http.converter.HttpMessageConverter;

```

```

import org.springframework.http.converter.json.Jackson2ObjectMapperBuilder;
import org.springframework.http.converter.json.
MappingJackson2HttpMessageConverter;
import org.springframework.http.converter.xml.
MappingJackson2XmlHttpMessageConverter;
import org.springframework.jdbc.datasource.embedded.
EmbeddedDatabaseBuilder;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType;
import org.springframework.orm.jpa.JpaTransactionManager;
import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;
import org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter;
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.annotation.
EnableTransactionManagement;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;
import org.springframework.web.servlet.config.annotation.
ResourceHandlerRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
import org.springframework.web.servlet.resource.WebJarsResourceResolver;
import org.springframework.web.servlet.view.InternalResourceViewResolver;

import javax.sql.DataSource;
import java.text.SimpleDateFormat;
import java.util.List;

```

@Configuration

```
@EnableJpaRepositories(basePackages="com.apress.todo.repository")
```

```
@EnableTransactionManagement
```

```
@EnableWebMvc
```

```
public class ToDoConfig implements WebMvcConfigurer {
```

```
    @Override
```

```
    public void addResourceHandlers(ResourceHandlerRegistry registry) {  
        registry  
            .addResourceHandler("/webjars/**")  
    }
```

```

        .addResourceLocations("classpath:/META-INF/resources/webjars/",
            "/resources/", "/webjars/")
        .resourceChain(true).addResolver(new WebJarsResourceResolver());
    }

```

@Override

```

public void configureMessageConverters(List<HttpMessageConverter<?>>
    converters) {
    Jackson2ObjectMapperBuilder builder = new
        Jackson2ObjectMapperBuilder();
    builder.indentOutput(true).dateFormat(new
        SimpleDateFormat("yyyy-MM-dd HH:mm:ss"));
    converters.add(new MappingJackson2HttpMessageConverter(builder.
        build()));
    converters.add(new MappingJackson2XmlHttpMessageConverter(builder.
        createXmlMapper(true).build()));
}

```

@Bean

```

public InternalResourceViewResolver jspViewResolver() {
    InternalResourceViewResolver bean = new InternalResourceViewResolver();
    bean.setPrefix("/WEB-INF/views/");
    bean.setSuffix(".jsp");
    return bean;
}

```

@Bean

```

public DataSource dataSource() {
    EmbeddedDatabaseBuilder builder = new EmbeddedDatabaseBuilder();
    return builder.setType(EmbeddedDatabaseType.H2).
        addScript("META-INF/sql/schema.sql")
            .addScript("META-INF/sql/data.sql").build();
}

```

@Bean

```

public LocalContainerEntityManagerFactoryBean entityManagerFactory() {

```

```

        HibernateJpaVendorAdapter vendorAdapter = new
        HibernateJpaVendorAdapter();
        vendorAdapter.setShowSql(true);

        LocalContainerEntityManagerFactoryBean factory = new
        LocalContainerEntityManagerFactoryBean();
        factory.setJpaVendorAdapter(vendorAdapter);
        factory.setDataSource(dataSource());
        return factory;
    }

    @Bean
    public PlatformTransactionManager transactionManager() {
        JpaTransactionManager txManager = new JpaTransactionManager();
        txManager.setEntityManagerFactory(entityManagerFactory().
        getNativeEntityManagerFactory());
        return txManager;
    }
}

```

Listing 1-12 is actually the same as the XML configuration, but this time, it is using a Java Config class, where programmatically we declare Spring beans, and it is necessary to override some web configuration.

If you want to run it to test this JavaConfig class, you need to do something. Open `dispatcherServlet-servlet.xml`, which should look like the following.

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:context="http://www.springframework.org/schema/context"
        xsi:schemaLocation="
            http://www.springframework.org/schema/beans http://www.
            springframework.org/schema/beans/spring-beans.xsd
            http://www.springframework.org/schema/context http://www.
            springframework.org/schema/context/spring-context-4.3.xsd">
    <context:component-scan base-package="com.apress.todo" />
</beans>

```

At the end, it is necessary to tell Spring where to find the `@Configuration` marked class (another alternative is to use the `WebApplicationInitializer` class); once it finds it, it wires up everything based on the declarations of the Java Config class.

Remember to clean and repackage your app with the `mvn clean package` to generate the WAR file again. You can run it, and you get the same result as using the XML configuration.

So, what do you think of the Spring Framework? Yes, you need to understand what is happening. You need to know how the Spring beans lifecycle works and how the dependency injection is being used. Also, it is important to know a little AOP (aspect-oriented programming) because it's part of the magic of wiring everything to work for us.

Do you think it's too much? Well, if you try to make the same app with a regular Java 2 EE profile, it will be even more work. Remember, it's not only exposing a REST API, but working with a database, transactions, message converters, view resolvers, and more; that's why with Spring, web apps are easier to create.

But guess what? Spring Boot does all the boilerplate configuration for you, which speeds up development time by creating enterprise Spring apps!

Note Remember that you can get this book's source code from the Apress website or on GitHub at <https://github.com/Apress/pro-spring-boot-2>.

Summary

There is a lot to learn about the Spring Framework and the role it plays with Spring Boot. A single chapter won't be enough. So, if you want to know more about it, I encourage you to review the Spring documentation at <https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/>.

In the next chapter, we start with Spring Boot and learn how easy it is to create the same application we did in this chapter, but "à la Boot."

CHAPTER 2

Introduction to Spring Boot

In the previous chapter, I showed you what the Spring Framework is, some of its main features (such as the implementation of the dependency injection design pattern), and how to use it (by creating a simple web/data application and deploy it to a Tomcat server). I also showed you every step needed in creating Spring applications (e.g., the configuration options to add the various XML files, and how to run the app).

In this chapter, I show you what Spring Boot is—its main components, how to use it to create Spring applications, and how to run or deploy it. It's a more simplified way to create Spring apps. The rest of the book covers more details; this is only a small introduction to the Spring Boot technology.

Spring Boot

I can say that Spring Boot is the next chapter of the Spring Framework, but don't get me wrong: Spring Boot won't replace the Spring Framework because Spring Boot *is* the Spring Framework! You can look at Spring Boot as a new way to create Spring applications with ease.

Spring Boot simplifies the way we develop because it makes it easy to create production-ready Spring-based applications that you can “just run.” You will find out that with Spring Boot, you can create stand-alone applications with an embedded server (Tomcat by default, or Netty if you are using the new *web-reactive* modules), making them 100% runnable and deployable! I talk more about this in several chapters of the book. applications.

One of Spring Boot’s most important features is an *opinionated* runtime, which helps you follow the best practices for creating robust, extensible, and scalable Spring applications.

You can find the Spring Boot project at <https://projects.spring.io/spring-boot/>. Very extensive documentation is at <https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/>. The Spring Boot home page is shown in Figure 2-1.

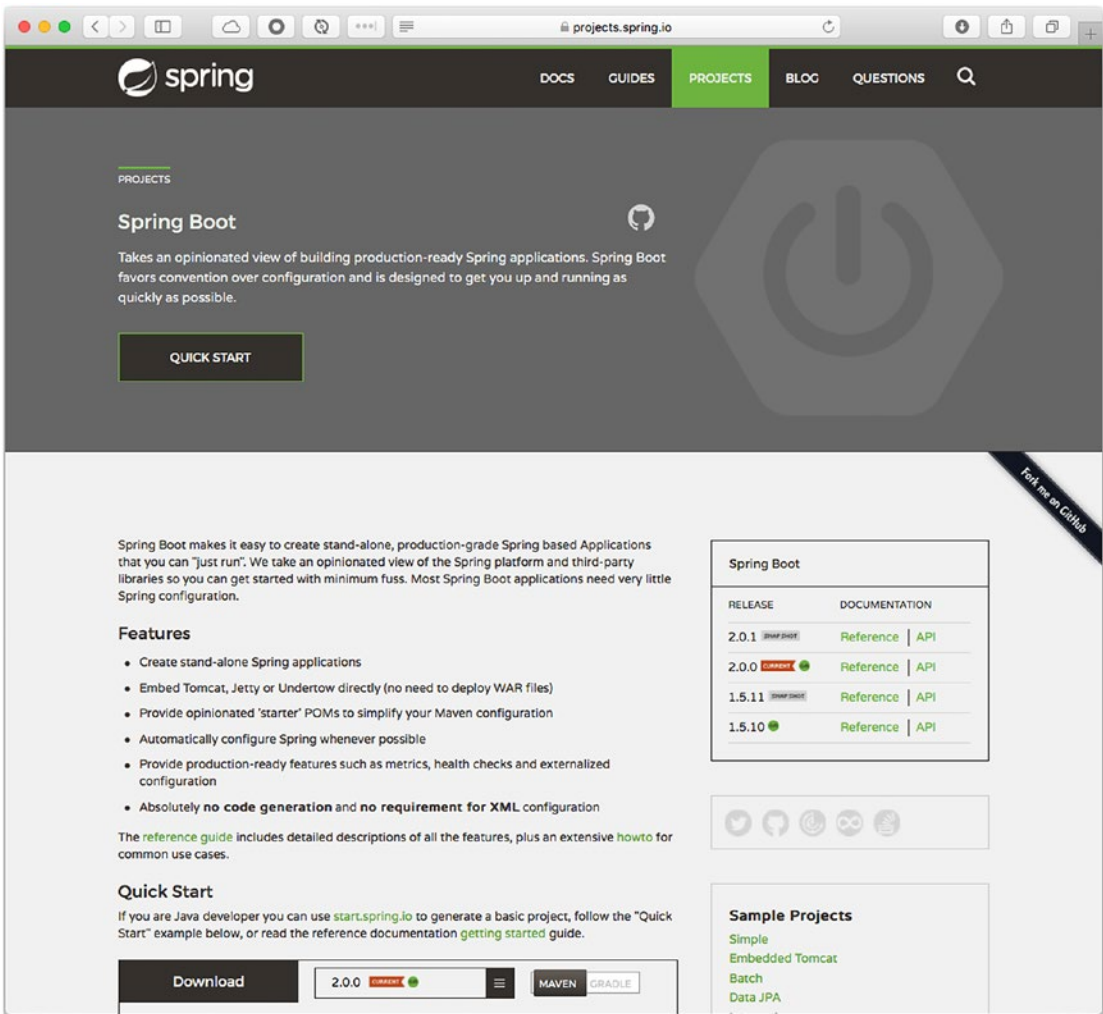


Figure 2-1. Spring Boot home page (<http://projects.spring.io/spring-boot/>)

Spring Boot to the Rescue

To create Spring apps, you need to know all the configuration techniques and/or requirements for the technology. There are a lot of steps required to run even the simplest Spring app. Four years ago, the Spring team brought out the first beta release, which I was lucky to test. The results were amazing. Now, with more features added to the technology, it has really become the “de facto” way to create Spring applications. Spring Boot makes it easier to create enterprise-ready apps.

If you take a look at the Spring Boot project webpage, a statement that says: *Absolutely no code generation and no requirement for XML configuration.* Maybe you are wondering about how you can create Spring apps and run them without any configuration. The Spring container at least needs to know how to wire up your classes, right? Or the Spring container needs to know how to use the technology you added to your app. Well, don’t worry. I will tell you all the secrets behind this amazing technology. But first, let’s create the simplest Spring web application possible (see Listing 2-1).

Listing 2-1. app.groovy

```
@RestController
class WebApp{
    @GetMapping("/")
    String welcome(){
        "<h1><font face='verdana'>Spring Boot Rocks!</font></h1>"
    }
}
```

Listing 2-1 is a Groovy application and the simplest possible Spring web application. Why Groovy? I always tell my students that if you know Java, then you know Groovy. Groovy removes all the Java boilerplate, and with a few lines of code, you have a web app (but don’t worry, this book mostly covers Java; except in the last chapter I talk about Groovy and Kotlin, a new addition to the Spring language support). How do you run it? It’s as simple as executing

```
$ spring run app.groovy
```

You should then see output logs with a Spring Boot banner, a Tomcat container initialization, and a note that the application has started on port 8080. If you open a browser and click `http://localhost:8080`, then you should see the text **Spring Boot Rocks!**

You may be saying, “Wait a minute! What is this `spring run` command? How can I install it? What else do I need? Is this Spring Boot?” Well, this is one of the many ways to create and run a Spring app. This was one of my first attempts to show the power of this technology (four years ago), a simple script that can run a full Spring web app. The Spring Boot team created *Spring Boot CLI*.

Spring Boot CLI

The Spring Boot CLI (command-line interface) is one of the many ways to create Spring apps, but this approach is normally used for prototype apps. You may consider it as a Spring Boot playground. A reference model is covered in the following sections. I just wanted to give you a small taste of the power of Spring Boot using simple Groovy or Java scripts. For me, the Spring Boot CLI is a fundamental piece of the Spring Boot ecosystem.

Now, let’s get back to the previous code. Did you notice that there were no imports in Listing 2-1? How does the Spring Boot CLI know about a web application and how to run it?

Spring Boot CLI inspects your code, and based on the Spring MVC annotations (the `@RestController` and `@GetMapping`), it tries to execute your code as a web application using an embedded Tomcat server, and runs the web app from within. The magic behind the scenes is that the Groovy programming language provides an easy way to intercept statements and create on-the-fly code by using an AST (abstract syntax tree); therefore, it is easy to inject the missing Spring code and run it. In other words, the Spring Boot CLI finds out about your app and it injects the missing pieces to have a full Spring web app up and running.

Remember when I mentioned that it can run also Java scripts? Let’s look at a Java version of the same web app. I’ll show you the code just for now; if you want to run these apps, you can read the appendix, where I explain how to install the Spring Boot CLI and its features (see Listing 2-2).

Listing 2-2. SimpleWebApp.java

```
package com.apress.spring;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
```

```

@RestController
@SpringBootApplication
public class SimpleWebApp {

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

    @RequestMapping("/")
    public String greetings(){
        return "<h1>Spring Boot Rocks in Java too!</h1>";
    }
}

```

Listing 2-2 shows the entry point for a Spring Boot application in Java. Primarily, it uses a `@SpringBootApplication` annotation and the `SpringApplication` singleton class in the main method, which executes the application. The `SpringApplication.run` method call accepts two parameters. The first parameter is the main configuration class that contains the `@Configuration` annotation (that happens to be the same name of the class; but more about this later on). The second parameter are the application arguments (that we review in the following chapters). As you can see from this Java version, we are using the Spring MVC annotations: the `@RestController` and the `@GetMapping`.

You can run this example by executing

```
$ spring run SimpleWebApp.java
```

If you open your browser and click `http://localhost:8080/`, you see the message “Spring Boot Rocks in Java too!”

If you want to set up your Spring Boot CLI, you can jump to the appendix, where I include a step-by-step installation, all its features, and the benefits of the Spring Boot CLI. For a quick prototype of Spring cloud applications, the Spring Boot CLI is the perfect player; that’s why I include the Spring Boot CLI in this book.

Spring Boot Application Model

Spring Boot defines a way to easily create Spring applications and a programming model that follows the best practices for Spring apps. To create a Spring Boot app, you need the following components:

- A build/dependency management tool, such as Maven or Gradle (Spring Boot also supports *Ant* and *Ivy*; in this book, you only need Maven or Gradle for each example).
- The right dependency management and plugins within the building tool. If you use Maven, a `<parent/>` tag is required (of course, there are more ways to configure a Spring Boot but adding a `<parent/>` tag is the easiest) and the `spring-boot-maven-plugin`. If you are using Gradle, you need to apply the `org.springframework.boot` and the `io.spring.dependency-management` plugins.
 - Add the required dependencies using `spring-boot-starters`.
- Create a main class that contains
 - The `@SpringBootApplication` annotation
 - The `SpringApplication.run` statement in the main method.

In the next section, we are going to create our first Spring Boot application, and I'll explain all the preceding components. It is very straightforward, but how do we start? Is there any tool that can help us start a Spring Boot project? The answer is yes! We can actually use the Spring Boot CLI because it provides a way to create Spring Boot projects. We also have IDEs (integrated development environments), such as *STS* (Spring Tool Suite <https://spring.io/tools>), *IntelliJ IDEA* from JetBrains (<https://www.jetbrains.com/idea/>), *NetBeans* (<https://netbeans.org>), GitHub *Atom* (<https://atom.io>), and Microsoft *VSCode* (<https://code.visualstudio.com>). *Atom* and *VSCode* have plugins for handling Spring Boot apps in a very light fashion; but *Spring Initializr* (<http://start.spring.io>) is my preference for starting a Spring Boot project. In this book, I use *IntelliJ IDEA*.

Let's look at how to use the Spring Boot Initializr web service by creating our first Spring Boot application.

My First Spring Boot Application

To create our first Spring Boot application, open your browser and go to <http://start.spring.io> (see Figure 2-2).

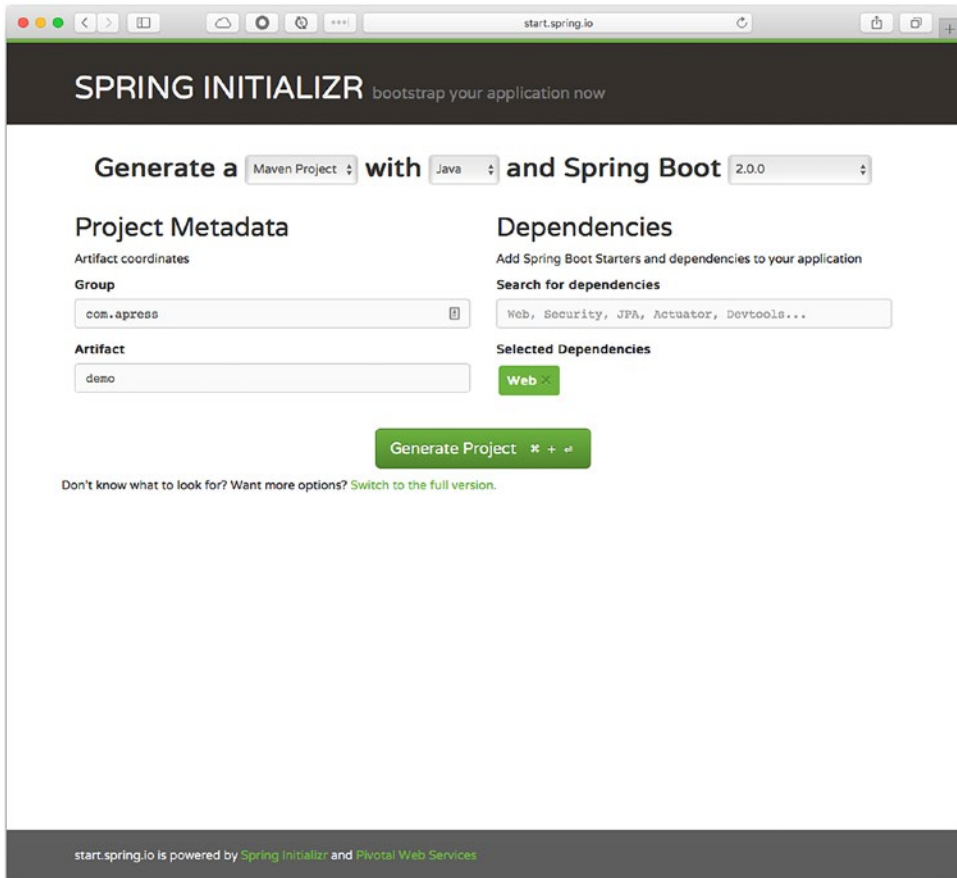


Figure 2-2. <http://start.spring.io>

Figure 2-2 shows the homepage for Spring Boot Initializr, a web service provided by Pivotal that helps you easily create Spring Boot projects.

1. Let's start by filling out the fields.
 - Group: com.apress
 - Artifact: demo
 - Dependencies: web

You can choose either a Maven or Gradle project type. You can choose the programming language (Java, Groovy, or Kotlin) and the Spring Boot version. Below the Generate Project button, there is a link that says “Switch to the full version” link, which shows the dependencies that you need. In this case, you can enter **Web** in the Dependencies field and hit Enter, as shown in Figure 2-2.

2. Click the Generate Project button to save a file named `demo.zip`.
3. Uncompress the `demo.zip` file and import the project into your favorite IDE (I use IntelliJ IDEA). If you look closely, you see that inside the `.zip` file there is a wrapper, depending on the project type you chose. If it is a Gradle project, then there is a *gradlew* (Gradle wrapper); if it is a Maven project, then it should be an *mvnw* (Maven wrapper). This means that you don’t need to install any of these building/management tools because Spring Boot Initializr brings them to you.
4. Inspect the build/dependencies management files. Open the `pom.xml` or `build.gradle` file.

If you selected Maven, see Listing 2-3.

Listing 2-3. Maven `pom.xml`

```
<?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</groupId>
  <artifactId>demo</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>demo</name>
  <description>Demo project for Spring Boot</description>
```

```

<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.0.0.RELEASE</version>
  <relativePath/> <!-- lookup parent from repository -->
</parent>

<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <project.reporting.outputEncoding>UTF-8</project.reporting.
  outputEncoding>
  <java.version>1.8</java.version>
</properties>

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

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

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

```

As you can see, we have part of the main components: the `<parent/>` tag, the `spring-boot-starter-web` dependency, and the `spring-boot-maven-plugin`.

If you selected Gradle, see Listing 2-4.

Listing 2-4. `build.gradle`

```

buildscript {
    ext {
        springBootVersion = '2.0.0.RELEASE'
    }
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-
            plugin:${springBootVersion}")
    }
}

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

group = 'com.apress'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = 1.8

repositories {
    mavenCentral()
}

dependencies {
    compile('org.springframework.boot:spring-boot-starter-web')
    testCompile('org.springframework.boot:spring-boot-starter-test')
}

```

The build.gradle file shows some of the components required: the org.springframework.boot and io.spring.dependency-management plugins, and the spring-boot-starter-web dependency.

5. Open the com.apress.demo.DemoApplication.java class (see Listing 2-5).

Listing 2-5. com.apress.demo.DemoApplication.java

```
package com.apress.demo;

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

@SpringBootApplication
public class DemoApplication {

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

As you can see, we have the other components to run our application: the @SpringBootApplication annotation and the SpringApplication.run statement.

6. Add a new class for the web controller that shows as text. Create the com.apress.demo.WebController.java class (see Listing 2-6).

Listing 2-6. com.apress.demo.WebController.java

```
package com.apress.demo;

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

@RestController
public class WebController {

    @GetMapping
    public String index(){
        return "Hello Spring Boot";
    }
}
```


This is very similar to the scripts—a simple `@RestController` that returns a `String`.

7. To run your application, you can use the IDE or you can go to the root of your project and execute the following command. For maven: `./mvnw spring-boot:run`, for Gradle: `./gradlew bootRun`.

Then you can go to your browser and click `http://localhost:8080`. You see the “Hello Spring Boot” text.

Congratulations! You have just created your first Spring Boot application.

Note All the companion code from the book is on the Apress site. In this case, I created two projects: one with Maven and the other with Gradle.

Why Spring Boot?

Why do we want to use Spring Boot? It is an amazing technology that is suitable for

- cloud native applications that follow the 12 factor patterns developed by the Netflix engineering team (<http://12factor.net>)
- better productivity by reducing time of development and deployment.
- Enterprise-Production-Ready Spring applications.
- non-functional requirements, like the Spring Boot Actuator (a module that provides metrics with the new platform-agnostic *Micrometer* (<https://micrometer.io>), health checks, and management) and embedded containers for running web applications (Tomcat, Netty, Undertow, Jetty, etc.).
- Microservices, which are getting attention for creating scalable, highly available, and robust applications. Spring Boot allows developers to focus only on business logic, leaving the heavy lifting to the Spring Framework.

Spring Boot Features

Spring Boot has a lot of features, which I show you in the following chapters, but I can describe some of them in this section.

Spring Boot

- Offers the `SpringApplication` class. I showed you that in a Java Spring Boot application, the main method executes this singleton class. This particular class provides a convenient way to initiate a Spring application.
- Allows you to create applications without requiring any XML configuration. Spring Boot doesn't do any code generation.
- Provides a fluent builder API through the `SpringApplicationBuilder` singleton class, which allows you to create hierarchies with multiple application contexts. This particular feature is more related to the Spring Framework and how it works internally. If you are a Spring developer, I explain this feature in the following chapters, but if you are new to Spring and Spring Boot, then you only need to know that you can extend Spring Boot to get more control over your applications.
- Offers more ways to configure the Spring application events and listeners.
- Provides “opinionated” technology; this particular feature attempts to create the right type of application, either as a web application (embedded Tomcat, Netty, Undertow, or Jetty container) or as a single application.
- Offers the `org.springframework.boot.ApplicationArguments` interface, which allows access to any application argument. This is a useful feature when you try to run your application with parameters.
- Allows you to execute code after the application has started. The only thing you need to do is to implement the `CommandLineRunner` interface and provide the implementation of the `run(String ...args)` method. A particular example is to initialize records in a database during the start, or maybe you want to check if services are running before your application executes.

- Allows you to externalize configurations by using `application.properties` or `application.yml` files. More about this in the following chapters.
- Allows you to add administration-related features, normally through JMX, by enabling the `spring.application.admin.enabled` property in the `application.properties` or `application.yml` files.
- Allows you to have *profiles* that help your application to run in different environments.
- Allows you to configure and use logging in a very simple way.
- Provides a simple way to configure and manage your dependencies by using starter poms. In other words, if you are going to create a web application, you only need to include the `spring-boot-start-web` dependency in your Maven `pom.xml` or `build.gradle` file.
- Provides out-of-the box non-functional requirements by using the Spring Boot Actuator with the new Micrometer platform-agnostic framework, which allows you to instrument your apps.
- Provides `@Enable<feature>` annotations that help you include, configure, and use technologies such as databases (SQL and NoSQL), caching, scheduling, messaging, Spring Integration, Spring Batch, Spring Cloud, and more.

Spring Boot has all these features and more. I get into more detail of these features in the following chapters. Now, it is time to start learning more about of Spring Boot by seeing how it works internally.

Summary

In this chapter, I gave you a quick overview of the Spring Boot technology, which specializes in creating Spring enterprise-ready applications with ease.

In the following chapters, I show you the internals of Spring Boot and the behind-the-scenes magic to create the right application based on your dependencies and code. I talk about all the Spring Boot cool features as you create different projects.

CHAPTER 3

Spring Boot Internals and Features

In the previous chapter, I gave a quick overview of Spring Boot, the main components for creating a Spring Boot app, and discussed how easy it is to use Spring Initializr for creating Spring Boot projects.

In this chapter, I show you what is happening behind the scenes when Spring Boot starts your application. Everything is about auto-configuration! I start with the Groovy scripts (again, you can jump to the appendix section and install the Spring Boot CLI). I use a regular Java project, as with the Spring Boot app in Chapter 2. Let's start with learning how auto-configuration works.

Auto-Configuration

Auto-configuration is one of the important features in Spring Boot because it configures your Spring Boot application according to your classpath, annotations, and any other configuration declarations, such as JavaConfig classes or XML.

Listing 3-1 is the same example in previous chapters, but in this case, I use it to explain what happens behind the scenes when Spring Boot runs it.

Listing 3-1. app.groovy

```

@RestController
class WebApp{

    @GetMapping('/')
    String index(){
        "Spring Boot Rocks"
    }
}

```

You run this program using the Spring Boot CLI (command-line interface) with

```
$ spring run app.groovy
```

Spring Boot won't generate any source code but it adds some on the fly. This is one of the advantages of Groovy: you have access to the AST (abstract syntax tree) at runtime. Spring Boot starts by importing missing dependencies, such as the `org.springframework.web.bind.annotation.RestController` annotation, among other imports.

Next, it identifies that you need a *spring-boot-starter-web* (I talk more about it in the following sections) because you marked your class and method with the `@RestController` and the `@GetMapping` annotations, respectively. It adds to the code the `@Grab("spring-boot-web-starter")` annotation (useful for imports in Groovy scripts).

Next, it adds the necessary annotation that triggers auto-configuration, the `@EnableAutoConfiguration` annotation (later, I talk about this annotation, which happens to be the *magic behind* Spring Boot), and then it adds the main method that is the entry point for the application. You can see the result code in Listing 3-2.

Listing 3-2. app.groovy Modified by Spring Boot

```

import org.springframework.web.bind.annotation.RestController
// Other Imports

@Grab("spring-boot-web-starter")
@EnableAutoConfiguration
@RestController
class WebApp{
    @GetMapping("/")

```

```

String greetings(){
    "Spring Boot Rocks"
}

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

```

Listing 3-2 shows the actual modified program that Spring Boot runs. You can see in action how the auto-configuration works, but by running Listing 3-1 with the `--debug` parameter. Let's take a look.

```

$ spring run app.groovy --debug
...
DEBUG 49009 --- [] autoConfigurationReportLoggingInitializer :
=====
AUTO-CONFIGURATION REPORT
=====

Positive matches:
-----
//You will see all the conditions that were met to enable a Web
application. And this is because you have the //@RestController annotation.

Negative matches:
-----
//You will find all the conditions that failed. For example you will find
that the ActiveMQAutoConfiguration class did //not match, because you don't
have any reference of the ActiveMQConnectionFactory.

```

Review the output from the command in your terminal. Note all the positive and negative matches that Spring Boot did before running this simple application. Because you are running the Spring Boot CLI, it's doing a lot by trying to guess what kind of application you want to run. When you create either a Maven or Gradle project, and you specify dependencies (`pom.xml` or `build.gradle`, respectively) you are helping Spring Boot to make a decision based on your dependencies.

Disable a Specific Auto-Configuration

In Chapter 2, I talked about the `@SpringBootApplication` annotation, which is one of the main components of a Spring Boot app. This annotation is equivalent to declaring the `@Configuration`, `@ComponentScan`, and `@EnableAutoConfiguration` annotations. Why am I mentioning this? Because you can disable a specific auto-configuration by adding the `exclude` parameter by using either the `@EnableAutoConfiguration` or the `@SpringBootApplication` annotations in your class. Let's look at an example in a Groovy script in Listing 3-3.

Listing 3-3. app.groovy

```
import org.springframework.boot.autoconfigure.jms.activemq.
ActiveMQAutoConfiguration

@RestController
@EnableAutoConfiguration(exclude=[ActiveMQAutoConfiguration.class])
class WebApp{

    @RequestMapping("/")
    String greetings(){
        "Spring Boot Rocks"
    }
}
```

Listing 3-3 shows the `@EnableAutoConfiguration` annotation that has the `exclude` parameter. This parameter receives an array of auto-configuration classes. If you run this again with the following, you see the exclusion on what you did.

```
$ spring run app.groovy --debug
...
Exclusions:
-----

org.springframework.boot.autoconfigure.jms.activemq.
ActiveMQAutoConfiguration
...
```

This is a very useful technique for Groovy scripts when you want Spring Boot to skip certain and unnecessary auto-configurations.

Let's look at how you can use this on a Java Spring Boot app (see Listing 3-4).

Listing 3-4. DemoApplication.java: Spring Boot Snippet

```
package com.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.autoconfigure.jdbc.
DataSourceAutoConfiguration;
import org.springframework.boot.autoconfigure.jms.activemq.
ActiveMQAutoConfiguration;

@SpringBootApplication(exclude={ActiveMQAutoConfiguration.class,DataSourceA
utoConfiguration.class})
public class DemoApplication {

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

Listing 3-4 shows a Java version; in this example, the main class is declaring only the `@SpringBootApplication` annotation, and within this annotation you can exclude the auto-configuration classes. Listing 3-4 shows two classes being excluded: the `ActiveMQAutoConfiguration` and the `DataSourceAutoConfiguration`. Why is `@EnableAutoConfiguration` annotation not used? Remember that the `@SpringBootApplication` annotation inherits `@EnableAutoConfiguration`, `@Configuration`, and `@ComponentScan`, so that's why you can use the `exclude` parameter within the `@SpringBootApplication` annotation.

When running a Maven or Gradle project (using the example Listing 3-4) with the `debug` option, you see in the console output something like this:

```
...
Exclusions:
-----

org.springframework.boot.autoconfigure.jms.activemq.ActiveMQAutoConfiguration
org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration
...
```


@EnableAutoConfiguration and @Enable<Technology> annotations

The Spring Framework and some of its modules, such as Spring Data, Spring AMQP, and Spring Integration, provide @Enable<Technology> annotations; for example, @EnableTransactionManagement, @EnableRabbit, and @EnableIntegration are part of the modules mentioned. Within Spring applications, you can use these annotations to follow the *convention over configuration* pattern, making your apps easier to develop and maintain, and without worrying too much about its configuration.

Spring Boot takes advantage of these annotations, which are used within the @EnableAutoConfiguration annotation to do the auto-configuration. Let's take a closer look at the @EnableAutoConfiguration annotation to see the logic behind it and where the @Enable<Technology> annotations fit (see Listing 3-5).

Listing 3-5. org.springframework.boot.autoconfigure.EnableAutoConfiguration.
java

```
...
@AutoConfigurationPackage
@Import(AutoConfigurationImportSelector.class)
public @interface EnableAutoConfiguration {

    Class<?>[] exclude() default {};

    String[] excludeName() default {};

}
```

Listing 3-5 shows the @EnableAutoConfiguration annotation; as you already know, this class attempts to configure the beans that your application likely needs. The auto-configuration classes are applied based on the *classpath* and which beans your app has defined, but what this makes more powerful is the org.springframework.boot.autoconfigure.AutoConfigurationImportSelector class that finds all the necessary configuration classes.

The AutoConfigurationImportSelector class has several methods, but one of the most important for the auto-configuration is the getCandidateConfigurations method (see Listing 3-6).

Listing 3-6. org.springframework.boot.autoconfigure.

AutoConfigurationImportSelector Snippet

```

...
protected List<String> getCandidateConfigurations(AnnotationMetadata metadata,
        AnnotationAttributes attributes) {
    List<String> configurations = SpringFactoriesLoader.loadFactoryNames(
        getSpringFactoriesLoaderFactoryClass(),
        getBeanClassLoader());

    Assert.notEmpty(configurations,
        "No auto configuration classes found in META-INF/spring.factories. If you
        are using a custom packaging, make sure that file is correct.");
    return configurations;
}
...

```

Listing 3-6 shows you a snippet of the `AutoConfigurationImportSelector` class, where the `getCandidateConfigurations` method returns a `SpringFactoriesLoader.loadFactoryNames`. The `SpringFactoriesLoader.loadFactoryNames` looks for the `META-INF/spring.factories` defined in the `spring-boot-autoconfigure` jar (see Listing 3-7 for its contents).

Listing 3-7. spring-boot-autoconfigure-<version>.jar/META-INF/spring.factories Snippet

```

# Initializers
org.springframework.context.ApplicationContextInitializer=\
org.springframework.boot.autoconfigure.
SharedMetadataReaderFactoryContextInitializer,\
org.springframework.boot.autoconfigure.logging.
ConditionEvaluationReportLoggingListener

# Application Listeners
org.springframework.context.ApplicationListener=\
org.springframework.boot.autoconfigure.BackgroundPreinitializer

```

```

# Auto Configure
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
org.springframework.boot.autoconfigure.admin.\
SpringApplicationAdminJmxAutoConfiguration,\
org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\
org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration,\
org.springframework.boot.autoconfigure.MessageSourceAutoConfiguration,\
org.springframework.boot.autoconfigure.\
PropertyPlaceholderAutoConfiguration,\
org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration,\
org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration,\
org.springframework.boot.autoconfigure.cassandra.\
CassandraAutoConfiguration,\
org.springframework.boot.autoconfigure.cloud.CloudAutoConfiguration,\
....
....

```

As you can see from Listing 3-7, the `spring.factories` defined all the auto-configuration classes that are used to set up any configuration that your application needs for running. Let's take a look at the `CloudAutoConfiguration` class (see Listing 3-8).

Listing 3-8. `org.springframework.boot.autoconfigure.cloud.CloudAutoConfiguration.java`

```

package org.springframework.boot.autoconfigure.cloud;

import org.springframework.boot.autoconfigure.AutoConfigureOrder;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.boot.autoconfigure.condition.ConditionalOnClass;
import org.springframework.boot.autoconfigure.condition.\
ConditionalOnMissingBean;
import org.springframework.boot.autoconfigure.condition.\
ConditionalOnProperty;
import org.springframework.cloud.Cloud;
import org.springframework.cloud.app.ApplicationInstanceInfo;
import org.springframework.cloud.config.java.CloudScan;
import org.springframework.cloud.config.java.CloudScanConfiguration;

```

```

import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;
import org.springframework.context.annotation.Profile;
import org.springframework.core.Ordered;

@Configuration
@Profile("cloud")
@AutoConfigureOrder(CloudAutoConfiguration.ORDER)
@ConditionalOnClass(CloudScanConfiguration.class)
@ConditionalOnMissingBean(Cloud.class)
@ConditionalOnProperty(prefix = "spring.cloud", name = "enabled",
havingValue = "true", matchIfMissing = true)
@Import(CloudScanConfiguration.class)
public class CloudAutoConfiguration {

    // Cloud configuration needs to happen early (before data, mongo etc.)
    public static final int ORDER = Ordered.HIGHEST_PRECEDENCE + 20;
}

```

Listing 3-8 shows you the `CloudAutoConfiguration` class. As you can see, it's very short class, but it configures a cloud application if it finds on the application classpath the `spring-cloud` classes, but how? It uses the `@ConditionalOnClass` and `@ConditionalOnMissingBean` annotations to decide if the application is a cloud app or not. Don't worry too much about this, because you are going to use these annotations when you create your own auto-configuration class in the chapter of the book: *Extending Spring Boot*.

Another things to see in Listing 3-8 is the use of the `@ConditionalOnProperty` annotation, that only applies if the property `spring.cloud` is enabled. It's worth mentioning that this auto-configuration is executed in a cloud profile, denoted by the `@Profile` annotation. The `@Import` annotation is applied only if the other annotations met their conditions (`@Conditional*` annotations are used), meaning that the import of the `CloudScanConfiguration` class is executed if in the classpath are the `spring-cloud-*` classes. I cover more in Chapter 13. For now you need to understand that the auto-configuration uses your classpath to figure out what to configure for your app. That's why we say that Spring Boot is an *opinionated runtime*, remember?

Spring Boot Features

In this section, I show you some of the Spring Boot features. Spring Boot is highly customizable, from the auto-configuration that sets up your application (based on the classpath) to customizing how it starts, what to show, and what to enable or disable based on its own properties. So let's get to know some Spring Boot features that customize your Spring app.

Let's create a Spring Boot Java project using the Spring Boot Initializr. Open your browser and go to <https://start.spring.io>. Add the following values to the fields. Make sure to click in the "Switch to the full version" field so that you can modify the package name.

- Group: `com.apress.spring`
- Artifact: `spring-boot-simple`
- Name: `spring-boot-simple`
- Package Name: `com.apress.spring`

You can select either a Maven or a Gradle project type. Then press the Generate Project button, which downloads a ZIP file. Uncompress it wherever you like and import it into your favorite IDE (see Figure 3-1).

Generate a Maven Project with Java and Spring Boot 2.0.0

Project Metadata

Artifact coordinates

Group
com.apress.spring

Artifact
spring-boot-simple

Name
spring-boot-simple

Description
Demo project for Spring Boot

Package Name
com.apress.spring

Packaging
Jar

Java Version
8

Too many options? [Switch back to the simple version.](#)

Dependencies
Add Spring Boot Starters and dependencies to your application

Search for dependencies
Web, Security, JPA, Actuator, Devtools...

Selected Dependencies

[Generate Project](#)

Figure 3-1. Spring Boot project

Note You can download the source code from the Apress website, and in every project you find the Maven pom.xml and the Gradle build.gradle files, so you can choose which building tool you want to use.

Now, run the Spring Boot app. Use your IDE or open a terminal and execute the following command if you are using Maven.

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

If you are using Gradle, you can execute

```
$ ./gradlew bootRun
```

```

    .
    /\ /  ___'  _ _ _ _ _  (\_)  _ _ _ _ _  \ \ \ \ \
  ( ( )\  ___ | ' _ | ' _ | | ' _ \ \ _ ' | \ \ \ \ \
  \ \ /  ___| |_) | | | | | | | | | | ( _ | | ) ) ) )
    ' | ___ | . _ | | | | | | | | | | | | / / / / /
  =====|_|=====|_|/=//_/_/_/_/_/_/
  :: Spring Boot ::           (v2.0.0.RELEASE)

```

```

INFO 10669 --- [    main] c.a.spring.SpringBootSimpleApplication :
Starting SpringBootSimpleApplication on ...
INFO 10669 --- [    main] c.a.spring.SpringBootSimpleApplication : No
active profile set, falling back to default profiles: default
INFO 10669 --- [    main] s.c.a.AnnotationConfigApplicationContext :
Refreshing org.springframework.context.annotation...
INFO 10669 --- [    main] o.s.j.e.a.AnnotationMBeanExporter      :
Registering beans for JMX exposure on startup
INFO 10669 --- [    main] c.a.spring.SpringBootSimpleApplication : Started
SpringBootSimpleApplication in 1.582 seconds (JVM running for 4.518)
INFO 10669 --- [Thread-3] s.c.a.AnnotationConfigApplicationContext :
Closing org.springframework.context.annotation...
INFO 10669 --- [Thread-3] o.s.j.e.a.AnnotationMBeanExporter      :
Unregistering JMX-exposed beans on shutdown

```

You should see something similar to this output. It shows a banner (Spring Boot) and some logs. Let's look at the main application in Listing 3-9.

Listing 3-9. `src/main/java/com/apress/spring/SpringBootSimpleApplication.java`

```

package com.apress.spring;

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

```

```

@SpringBootApplication
public class SpringBootSimpleApplication {

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

```

Listing 3-9 shows the main application. You already know about the Spring Boot components from the previous chapter, but let's review it again.

- `@SpringBootApplication`. This annotation is actually the `@ComponentScan`, `@Configuration`, and `@EnableAutoConfiguration` annotations. You already know everything about the `@EnableAutoConfiguration` from the previous sections.
- `SpringApplication`. This class provides the bootstrap for the Spring Boot application that is executed in the main method. You need to pass the class that is executed.

Now, you are ready to start customizing the Spring Boot app.

SpringApplication Class

You can have a more advance configuration using the `SpringApplication`, because you can create an instance out of it and do a lot more (see Listing 3-10).

Listing 3-10. `src/main/java/com/apress/spring/SpringBootSimpleApplication.java` Version 2

```

package com.apress.spring;

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

@SpringBootApplication
public class SpringBootSimpleApplication {

    public static void main(String[] args) {

```



```

    SpringApplication app =
        new SpringApplication(SpringBootSimpleApplication.class);
        //add more features here.
        app.run(args);
    }
}

```

SpringApplication allows you to configure the way your app behaves, and you have control over the main ApplicationContext where all your Spring beans are used. If you need to know more about ApplicationContext and how to use it, I recommend *Pro Spring Framework 5* (Apress, 2017) in which the authors explain everything about Spring. In this case, we focus on some of the Spring Boot features. Let's start with something cool.

Custom Banner

Every time you run your application, you see a banner displayed at the beginning of the application. It can be customized in different ways.

Implement the `org.springframework.boot.Banner` interface (see Listing 3-11).

Listing 3-11. `src/main/java/com/apress/spring/SpringBootSimpleApplication.java` Version 3

```

package com.apress.spring;

import java.io.PrintStream;

import org.springframework.boot.Banner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.core.env.Environment;

@SpringBootApplication
public class SpringBootSimpleApplication {

    public static void main(String[] args) {

```

```

SpringApplication app = new SpringApplication(SpringBootSimpleApplica
tion.class);
app.setBanner(new Banner() {
    @Override
    public void printBanner(Environment environment, Class<?>
sourceClass, PrintStream out) {
        out.print("\n\n\tThis is my own banner!\n\n".toUpperCase());
    }
});
    app.run(args);
}
}

```

When you run the application, you see something like this:

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

```
    THIS IS MY OWN BANNER!
```

```

INFO[main] c.a.spring.SpringBootSimpleApplication : Starting
SpringBootSimpleApplication ...
...
...
INFO[main] c.a.spring.SpringBootSimpleApplication : Started
SpringBootSimpleApplication in 0.789seconds (JVM running for 4.295)
INFO[Th-1] s.c.a.AnnotationConfigApplicationContext : Closing org.
springframework.context.annotation.AnnotationConfigApplicationContext@203f
6b5: startup date [Thu Feb 25 19:00:34 MST 2016]; root of context hierarchy
INFO[Th-1] o.s.j.e.a.AnnotationMBeanExporter : Unregistering JMX-
exposed beans on shutdown

```

You can create your own ASCII banner and display it. How? There is a very cool site that creates ASCII art from text (<http://patorjk.com>), as shown in Figure 3-2.

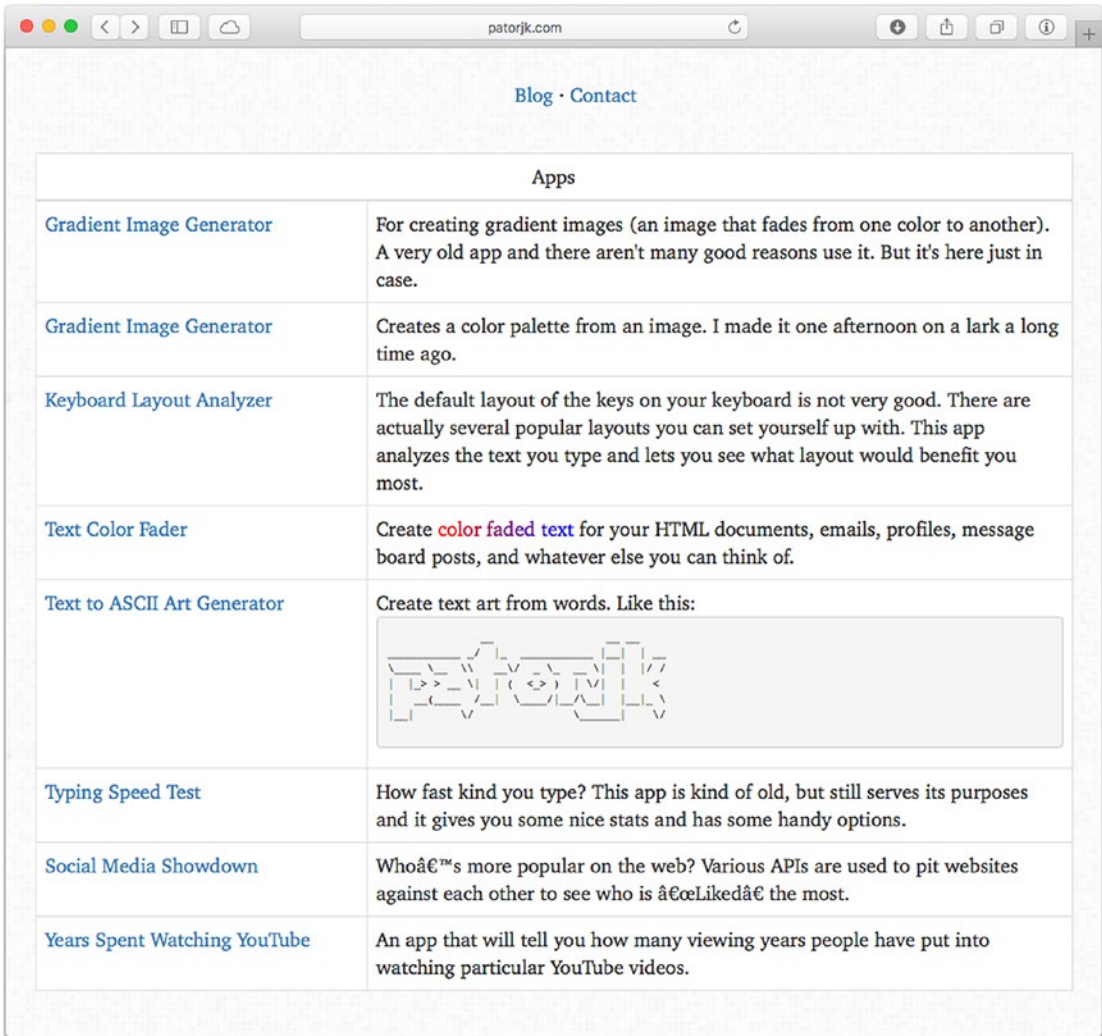


Figure 3-2. Text to ASCII art generator

Figure 3-2 shows the website at <http://patorjk.com>. Click the *Text to ASCII Art Generator* link. Then, add **Pro Spring Boot 2.0** (or whatever you want) in the text field. Then, click *Test All* to see all the ASCII art (see Figure 3-3).

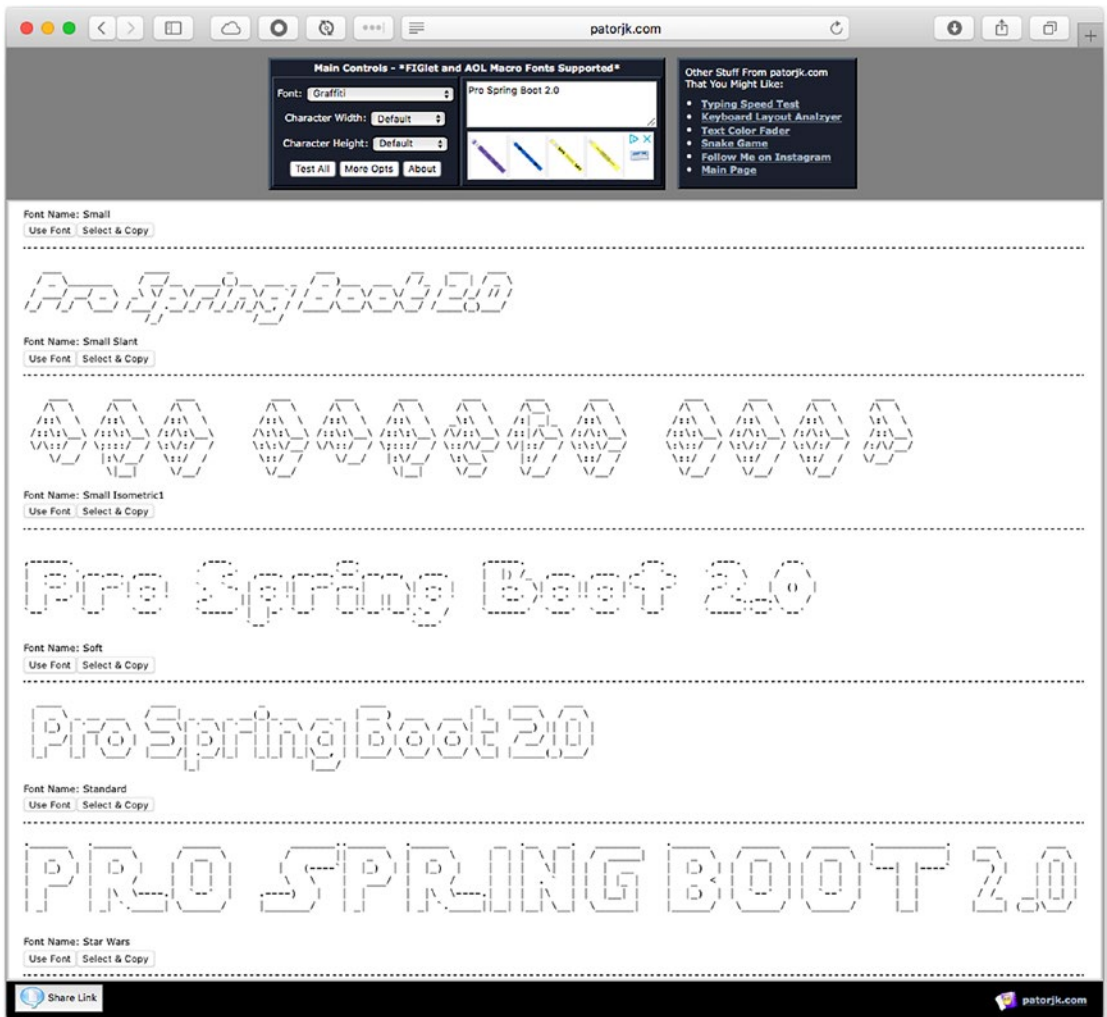


Figure 3-3. ASCII art

Figure 3-3 shows all the ASCII art (about 314). Now, you can select one. Click the Select Text button, copy (Ctrl+C Windows/Cmd+C macOS) it, and create a file named `banner.txt` in the `src/main/resources/` directory (see Figure 3-4).



Figure 3-4. `src/main/resource/banner.txt` content

You can run your application again.

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

You see the ASCII art that you added in the `banner.txt` file. If you run your app using Listing 3-11 (where you are setting the banner), it overrides it and uses the `banner.txt` file that is in your classpath; that's the default.

By default, Spring Boot looks for the `banner.txt` file in the classpath. But you can change its location. Create another `banner.txt` file (or copy the one you have already) in the `src/main/resources/META-INF/` directory. Run the application by passing a `-D` parameter. Execute the following command if you are using Maven.

```
$ ./mvnw spring-boot:run -Dspring.banner.location=classpath:/META-INF/
banner.txt
```

If you are using Gradle, you need first to add this configuration at the end of the `build.gradle` file.

```
bootRun {
    systemProperties = System.properties
}
```

Execute the following.

```
$ ./gradlew bootRun -Dspring.banner.location=classpath:/META-INF/banner.txt
```

This command is using the flag `-D` to pass the `spring.banner.location` property that is pointing to the new classpath location, `/META-INF/banner.txt`. You can declare this property in the `src/main/resources/application.properties` file, as follows.

```
spring.banner.location=classpath:/META-INF/banner.txt
```

Run it like this if using Maven:

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

Run it like this if you are using Gradle:

```
$ ./gradlew bootRun
```

You have several options for setting up the `banner.txt` file.

You can completely remove the banner. You can define it in the `src/main/resources/application.properties` like this:

```
spring.main.banner-mode=off
```

This command has precedence over the default `banner.txt` file located at the `classpath:banner.txt` location. Also, you can do it programmatically (just remember to comment out the property) (see Listing 3-12).

Listing 3-12. `src/main/java/com/apress/spring/SpringBootApplication.java` Version 4

```
package com.apress.spring;

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

@SpringBootApplication
public class SpringBootApplication {

    public static void main(String[] args) {
        SpringApplication app = new SpringApplication(SpringBootApplication.class);
        app.setBannerMode(Banner.Mode.OFF);
        app.run(args);
    }
}
```

SpringApplicationBuilder

The `SpringApplicationBuilder` class provides a fluent API and is a builder for the `SpringApplication` and the `ApplicationContext` instances. It also provides hierarchy support and everything that I showed you so far (with the `SpringApplication`) can be set with this builder (see Listing 3-13).

Listing 3-13. src/main/java/com/apress/spring/SpringBootSimpleApplication.
java Version 5

```
package com.apress.spring;

import org.springframework.boot.Banner;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.builder.SpringApplicationBuilder;

@SpringBootApplication
public class SpringBootSimpleApplication {

    public static void main(String[] args) {
        new SpringApplicationBuilder()
            .bannerMode(Banner.Mode.OFF)
            .sources(SpringBootSimpleApplication.class)
            .run(args);
    }
}
```

Listing 3-13 shows the `SpringApplicationBuilder` fluent API. Next, let's look at more examples.

You can have a hierarchy when creating a Spring app. (If you want to know more about application context in Spring, I recommend *Pro Spring* 5th Edition.) You can create it with `SpringApplicationBuilder`.

```
new SpringApplicationBuilder(SpringBootSimpleApplication.class)
    .child(OtherConfig.class)
    .run(args);
```

If you have a web configuration, make sure that it's being declared as a child. Also parent and children must share the same `org.springframework.core.env.Environment` interface (this represents the environment in which the current application is running; it is related to profiles and properties declarations).

You can log the information at startup; by default it is set to true.

```
new SpringApplicationBuilder(SpringBootSimpleApplication.class)
    .logStartupInfo(false)
    .run(args);
```

You can activate profiles.

```
new SpringApplicationBuilder(SpringBootSimpleApplication.class)
    .profiles("prod", "cloud")
    .run(args);
```

Later, I'll show you profiles, so that you can make sense of the preceding line.

You can attach listeners for some of the `ApplicationEvent` events.

```
Logger log = LoggerFactory.getLogger(SpringBootSimpleApplication.class);
new SpringApplicationBuilder(SpringBootSimpleApplication.class)
    .listeners(new ApplicationListener<ApplicationEvent>() {
        @Override
        public void onApplicationEvent(ApplicationEvent event) {
            log.info("#### > " + event.getClass().getCanonicalName());
        }
    })
    .run(args);
```

When you run your application, you should see the following output.

```
...
#### > org.springframework.boot.context.event.ApplicationPreparedEvent
...
#### > org.springframework.context.event.ContextRefreshedEvent
#### > org.springframework.boot.context.event.ApplicationReadyEvent
...
#### > org.springframework.context.event.ContextClosedEvent
...
```

Your application can add the necessary logic to handle those events. You can also have these additional events: `ApplicationStartedEvent` (this is sent at the start), `ApplicationEnvironmentPreparedEvent` (this is sent when the environment is known), `ApplicationPreparedEvent` (this is sent after the bean definitions), `ApplicationReadyEvent` (this is sent when the application is ready), and `ApplicationFailedEvent` (this is sent in case of exception during the startup). I showed you the other one in the output (more related to the Spring container).

You can remove any web environment auto-configuration from happening. Remember that Spring Boot guesses which kind of app you are running based on the classpath. For a web app, the algorithm is very simple; but imagine that you are using libraries that actually run without a web environment, and your app is not a web app, but Spring Boot configures it as such.

```
new SpringApplicationBuilder(SpringBootTestApplication.class)
    .web(WebApplicationType.NONE)
    .run(args);
```

You find that the `WebApplicationType` is an enum. Your app can be configured as `WebApplicationType.NONE`, `WebApplicationType.SERVLET`, and `WebApplicationType.REACTIVE`. As you can see, its meaning is very straightforward.

Application Arguments

Spring Boot allows you to get the arguments passed to the application. When you have `SpringApplication.run(SpringBootTestApplication.class, args)`, you have access to the args in your beans (see Listing 3-14).

Listing 3-14. `src/main/java/com/apress/spring/SpringBootTestApplication.java` -version 6

```
package com.apress.spring;

import java.io.IOException;
import java.util.List;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.ApplicationArguments;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.stereotype.Component;

@SpringBootApplication
public class SpringBootTestApplication {
```

```

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

@Component
class MyComponent {

    private static final Logger log = LoggerFactory.getLogger(MyComponent.
class);

    @Autowired
    public MyComponent(ApplicationArguments args) {
        boolean enable = args.containsOption("enable");
        if(enable)
            log.info("## > You are enabled!");

        List<String> _args = args.getNonOptionArgs();
        log.info("## > extra args ...");
        if(!_args.isEmpty())
            _args.forEach(file -> log.info(file));
    }
}

```

When you execute `containsOption`, it expects the argument to be `--<arg>`. In Listing 3-14, it is expecting the `--enable` argument; the `getNonOptionArgs` takes other arguments. To test it, you can execute the following command.

```
$ ./mvnw spring-boot:run -Dspring-boot.run.arguments="--enable"
```

You should see `## > You are enabled`.

Also, you can run it like this:

```
$ ./mvnw spring-boot:run -Dspring-boot.run.arguments="arg1,arg2"
```

If you are using Gradle (and at the time of this writing), you need to wait a little bit because passing parameters is still an issue (see <https://github.com/spring-projects/spring-boot/issues/1176>); but you can pass a parameter in an executable jar, which I describe in the next section.

Accessing Arguments with an Executable JAR

You have the option to create a stand-alone app—an executable JAR (you will learn more about this). To create an executable JAR, execute the following command if you are using Maven.

```
$ ./mvnw package
```

This command creates the executable JAR in the target directory.

Or if you are using Gradle, you can execute

```
$./gradlew build
```

This command creates an executable JAR in the build/libs directory.

Now you can run the executable JAR.

```
$ java -jar spring-boot-simple-0.0.1-SNAPSHOT.jar
```

You can pass arguments like this:

```
$ java -jar spring-boot-simple-0.0.1-SNAPSHOT.jar --enable arg1 arg2
```

You should get the same text for the enable arg and a list of arg1 and arg2.

ApplicationRunner and CommandLineRunner

If you notice Spring Boot after executing the `SpringApplication`, it ends. If you are curious, you cannot use beans after this class is executed, but there is a solution. Spring Boot has the `ApplicationRunner` and `CommandLineRunner` interfaces that exposes a `run` method (see Listing 3-15).

Listing 3-15. `src/main/java/com/apress/spring/SpringBootSimpleApplication.java` Version 7

```
package com.apress.spring;

import java.io.IOException;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.ApplicationArguments;
```

```

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

@SpringBootApplication
public class SpringBootSimpleApplication implements CommandLineRunner,
ApplicationRunner{
    private static final Logger log = LoggerFactory.getLogger(SpringBoot
SimpleApplication.class);

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

@Bean
    String info(){
        return "Just a simple String bean";
    }

    @Autowired
    String info;

    @Override
    public void run(ApplicationArguments args) throws Exception {
        log.info("## > ApplicationRunner Implementation...");
        log.info("Accessing the Info bean: " + info);
        args.getNonOptionArgs().forEach(file -> log.info(file));
    }

    @Override
    public void run(String... args) throws Exception {
        log.info("## > CommandLineRunner Implementation...");
        log.info("Accessing the Info bean: " + info);
        for(String arg:args)
            log.info(arg);
    }
}

```

Listing 3-15 shows both `CommandLineRunner` and `ApplicationRunner` and their implementations. `CommandLineRunner` exposes the public `void run(String... args)` method and `ApplicationRunner` exposes the public `void run(ApplicationArguments args)` method. They are practically the same. It is not necessary to implement both at the same time. Listing 3-16 shows another way to use the `CommandLineRunner` interface.

Listing 3-16. `src/main/java/com/apress/spring/SpringBootSimpleApplication.java` Version 8

```
package com.apress.spring;

import java.io.IOException;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
public class SpringBootSimpleApplication {
    private static final Logger log = LoggerFactory.getLogger(SpringBoot
        SimpleApplication.class);

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

    @Bean
    String info(){
        return "Just a simple String bean";
    }

    @Autowired
    String info;
```

```

@Bean
CommandLineRunner myMethod(){
    return args -> {
        log.info("## > CommandLineRunner Implementation...");
        log.info("Accessing the Info bean: " + info);
        for(String arg:args)
            log.info(arg);
    };
}
}

```

Listing 3-16 shows a method annotated with the `@Bean` annotation returning a `CommandLineRunner` implementation. This example uses the Java 8 syntax (lambda) to do the return. You can add as many methods that return `CommandLineRunner` as you want. If you wanted to execute in a certain order, you can use the `@Order` annotation.

Application Configuration

We developers know that we are never going to get rid of application configuration. We always look for where to persist for example URLs, IPs, credentials, database information, and so forth—data that we normally use quite often in our applications. We know that as a best practice, we need to avoid hard-coding this kind of configuration information. We need to externalize so that it can be secure, easy to use, and deploy.

With Spring, you have the option to use XML and the `<context:property-placeholder/>` tag, or you can use the `@PropertySource` annotation to declare your properties pointing to a file that has declared them. Spring Boot offers you the same mechanism but with more improvements.

- Spring Boot gives you different options for saving your application configuration.
 - You can use a file named `application.properties` that should be located in the root classpath of your application. (There are more options where you can add this file, which I show you later.)
 - You can use a YAML notation file named `application.yml` that also needs to be located in the root classpath. (There are more options where you can add this file, which I show you later.)

- You can use environment variables. This is becoming the default practice for cloud scenarios.
- And you can use command-line arguments.

Remember that Spring Boot is an opinionated technology. Most of its application configuration is based on a common `application.properties` or `application.yml` file. If none are specified, it already has those properties values as defaults. You can get the complete list of the common application properties at <https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html>.

One of the best features from Spring (and Spring Boot) is access to the properties values by using the `@Value` annotation (with the name of the property). Or they can be accessed from the `org.springframework.core.env.Environment` interface that extends from the `org.springframework.core.env.PropertyResolver` interface. For example if you have a `src/main/resources/application.properties` file with

```
data.server=remoteserver:3030
```

You can access the `data.server` property in your application by using the `@Value` annotation, like in the following snippet.

```
//...
@Service
public class MyService {

    @Value("${data.server}")
    private String server;

    //...
}
```

This code snippet shows the usage of the `@Value` annotation. Spring Boot injects the `data.server` property value from the `application.properties` file in the `server` variable with its value `remoteserver:3030`.

If you don't want to use the `application.properties`, you have the option to inject the properties via a command line.

```
$ java -jar target/myapp.jar --data.server=remoteserver:3030
```

And you have the same result. If you don't like the `application.properties` file or you hate the YAML syntax, you can use a specialized environment variable named `SPRING_APPLICATION_JSON` to expose the same properties and its values.

```
$ SPRING_APPLICATION_JSON='{ "data":{"server":"remoteserver:3030"}}' java
-jar target/myapp.jar
```

And again, you have the same result. (Windows users should use the SET instruction to set the environment variable first.) So, Spring Boot gives you options to expose application properties.

Configuration Properties Examples

Let's create a simple project that helps you better understand the application configuration. Open your browser and go to <https://start.spring.io>. Use the following field values.

- Group: `com.apress.spring`
- ArtifactId: `spring-boot-config`
- Package Name: `com.apress.spring`
- Name: `spring-boot-config`

You can choose whatever project type you feel comfortable. Click the Generate Project button, save the ZIP file, uncompress it and import it in your favorite IDE.

Before continuing with the project, you must know that Spring Boot uses an order if you want to override your application configuration properties:

- Command-line arguments
- `SPRING_APPLICATION_JSON`
- JNDI (`java:comp/env`)
- `System.getProperties()`
- OS Environment variables
- `RandomValuePropertySource (random.*)`
- Profile-specific (`application-{profile}.jar`) outside of the package jar.
- Profile-specific (`application-{profile}.jar`) inside of the package jar.

- Application properties (application.properties) outside of the package jar.
- Application properties (application.properties) inside of the package jar.
- @PropertySource
- SpringApplication.setDefaultProperties

as you can see, that's the order for overriding the application properties. Let's start with some examples.

Command-Line Arguments

Go to your project and edit the main class to look like Listing 3-17.

Listing 3-17. src/main/java/com/apress/spring/SpringBootConfigApplication.java

```
package com.apress.spring;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
public class SpringBootConfigApplication {

    private static Logger log = LoggerFactory.getLogger(SpringBootConfig
        Application.class);

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

    @Value("${server.ip}")
    String serverIp;
```

```

@Bean
CommandLineRunner values(){
    return args -> {
        log.info(" > The Server IP is: " + serverIp);
    };
}
}

```

Listing 3-17 shows the main class. As you can see, it is using the `@Value("${server.ip}")` annotation. This annotation translates the text `"${server.ip}"`. It looks for this property and its value in the order that I mentioned before.

You can run this example by executing the following command in the root of your project.

```
$ ./mvnw spring-boot:run -Dserver.ip=192.168.12.1
```

If you package your app first (to create an executable JAR), then run it as follows.

```
$ ./mvnw package
$ java -jar target/spring-boot-config-0.0.1-SNAPSHOT.jar --server.ip=192.168.12.1
```

In either case, you see something similar to the following output.

```

.
/\ / ___' _ _ _ _ _ ( _ ) _ _ _ _ _ \ \ \ \ \
( ( ) \__ | ' _ | ' _ | | ' _ \ \_ | \ \ \ \ \
\ \ / __ ) | | ) | | | | | | | ( _ | | ) ) ) )
' | ___ | . _ | | | | | | | | \ \_ , | / / / /
=====|_|=====|__/=/_/_/_/
:: Spring Boot ::          (v2.0.0.RELEASE)

INFO - [main] c.a.spring.SpringBootConfigApplication : Starting
SpringBootConfigApplication v0.0..
INFO - [main] c.a.spring.SpringBootConfigApplication : No active profile
set, falling back to de..
INFO - [main] s.c.a.AnnotationConfigApplicationContext : Refreshing org.
springframework.context.an..

```

```

INFO - [main] o.s.j.e.a.AnnotationMBeanExporter      : Registering beans
for JMX exposure on sta..
INFO - [main] c.a.spring.SpringBootConfigApplication : Started
SpringBootConfigApplication in 0...
INFO - [main] c.a.spring.SpringBootConfigApplication : > The Server IP
is: 192.168.34.56
INFO - [main] c.a.spring.SpringBootConfigApplication : > App Name: My
Config App
INFO - [main] c.a.spring.SpringBootConfigApplication : > App Info: This
is an example
INFO - [ T-2] s.c.a.AnnotationConfigApplicationContext : Closing org.
springframework.context.annot..
INFO - [ T-2] o.s.j.e.a.AnnotationMBeanExporter      : Unregistering JMX-
exposed beans on shutdo...

```

You see this from the output: > The Server IP is: 1921.68.12.1.

Now, let's create the `application.properties` file (see Listing 3-18 for its contents).

Listing 3-18. `src/main/resources/application.properties`

```
server.ip=192.168.23.4
```

If you run the application with the same command-line arguments, you see that the arguments have precedence from the `application.properties` file, but if you run it without the arguments, as follows.

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

or

```
$ ./mvnw package
```

```
$ java -jar target/spring-boot-config-0.0.1-SNAPSHOT.jar
```

You get the text `The Server IP is: 192.168.3.4`. If you do a lot of JSON formatting, perhaps you are interested in passing your properties in this format. You can use the `spring.application.json` property. So, you can run it like this:

```
$ ./mvnw spring-boot:run -Dspring.application.json='{"server":{"ip":"192.168.145.78"}}'
```

or

```
$ java -jar target/spring-boot-config-0.0.1-SNAPSHOT.jar --spring.
application.json='{ "server":{ "ip": "192.168.145.78" } }'
```

You can also add it as environment variable.

```
$ SPRING_APPLICATION_JSON='{ "server":{ "ip": "192.168.145.78" } }' java -jar
target/spring-boot-config-0.0.1-SNAPSHOT.jar
```

You see the text > The Server IP is: 192.168.145.78. And, yes, you can add your environment variable that refers to your property like this:

```
$ SERVER_IP=192.168.150.46 ./mvnw spring-boot:run
```

or

```
$ SERVER_IP=192.168.150.46 java -jar target/spring-boot-config-0.0.1-
SNAPSHOT.jar
```

You see the text > The Server IP is: 192.168.150.46.

Note Remember for Windows users, you need to use the SET instruction for using environment variables.

How does Spring Boot know that the environment variable is related to the `server.ip` property?

Relaxed Binding

Spring Boot use relaxed rules for binding (see Table 3-1).

Table 3-1. *Spring Boot Relaxed Binding*

Property	Description
<code>message.destinationName</code>	Standard camel case
<code>message.destination-name</code>	Dashed notation, which is the recommended way to add in the <code>application.properties</code> or YML file.
<code>MESSAGE_DESTINATION_NAME</code>	Uppercase, which is the recommended way to use with OS environment variables

Table 3-1 shows the relaxed rules that apply to the property names. That's why in the previous example, the `server.ip` property is recognized also as `SERVER_IP`. This relaxed rule has to do with the `@ConfigurationProperties` annotation and its prefix, which you see in a later section.

Changing Location and Name

Spring Boot has an order for finding the `application.properties` or YAML file.

1. It looks at the `/config` subdirectory located in the current directory.
2. The current directory
3. The classpath `/config` package
4. The classpath root

You can test by creating a `/config` subdirectory in your current directory and add a new `application.properties`, and test that the order is true. Remember that you should already have an `application.properties` file in the classpath root (`src/main/resources`).

Spring Boot allows you to change the name of the properties file and its location. So for example, imagine that you use the `/config` subdirectory, and the name of the properties file is now `mycfg.properties` (its content: `server.ip=127.0.0.1`). Then you can run the app with the following command.

```
$/mvnw spring-boot:run -Dspring.config.name=mycfg
```

or

```
$ java -jar target/spring-boot-config-0.0.1-SNAPSHOT.jar --spring.config.name=mycfg
```

or

```
$ SPRING_CONFIG_NAME=mycfg java -jar target/spring-boot-config-0.0.1-SNAPSHOT.jar
```

You should see the text: > The Server IP is: 127.0.0.1. It is not necessary to include `properties` to the name; it automatically uses it. You can also change its location. For example, create a subdirectory named `app` and add a `mycfg.properties` file (its content: `server.ip=localhost`). Then you can run or execute your app with

```
$ ./mvnw spring-boot:run -Dspring.config.name=mycfg -Dspring.config.location=file:app/
```

or

```
$ java -jar target/spring-boot-config-0.0.1-SNAPSHOT.jar --spring.config.location=file:app/ --spring.config.name=mycfg
```

Or you can add the `mycfg.properties` in the `src/main/resources/META-INF/conf` (you can create it) and execute this:

```
$ mkdir -p src/main/resources/META-INF/conf
$ cp config/mycfg.properties src/main/resources/META-INF/conf/
$ ./mvnw clean spring-boot:run -Dspring.config.name=mycfg -Dspring.config.location=classpath:META-INF/conf/
```

You should see the text > The Server IP is: 127.0.0.1. Try to change that value of the property so you can see that it is actually looking in the classpath.

Spring Boot also has an order for searching for the properties file.

1. classpath
2. classpath:/config
3. file:
4. file:config/

Unless you change it with the `spring.config.location` property, which environment variable is to set to change the location of the properties file? It is `SPRING_CONFIG_LOCATION`.

Profile Based

Since version 3.1, if I'm not mistaken, the Spring Framework added a cool feature that allows the developer to create custom properties and beans based on profiles. This is a useful way to separate environments without having to recompile or package a Spring app. The only thing to do is specify the active profile with the `@ActiveProfiles`

annotation, or getting the current Environment and use the `setActiveProfiles` method. Or you can use the environment variable `SPRING_PROFILES_ACTIVE`, or the `spring.profiles.active` property.

You can use the properties file using this format: `application-{profile}.properties`. Lets' create two files in your `config/` subdirectory: `application-qa.properties` and `application-prod.properties`. Let's look at the contents of each one.

- `application-qa.properties`
`server.ip=localhost`
- `application-prod.properties`
`server.ip=http://my-remote.server.com`

Now you can run your example with

```
$ ./mvnw clean spring-boot:run -Dspring.profiles.active=prod
```

When you execute this command, take a look at the beginning of the logs. You should see something similar to the following output.

```
...
INFO 58270 --- [main] c.a.spring... : The following profiles are active: prod
...
INFO 58270 --- [main] c.a.spring... : > The Server IP is: http://my-remote.server.com
INFO 58270 --- [main] c.a.spring... : > App Name: Super App
INFO 58270 --- [main] c.a.spring... : > App Info: This is production
```

You should see the legend, The following profiles are active: `prod`, and of course, the profile `application properties active (application-prod.properties)` value: `> The Server IP is: http://my-remote.server.com`. As an exercise, try to change the name of the `application-prod.properties` to `mycfg-prod.properties` and the `application-qa.properties` to `mycfg-qa.properties`. Use the Spring properties that get the new name. If you don't set any active profiles, it uses the default, meaning that it grabs `application.properties`.

Custom Properties Prefix

Spring Boot allows you to write and use your own custom property prefix for your properties. The only thing you need to do is annotate with the `@ConfigurationProperties` annotation, which is a Java class that has setters and getters as its properties.

If you are using the STS IDE, I recommend including a dependency in your `pom.xml` or your `build.gradle` file (depending of which dependency manager you are using). This dependency creates a code-insight, and it triggers the editor's code completion for the properties. So if you are using Maven, you can add the next dependency in your `pom.xml` file.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-configuration-processor</artifactId>
    <optional>true</optional>
</dependency>
```

If you are using Gradle, you can add the following dependency to your `build.gradle` file.

```
dependencies {
    optional "org.springframework.boot:spring-boot-configuration-processor"
}

compileJava.dependsOn(processResources)
```

This dependency allows you to process your custom properties and have code completion. Now, let's look at the example. Modify your `src/main/resources/application.properties` file to look like Listing 3-19.

Listing 3-19. `src/main/resources/application.properties`

```
server.ip=192.168.3.5

myapp.server-ip=192.168.34.56
myapp.name=My Config App
myapp.description=This is an example
```


Listing 3-19 shows the `application.properties` file. The second block is new. It defines the custom properties with `myapp` as the prefix. Open your main app class and edit it to look like Listing 3-20.

Listing 3-20. `src/main/java/com/apress/spring/SpringBootConfigApplication.java` Version 9

```
package com.apress.spring;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.context.annotation.Bean;
import org.springframework.stereotype.Component;

@SpringBootApplication
public class SpringBootConfigApplication {

    private static Logger log = LoggerFactory.getLogger(SpringBootConfig
Application.class);

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

    @Value("${myapp.server-ip}")
    String serverIp;

    @Autowired
    MyAppProperties props;
```

```

@Bean
CommandLineRunner values(){
    return args -> {
        log.info(" > The Server IP is: " + serverIp);
        log.info(" > App Name: " + props.getName());
        log.info(" > App Info: " + props.getDescription());
    };
}

@Component
@ConfigurationProperties(prefix="myapp")
public static class MyAppProperties {
    private String name;
    private String description;
    private String serverIp;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    public String getServerIp() {
        return serverIp;
    }

    public void setServerIp(String serverIp) {
        this.serverIp = serverIp;
    }
}
}

```

Listing 3-19 shows the main app class. Let's examine it.

- `@Value("${myapp.server-ip}")`. Now the annotation has a `myapp.server-ip`, which means that the value is equal to `192.168.34.56`.
- `@Autowired MyAppProperties props`. This creates an instance of the `MyAppProperties` type.
- `@Component @ConfigurationProperties(prefix="myapp")`. The `@ConfigurationProperties` annotation is telling Spring Boot that the class is used for all the properties defined in the application. properties file that has the prefix `myapp`. This means that it recognizes when you have `myapp.serverIp` (or `myapp.server-ip`), `myapp.name`, and `myapp.description`. The `@Component` annotation only makes sure that the class is picked up as a bean.

Spring Boot uses relaxed rules binding Environment properties to `@ConfigurationProperties` beans, so you don't have any collision names.

Now if you run your app, you should see all your `myapp` properties.

```
$ ./mvnw clean spring-boot:run
...
> The Server IP is: 192.168.34.56
> App Name: My Config App
> App Info: This is an example
...
```

As you can see, you have plenty of options for using your application configuration properties, and if you want to use the YAML syntax, please refer to the Spring Boot documentation for examples.

You can add hits to your IDE by providing metadata information that helps with property completion. Take a look at the reference documentation at <https://docs.spring.io/spring-boot/docs/current/reference/html/configuration-metadata.html> on how to create this metadata.

Summary

This chapter gave you a tour of Spring Boot insights by explaining the auto-configuration feature, including how the `@EnableAutoConfiguration` annotation works behind the scenes. You learned how to exclude some of the auto-configuration classes as well.

You learned about some of the Spring Boot features and how to use the application configuration properties. You also learned how to customize your application configuration properties by adding a prefix.

In the next chapter, you learn more about web applications with Spring Boot.

CHAPTER 4

Web Applications with Spring Boot

Nowadays, the web is the main channel for any type of application—from desktop to mobile devices, from social and business applications to games, and from simple content to streaming data. With this in mind, Spring Boot can help you easily develop the next generation of web applications.

This chapter shows you how to create Spring Boot web applications with ease. You have already learned, with some examples in earlier chapters, what you can do with the web. You learned that Spring Boot makes it easier to create web apps with a few lines of code and that you don't need to worry about configuration files or look for an application server to deploy your web application. By using Spring Boot and its auto-configuration, you can have an embedded application server, such as Tomcat, Netty, Undertow, or Jetty, which makes your app very distributable and portable.

Spring MVC

Let's start talking about the Spring MVC technology and some of its features. Remember that the Spring Framework consists of about 20 modules or technologies, and the web technology is one of them. For the web technology, the Spring Framework has the `spring-web`, `spring-webmvc`, `spring-webflux`, and `spring-websocket` modules.

The `spring-web` module has basic web integration features, such as multipart file upload functionality, initialization of the Spring container (by using servlet listeners), and a web-oriented application context. The `spring-mvc` module (a.k.a., the web server module) contains all the Spring MVC (Model-View-Controller) and REST services implementations for web applications. These modules provide many features, such as very powerful JSP tag libraries, customizable binding and validation, flexible model transfer, customizable handler and view resolution, and so on.

The Spring MVC is designed around the `org.springframework.web.servlet.DispatcherServlet` class. This servlet is very flexible and has a very robust functionality that you won't find in any other MVC web framework out there. With the `DispatcherServlet`, you have several out-of-the-box resolutions strategies, including view resolvers, locale resolvers, theme resolvers, and exception handlers. In other words, the `DispatcherServlet` take a HTTP request and redirect it to the right handler (the class marked with the `@Controller` or `@RestController` and the methods that use the `@RequestMapping` annotations) and the right view (your JSPs).

Spring Boot MVC Auto-Configuration

Web applications can be created easily by adding the `spring-boot-starter-web` dependency to your `pom.xml` or `build.gradle` file. This dependency provides all the necessary `spring-web` jars and some extra ones, such as `tomcat-embed*` and `jackson` (for JSON and XML). This means that Spring Boot uses the power of the Spring MVC modules and provides all the necessary *auto-configuration* for creating the right web infrastructure, such as configuring the `DispatcherServlet`, providing defaults (unless you override it), setting up an embedded Tomcat server (so you can run your application without any application containers), and more.

Auto-configuration adds the following features to your web application.

- *Static content support.* This means that you can add static content, such as HTML, JavaScript, CSS, media, and so forth, in a directory named `/static` (by default) or `/public`, `/resources`, or `/META-INF/resources`, which should be in your classpath or in your current directory. Spring Boot picks it up and serves them upon request. You can change this easily by modifying the `spring.mvc.static-path-pattern` or the `spring.resources.static-locations` properties. One of the cool features with Spring Boot and web applications is that if you create an `index.html` file, Spring Boot serves it automatically without registering any other bean or the need for extra configuration.
- *HttpMessageConverters.* If you are using a regular Spring MVC application and you want to get a JSON response, you need to create the necessary configuration (XML or JavaConfig) for the

`HttpMessageConverters` bean. Spring Boot adds this support by default so you don't have to; this means that you get the JSON format by default (due to the Jackson libraries that the `spring-boot-starter-web` provides as dependencies). And if Spring Boot auto-configuration finds that you have the Jackson XML extension in your classpath, it aggregates an XML `HttpMessageConverter` to the converters, meaning that your application can server based on your `content-type` request, either `application/json` or `application/xml`.

- *JSON serializers and deserializers.* If you want to have more control over the serialization/deserialization to/from JSON, Spring Boot provides an easy way to create your own by extending from `JsonSerializer<T>` and/or `JsonDeserializer<T>`, and annotating your class with the `@JsonComponent` so that it can be registered for usage. Another feature of Spring Boot is the Jackson support; by default, Spring Boot serializes the date fields as `2018-05-01T23:31:38.141+0000`, but you can change this default behavior by changing the `spring.jackson.date-format=yyyy-MM-dd` property (you can apply any date format pattern); the previous value generates the output, such as `2018-05-01`.
- *Path matching and content negotiation.* One of the Spring MVC application practices is the ability to respond to any suffix to represent the *content-type* response and its content negotiation. If you have something like `/api/todo.json` or `/api/todo.pdf`, the `content-type` is set to `application/json` and `application/pdf`; so the response is JSON format or a PDF file, respectively. In other words, Spring MVC performs `.*` suffix pattern matching, such as `/api/todo.*`. Spring Boot disables this by default. You can still use a feature where you can add a parameter, by using the `spring.mvc.contentnegotiation.favor-parameter=true` property (`false` by default); so you can do something like `/api/todo?format=xml`. (`format` is the default parameter name; of course, you can change it with `spring.mvc.contentnegotiation.parameter-name=myparam`). This triggers the `content-type` to `application/xml`.

- *Error handling.* Spring Boot uses `/error` mapping to create a white labeled page to show all the global errors. You can change the behavior by creating your own custom pages. You need to create your custom HTML page in the `src/main/resources/public/error/` location, so you can create `500.html` or `404.html` pages for example. If you are creating a RESTful application, Spring Boot responds as JSON format. Spring Boot also supports Spring MVC to handle errors when you are using `@ControllerAdvice` or `@ExceptionHandler` annotations. You can register custom `ErrorPages` by implementing `ErrorPageRegistrar` and declaring it as a Spring bean.
- *Template engine support.* Spring Boot supports FreeMarker, Groovy Templates, Thymeleaf, and Mustache. When you include the `spring-boot-starter-<template engine>` dependency, Spring Boot auto-configure is necessary to enable and add all the necessary view resolvers and file handlers. By default, Spring Boot looks at the `src/main/resources/templates/` path.

And there are many other features that Spring Boot Web auto-configure provides. Right now, we are only looking at the *Servlet technology*, but very soon we get into the newest addition to the Spring Boot family: *WebFlux*.

Spring Boot Web: ToDo App

To better understand how Spring Boot works with web applications and the power of the Spring MVC modules, you are going to create a `ToDo` app that exposes a RESTful API. These are the requirements:

- Create a `ToDo` domain model that has the following fields and types: `id` (String), `description` (String), `completed` (Boolean), `created` (date with time), `modified` (date with time).
- Create a RESTful API that provides the basic CRUD (create, read, update, delete) actions. Use the most common HTTP methods: `POST`, `PUT`, `PATCH`, `GET`, and `DELETE`.

- Create a repository that handles the state of multiple `ToDo`'s. For now, an in-memory repository is enough.
- Add an error handler when there is a bad request or when submitting a new `ToDo` doesn't have the required fields. The only mandatory field is the description.
- All the requests and responses should be in JSON format.

ToDo App

Open your browser and go to <https://start.spring.io> to create your `ToDo` app by using the following values (also see Figure 4-1).

- Group: `com.apress.todo`
- Artifact: `todo-in-memory`
- Name: `todo-in-memory`
- Package Name: `com.apress.todo`
- Dependencies: Web, Lombok

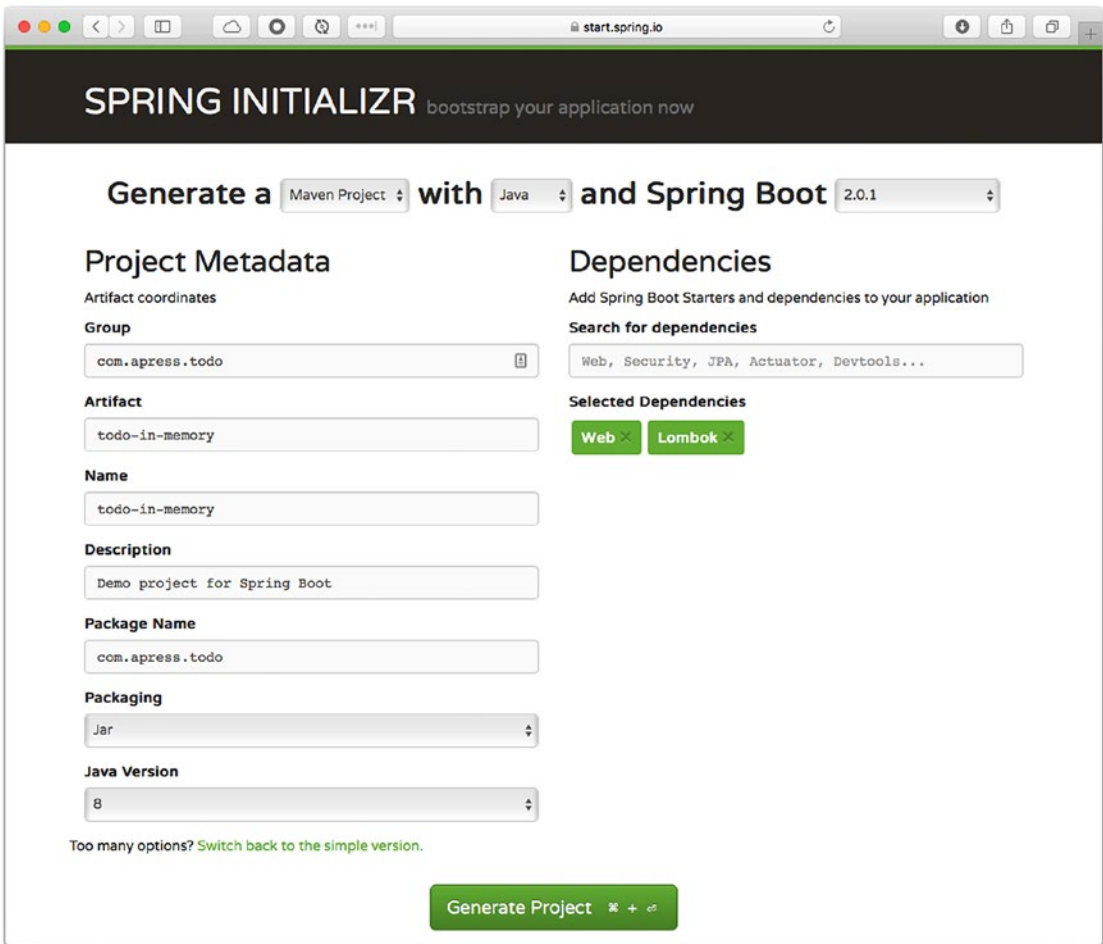


Figure 4-1. <https://start.spring.io> ToDo app

Choosing the Lombok dependency helps easily create the domain model classes and eliminates the boilerplate setters, getters, and other overrides.

Note If you need more information about Lombok, see the reference docs at <https://projectlombok.org>.

You can select either Maven or Gradle as the project type; in this book, we use both indistinctly. Press the Generate Project button and download the ZIP file. Uncompress it and import it into your favorite IDE. Some of the best IDEs are STS (<https://spring.io/tools/sts/all>), IntelliJ IDEA (www.jetbrains.com/idea/), and VSCode

(<https://code.visualstudio.com/>). I recommend one of these IDEs for the code completion feature, which helps you see the methods or parameters to add to your code.

Domain Model: ToDo

Based on the requirements, you need to create a ToDo domain model class (see Listing 4-1).

Listing 4-1. `com.apress.todo.domain.ToDo.java`

```
package com.apress.todo.domain;

import lombok.Data;

import javax.validation.constraints.NotBlank;
import javax.validation.constraints.NotNull;
import java.time.LocalDateTime;
import java.util.UUID;

@Data
public class ToDo {

    @NotNull
    private String id;
    @NotNull
    @NotBlank
    private String description;
    private LocalDateTime created;
    private LocalDateTime modified;
    private boolean completed;

    public ToDo(){
        LocalDateTime date = LocalDateTime.now();
        this.id = UUID.randomUUID().toString();
        this.created = date;
        this.modified = date;
    }
}
```

```

    public Todo(String description){
        this();
        this.description = description;
    }
}

```

Listing 4-1 shows you the `Todo` class, which has all the required fields. It also uses the `@Data` annotation, which is a Lombok annotation that generates a default constructor (if you don't have one) and all the setters, getters, and overrides, such as the `toString` method, to make the class cleaner. Also note that the class has the `@NotNull` and `@NotBlank` annotations in some of the fields; these annotations are used in the validation that we do later on. The default constructor has field initialization, so it is easy to create a `Todo` instance.

Fluent API: `ToDoBuilder`

Next let's create a Fluent API class that helps create a `Todo` instance. You can see this class a factory that creates a `Todo` with a description or with a particular ID (see Listing 4-2).

Listing 4-2. `com.apress.todo.domain.ToDoBuilder.java`

```

package com.apress.todo.domain;

public class ToDoBuilder {

    private static ToDoBuilder instance = new ToDoBuilder();
    private String id = null;
    private String description = "";

    private ToDoBuilder(){

    }

    public static ToDoBuilder create() {
        return instance;
    }

    public ToDoBuilder withDescription(String description){
        this.description = description;
        return instance;
    }
}

```

```

public ToDoBuilder withId(String id){
    this.id = id;
    return instance;
}

public ToDo build(){
    ToDo result = new ToDo(this.description);
    if(id != null)
        result.setId(id);
    return result;
}
}

```

Listing 4-2 is a simple Factory class that creates a `ToDo` instance. You extend its functionality in the next chapters.

Repository: `CommonRepository<T>`

Next, create an interface that has common persistence actions. This interface is generic, so it easy to use any other implementation, making the repo an extensible solution (see Listing 4-3).

Listing 4-3. `com.apress.todo.repository.CommonRepository<T>.java`

```

package com.apress.todo.repository;

import java.util.Collection;

public interface CommonRepository<T> {
    public T save(T domain);
    public Iterable<T> save(Collection<T> domains);
    public void delete(T domain);
    public T findById(String id);
    public Iterable<T> findAll();
}

```

Listing 4-3 is a common interface that can be used as a base for any other persistence implementation. Of course, you can change these signatures at any time. This is just an example on how to create something that is extensible.

Repository: `ToDoRepository`

Let's create a concrete class that implements the `CommonRepository<T>` interface. Remember the specification; for now, it is necessary only to have the `ToDo`'s in memory (see Listing 4-4).

Listing 4-4. `com.apress.todo.repository.ToDoRepository.java`

```
package com.apress.todo.repository;

import com.apress.todo.domain.ToDo;
import org.springframework.stereotype.Repository;

@Repository
public class ToDoRepository implements CommonRepository<ToDo> {

    private Map<String,ToDo> toDos = new HashMap<>();

    @Override
    public ToDo save(ToDo domain) {
        ToDo result = toDos.get(domain.getId());
        if(result != null) {
            result.setModified(LocalDateTime.now());
            result.setDescription(domain.getDescription());
            result.setCompleted(domain.isCompleted());
            domain = result;
        }
        toDos.put(domain.getId(), domain);
        return toDos.get(domain.getId());
    }

    @Override
    public Iterable<ToDo> save(Collection<ToDo> domains) {
        domains.forEach(this::save);
        return findAll();
    }
}
```

```

@Override
public void delete(ToDo domain) {
    toDos.remove(domain.getId());
}

@Override
public ToDo findById(String id) {
    return toDos.get(id);
}

@Override
public Iterable<ToDo> findAll() {
    return toDos.entrySet().stream().sorted(entryComparator).
        map(Map.Entry::getValue).collect(Collectors.toList());
}

private Comparator<Map.Entry<String,ToDo>> entryComparator = (Map.
Entry<String, ToDo> o1, Map.Entry<String, ToDo> o2) -> {
    return o1.getValue().getCreated().compareTo(o2.getValue().
getCreated());
};
}

```

Listing 4-4 shows the implementation of the `CommonRepository<T>` interface. Review the code and analyze it. This class is using a hash that holds all the `ToDo`'s. All the operations get simplify due nature of the hash, making it easy to implement.

Validation: `ToDoValidationError`

Next, let's create a validation class that exposes any possible errors in the app, such as a `ToDo` with no description. Remember that in the `ToDo` class, the ID and description fields are marked as `@NotNull`. The description field has an extra `@NotBlank` annotation to make sure that it is never empty (see Listing 4-5).

Listing 4-5. `com.apress.todo.validation.ToDoValidationError.java`

```
package com.apress.todo.validation;

import com.fasterxml.jackson.annotation.JsonInclude;

import java.util.ArrayList;
import java.util.List;

public class ToDoValidationError {

    @JsonInclude(JsonInclude.Include.NON_EMPTY)
    private List<String> errors = new ArrayList<>();

    private final String errorMessage;

    public ToDoValidationError(String errorMessage) {
        this.errorMessage = errorMessage;
    }

    public void addValidationError(String error) {
        errors.add(error);
    }

    public List<String> getErrors() {
        return errors;
    }

    public String getErrorMessage() {
        return errorMessage;
    }
}
```

Listing 4-5 shows the `ToDoValidationError` class, which holds any errors that arise with any requests. It uses an extra `@JsonInclude` annotation, which says that even if the errors field is empty, it must be included.

Validation: ToDoValidationErrorBuilder

Let's create another factory that helps build the `ToDoValidationError` instance (see Listing 4-6).

Listing 4-6. `com.apress.todo.validation.ToDoValidationErrorBuilder.java`

```
package com.apress.todo.validation;

import org.springframework.validation.Errors;
import org.springframework.validation.ObjectError;

public class ToDoValidationErrorBuilder {

    public static ToDoValidationError fromBindingErrors(Errors errors) {
        ToDoValidationError error = new ToDoValidationError("Validation
        failed. " + errors.getErrorCount() + " error(s)");
        for (ObjectError objectError : errors.getAllErrors()) {
            error.addValidationError(objectError.getDefaultMessage());
        }
        return error;
    }
}
```

Listing 4-6 is another Factory class that easily creates a `ToDoValidationError` instance with all the necessary information.

Controller: ToDoController

Now, it's time to create the RESTful API and use all the previous classes. You create the `ToDoController` class, in which you see all the Spring MVC features, the annotations, the way to configure endpoints, and how to handle errors. Let's review the code in Listing 4-7.

Listing 4-7. `com.apress.todo.controller.ToDoController.java`

```
package com.apress.todo.controller;

import com.apress.todo.domain.ToDo;
import com.apress.todo.domain.ToDoBuilder;
import com.apress.todo.repository.CommonRepository;
```

```

import com.apress.todo.validation.ToDoValidationError;
import com.apress.todo.validation.ToDoValidationErrorBuilder;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.validation.Errors;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.servlet.support.ServletUriComponentsBuilder;

import javax.validation.Valid;
import java.net.URI;

```

@RestController**@RequestMapping("/api")**

```

public class ToDoController {

    private CommonRepository<ToDo> repository;

    @Autowired
    public ToDoController(CommonRepository<ToDo> repository) {
        this.repository = repository;
    }

    @GetMapping("/todo")
    public ResponseEntity<Iterable<ToDo>> getTodos(){
        return ResponseEntity.ok(repository.findAll());
    }

    @GetMapping("/todo/{id}")
    public ResponseEntity<ToDo> getToDoById(@PathVariable String id){
        return ResponseEntity.ok(repository.findById(id));
    }

    @PatchMapping("/todo/{id}")
    public ResponseEntity<ToDo> setCompleted(@PathVariable String id){
        ToDo result = repository.findById(id);
        result.setCompleted(true);
        repository.save(result);
    }

```

```

URI location = ServletUriComponentsBuilder.fromCurrentRequest()
    .buildAndExpand(result.getId()).toUri();

return ResponseEntity.ok().header("Location",location.toString()).
    build();
}

```

```

@RequestMapping(value="/todo", method = {RequestMethod.POST,
RequestMethod.PUT})

```

```

public ResponseEntity<?> createToDo(@Valid @RequestBody ToDo todo,
Errors errors){

```

```

    if (errors.hasErrors()) {
        return ResponseEntity.badRequest().
body(ToDoValidationErrorHandler.fromBindingErrors(errors));
    }

```

```

    ToDo result = repository.save(todo);
    URI location = ServletUriComponentsBuilder.fromCurrentRequest().
        path("/{id}")
        .buildAndExpand(result.getId()).toUri();
    return ResponseEntity.created(location).build();
}

```

```

@DeleteMapping("/todo/{id}")

```

```

public ResponseEntity<ToDo> deleteToDo(@PathVariable String id){
    repository.delete(ToDoBuilder.create().withId(id).build());
    return ResponseEntity.noContent().build();
}

```

```

@DeleteMapping("/todo")

```

```

public ResponseEntity<ToDo> deleteToDo(@RequestBody ToDo todo){
    repository.delete(todo);
    return ResponseEntity.noContent().build();
}

```

```

@ExceptionHandler
  @ResponseStatus(value = HttpStatus.BAD_REQUEST)
  public ToDoValidationError handleException(Exception exception) {
    return new ToDoValidationError(exception.getMessage());
  }
}

```

Listing 4-7 is the `ToDoController` class. Let's review it.

- `@RestController`. Spring MVC offers the `@Controller` and `@RestController` to express request mappings, request input, exception handling, and more. All the functionality relies on these annotations, so there is no need to extend or implement interfaces specific interfaces.
- `@RequestMapping`. This annotation maps requests to controller methods. There are several attributes to match URLs, HTTP methods (GET, PUT, DELETE, etc.), request parameters, headers, and media types. It can be use at class level (to share mappings) or at method level for specific endpoint mapping. In this case, it is marked with `"/api"`, meaning that all the methods have this prefix.
- `@Autowired`. The constructor is annotated with `@Autowired`, meaning that it injects the `CommonRepository<ToDo>` implementation. This annotation can be omitted; Spring automatically injects any declared dependency since version 4.3.
- `@GetMapping`. This is a shortcut variant of the `@RequestMapping` annotation, useful for HTTP GET methods. `@GetMapping` is equivalent to `@RequestMapping(value="/todo", method = {RequestMethod.GET})`.
- `@PatchMapping`. This is a shortcut variant of the `@RequestMapping` annotation; in this class, it marks a `ToDo` as completed.
- `@DeleteMapping`. This is a shortcut variant of the `@RequestMapping` annotation; it is used to delete a `ToDo`. There are two overload methods: `deleteToDo`, one accepting a `String` and the other a `ToDo` instance.

- `@PathVariable`. This annotation is useful when you declare an endpoint that contains a URL expression pattern; in this case, `"/api/todo/{id}"`, where the ID must match the name of the method parameter.
- `@RequestBody`. This annotation sends a request with a body. Normally, when you submit a form or a particular content, this class receives a JSON format `ToDo`, then the `HttpMessageConverter` deserializes the JSON into a `ToDo` instance; this is done automatically thanks to Spring Boot and its auto-configuration because it registers the `MappingJackson2HttpMessageConverter` by default.
- `ResponseEntity<T>`. This class returns a full response, including HTTP headers, and the body is converted through `HttpMessageConverters` and written to the HTTP response. The `ResponseEntity<T>` class supports a fluent API, so it is easy to create the response.
- `@ResponseStatus`. Normally, this annotation is used when a method has a void return type (or null return value). This annotation sends back the HTTP status code specified in the response.
- `@Valid`. This annotation validates incoming data and is used as a method's parameters. To trigger a validator, it is necessary to annotate the data you want to validate with `@NotNull`, `@NotBlank`, and other annotations. In this case, the `ToDo` class uses those annotations in the ID and description fields. If the validator finds errors, they are collected in the `Errors` class (in this case, a *hibernate validator* that came with the `spring-webmvc` jars is registered and used as a global validator; you can create your own custom validation and override Spring Boot's defaults). Then you can inspect and add the necessary logic to send back an error response.
- `@ExceptionHandler`. The Spring MVC automatically declares built-in resolvers for exceptions and adds the support to this annotation. In this case, the `@ExceptionHandler` is declared inside this controller class (or you can use it within a `@ControllerAdvice` interceptor) and any exception is redirected to the `handleException` method. You can be more specific if needed. For example, you can have a `DataAccessException` and handle through a method.

```

    @ExceptionHandler
    @ResponseStatus(value = HttpStatus.BAD_REQUEST)
    public ToDoValidationError
        handleException(DataAccessException exception) {
        return new
            ToDoValidationError(exception.getMessage());
    }

```

In the class there is a method that accepts two HTTP methods: POST and PUT. `@RequestMapping` can accept multiple HTTP methods, so it is easy to assign one method to process them (e.g., `@RequestMapping(value="/todo", method = {RequestMethod.POST, RequestMethod.PUT})`).

We have covered all the necessary requirements for this application, so it's time to run it and see the results.

Running: ToDo App

Now, you are ready to run the ToDo app and test it. If you are using an IDE (STS or IntelliJ), you can right-click the main app class (`TodoInMemoryApplication.java`) and select Run Action. If you are using an editor that doesn't have these features, you can run your ToDo app by opening a terminal window and executing the commands in Listing 4-8 or Listing 4-9.

Listing 4-8. If you are using Maven project type

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

Listing 4-9. If you are using Gradle project type

```
./gradlew spring-boot:run
```

Spring Initializr (<https://start.spring.io>) always provides the project type wrappers you selected (Maven or Gradle wrappers), so there is no need to preinstall Maven or Gradle.

One of defaults for Spring Boot web apps is that it configures an embedded Tomcat server, so you can easily run your app without deploying it to an application servlet container. By default, it chooses port 8080.

Testing: ToDo App

Testing the ToDo app should be very simple. This testing is through commands or a specific client. If you are thinking about unit or integration testing, I'll explain that in another chapter. Here we are going to use the cURL command. This command comes in any UNIX OS flavor by default, but if you are a Windows user, you can download it from <https://curl.haxx.se/download.html>.

When running for the first time, the ToDo app shouldn't have any ToDo's. You can make sure of this by executing the following command in another terminal.

```
curl -i http://localhost:8080/api/todo
```

You should see something similar to this output:

```
HTTP/1.1 200
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked
Date: Wed, 02 May 2018 22:10:19 GMT

[]
```

You are targeting the `/api/todo` endpoint, and if you take a look at Listing 4-7, a `getTodos` method returns `ResponseEntity<Iterable<ToDo>>`, which is a collection of ToDo's. The default response is a JSON format (see the `Content-Type` header). The response is sending back the HTTP headers and status.

Next, let's add some ToDo's with the following command.

```
curl -i -X POST -H "Content-Type: application/json" -d '{
"description":"Read the Pro Spring Boot 2nd Edition Book"}'
http://localhost:8080/api/todo
```

In the command, `post (-X POST)` and data (`-d`) are JSON format. You are sending only the `description` field. It is necessary to add the header (`-H`) with the right content-type, and point to the `/api/todo` endpoint. After executing the command, you should see output like this:

```
HTTP/1.1 201
Location: http://localhost:8080/api/todo/d8d37c51-10a8-4c82-a7b1-
b72b5301cdab
Content-Length: 0
Date: Wed, 02 May 2018 22:21:09 GMT
```

You get back the location header, where the `ToDo` is read. `Location` exposes the ID of the `ToDo` you have just created. This response was generated by the `createToDo` method. Add at least another two `ToDo`'s so that we can have more data.

If you execute the first command one more time to get all the `ToDo`'s, you should see something like this:

```
curl -i http://localhost:8080/api/todo
HTTP/1.1 200
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked
Date: Wed, 02 May 2018 22:30:16 GMT

[{"id":"d8d37c51-10a8-4c82-a7b1-b72b5301cdab","description":"Read the Pro Spring Boot 2nd Edition Book","created":"2018-05-02T22:27:26.042+0000","modified":"2018-05-02T22:27:26.042+0000","completed":false}, {"id":"fbb20090-19f5-4abc-a8a9-92718c2c4759","description":"Bring Milk after work","created":"2018-05-02T22:28:23.249+0000","modified":"2018-05-02T22:28:23.249+0000","completed":false}, {"id":"2d051b67-7716-4ee6-9c45-1de939-fa579f","description":"Take the dog for a walk","created":"2018-05-02T22:29:28.319+0000","modified":"2018-05-02T22:29:28.319+0000","completed":false}]
```

Of course, this does not print very well, but you have the entire `ToDo`'s list. You can format this output by using another command-line tool: `jq` (<https://stedolan.github.io/jq/>).

```
curl -s http://localhost:8080/api/todo | jq
[
  {
    "id": "d8d37c51-10a8-4c82-a7b1-b72b5301cdab",
    "description": "Read the Pro Spring Boot 2nd Edition Book",
    "created": "2018-05-02T22:27:26.042+0000",
    "modified": "2018-05-02T22:27:26.042+0000",
    "completed": false
  },
  {
    "id": "fbb20090-19f5-4abc-a8a9-92718c2c4759",
    "description": "Bring Milk after work",
```



```

    "created": "2018-05-02T22:28:23.249+0000",
    "modified": "2018-05-02T22:28:23.249+0000",
    "completed": false
  },
  {
    "id": "2d051b67-7716-4ee6-9c45-1de939fa579f",
    "description": "Take the dog for a walk",
    "created": "2018-05-02T22:29:28.319+0000",
    "modified": "2018-05-02T22:29:28.319+0000",
    "completed": false
  }
]

```

Next, you can modify one of the `ToDo`'s; for example,

```

curl -i -X PUT -H "Content-Type: application/json" -d '{
"description":"Take the dog and the cat for a walk", "id":"2d051b67-7716-
4ee6-9c45-1de939fa579f"}' http://localhost:8080/api/todo
HTTP/1.1 201
Location: http://localhost:8080/api/todo/2d051b67-7716-4ee6-9c45-
1de939fa579f
Content-Length: 0
Date: Wed, 02 May 2018 22:38:03 GMT

```

Here `Take the dog for a walk` is changed to `Take the dog and the cat for a walk`. The command is using the `-X PUT` and the `id` field is needed (we can get it from the location header from previous `POST`s or from accessing the `/api/todo` endpoint). If you review all the `ToDo`'s, you have a modified `ToDo`.

Next, let's complete a `ToDo`. You can execute the following command.

```

curl -i -X PATCH http://localhost:8080/api/todo/2d051b67-7716-4ee6-9c45-
1de939fa579f
HTTP/1.1 200
Location: http://localhost:8080/api/todo/2d051b67-7716-4ee6-9c45-
1de939fa579f
Content-Length: 0
Date: Wed, 02 May 2018 22:50:27 GMT

```

The command is using the `-X PATCH` that process by the `setCompleted` method. If you review the location link, `ToDo` should be completed.

```
curl -s http://localhost:8080/api/todo/2d051b67-7716-4ee6-9c45-1de939fa579f
| jq
{
  "id": "2d051b67-7716-4ee6-9c45-1de939fa579f",
  "description": "Take the dog and the cat for a walk",
  "created": "2018-05-02T22:44:57.652+0000",
  "modified": "2018-05-02T22:50:27.691+0000",
  "completed": true
}
```

The `completed` field is now `true`. If this `ToDo` is completed, then you can delete it.

```
curl -i -X DELETE http://localhost:8080/api/todo/2d051b67-7716-4ee6-9c45-
1de939fa579f
HTTP/1.1 204
Date: Wed, 02 May 2018 22:56:18 GMT
```

The `cURL` command has `-X DELETE`, which is processed by the `deleteToDo` method, removing it from the hash. If you take a look at all the `ToDo`'s, you should now have one less than before.

```
curl -s http://localhost:8080/api/todo | jq
[
  {
    "id": "d8d37c51-10a8-4c82-a7b1-b72b5301cdab",
    "description": "Read the Pro Spring Boot 2nd Edition Book",
    "created": "2018-05-02T22:27:26.042+0000",
    "modified": "2018-05-02T22:27:26.042+0000",
    "completed": false
  },
  {
    "id": "fbb20090-19f5-4abc-a8a9-92718c2c4759",
    "description": "Bring Milk after work",
    "created": "2018-05-02T22:28:23.249+0000",
```

```

    "modified": "2018-05-02T22:28:23.249+0000",
    "completed": false
  }
]

```

Now, let's test the validation. Execute the following command.

```
curl -i -X POST -H "Content-Type: application/json" -d '{"description":""}'
http://localhost:8080/api/todo
```

The command is sending data (-d option) with the description field empty (normally, this happens when you are submitting a HTML form). You should see the following output.

```

HTTP/1.1 400
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked
Date: Thu, 03 May 2018 00:01:53 GMT
Connection: close

{"errors":["must not be blank"],"errorMessage":"Validation failed. 1 error(s)"}

```

A 400 status code (Bad Request) and the errors and errorMessage (built by the `ToDoValidationErrorBuilder` class) response. Use the following command.

```
curl -i -X POST -H "Content-Type: application/json" http://localhost:8080/
api/todo
```

```

HTTP/1.1 400
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked
Date: Thu, 03 May 2018 00:07:28 GMT
Connection: close

{"errorMessage":"Required request body is missing: public org.
springframework.http.ResponseEntity<?> com.apress.todo.controller.
ToDoController.createToDo(com.apress.todo.domain.ToDo,org.springframework.
validation.Errors)"}

```

This command is posting but no data and is responding with an error message. This is from the `@ExceptionHandler` annotation and the `handleException` method. All the errors (different from the description being blank) are handled by this method.

You can keep testing more `ToDo`'s or modify some of the validation annotations to see how they work.

Note If you don't have the `cURL` command or you can't install it, you can use any other REST client, such as PostMan (<https://www.getpostman.com>) or Insomnia (<https://insomnia.rest>). If you like command lines, then `Httpie` (<https://httpie.org>) is another good option; it uses Python.

Spring Boot Web: Overriding Defaults

Spring Boot web auto-configuration sets defaults to run a Spring web application. In this section, I show you how to override some of them.

You can override the web defaults by either creating your own configuration (XML or `JavaConfig`) and/or using the `application.properties` (or `.yml`) file.

Server Overriding

By default, the embedded Tomcat server starts on port: 8080, but you can easily change that by using the following property.

```
server.port=8081
```

One of the cool features of Spring is that you can apply the SpEL (Spring Expression Language) and apply it to these properties. For example, when you create an executable jar (`./mvnw package` or `./gradlew build`), you can pass some parameters when running your application. You can do the following

```
java -jar todo-in-memory-0.0.1-SNAPSHOT.jar --port=8787
```

and in your `application.properties`, you have something like this:

```
server.port=${port:8282}
```

This expression means that if you pass the `--port` argument, it takes that value; if not, its set to 8282. This is just a small taste of what you can do with SpEL, but if you want to know more, go to <https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#expressions>.

You can also change the server address, which is useful when you want to run your application using a particular IP.

```
server.address=10.0.0.7
```

You can change the context of your application.

```
server.servlet.context-path=/my-todo-app
```

And you can do a cURL like this:

```
curl -I http://localhost:8080/my-todo-app/api/todo
```

You can have Tomcat with SSL by using the following properties.

```
server.port=8443
server.ssl.key-store=classpath:keystore.jks
server.ssl.key-store-password=secret
server.ssl.key-password=secret
```

We revisit these properties and make our app work with SSL in a later chapter.

You can manage a session by using the following properties.

```
server.servlet.session.store-dir=/tmp
server.servlet.session.persistent=true
server.servlet.session.timeout=15

server.servlet.session.cookie.name=todo-cookie.dat
server.servlet.session.cookie.path=/tmp/cookies
```

You can enable HTTP/2 support if your environment supports it.

```
server.http2.enabled=true
```

JSON Date Format

By default, the date types are exposed in the JSON response in a long format; but you can change that by providing your own pattern in the following properties.

```
spring.jackson.date-format=yyyy-MM-dd HH:mm:ss
spring.jackson.time-zone=MST7MDT
```

These properties format the date and also use the time zone that you specified (if you want to know more about the available IDs you can execute `java.util.TimeZone#getAvailableIDs`). If you modify the `ToDo` app, run it, add some `ToDo`'s, and get the list. You should get a response like this:

```
curl -s http://localhost:8080/api/todo | jq
[
  {
    "id": "f52d1429-432d-43c5-946d-15c7fa5f50eb",
    "description": "Get the Pro Spring Boot 2nd Edition Book",
    "created": "2018-05-03 11:40:37",
    "modified": "2018-05-03 11:40:37",
    "completed": false
  }
]
```

If you want to know more about which properties exists, bookmark <https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html>.

Content-Type: JSON/XML

Spring MVC uses `HttpMessageConverters` (client and server side) to negotiate content conversion in an HTTP exchange. Spring Boot sets the defaults to JSON if it finds the Jackson libraries in the classpath. But what happens if you want to expose XML as well, and request JSON or XML content?

Spring Boot makes this very easy by adding an extra dependency and a property.

If you are using Maven, see Listing 4-10.

Listing 4-10. Maven: pom.xml

```
<dependency>
  <groupId>com.fasterxml.jackson.dataformat</groupId>
  <artifactId>jackson-dataformat-xml</artifactId>
</dependency>
```

Or if you are using Gradle, see Listing 4-11.

Listing 4-11. Gradle: build.gradle

```
compile('com.fasterxml.jackson.dataformat:jackson-dataformat-xml')
```

You can add the following property to the application.properties file.

```
spring.mvc.contentnegotiation.favor-parameter=true
```

If you run the `ToDo` app with these changes, you can have a response in an XML format by executing the following.

```
curl -s http://localhost:8080/api/todo?format=xml
<ArrayList><item><id>b3281340-b1aa-4104-b3d2-77a96a0e41b8</
id><description>Read the Pro Spring Boot 2nd Edition Book</
description><created>2018-05-03T19:18:30.260+0000</created><modified>2018-
05-03T19:18:30.260+0000</modified><completed>>false</completed></item></
ArrayList>
```

In the previous command, `?format=xml` is appended to the URL; the same goes for a JSON response.

```
curl -s http://localhost:8080/api/todo?format=json | jq
[
  {
    "id": "b3281340-b1aa-4104-b3d2-77a96a0e41b8",
    "description": "Read the Pro Spring Boot 2nd Edition Book",
    "created": "2018-05-03T19:18:30.260+0000",
    "modified": "2018-05-03T19:18:30.260+0000",
    "completed": false
  }
]
```

If you want to prettily print the XML, in UNIX environments there exists the `xmllint` command.

```
curl -s http://localhost:8080/api/todo?format=xml | xmllint --format -
```

Spring MVC: Overriding Defaults

So far I haven't showed you how to create Web apps that expose a combination of technologies such as HTML, JavaScript and so for, and this is because nowadays the industry is leaning towards to use JavaScript/TypeScript apps for the front end.

This doesn't mean that you can't create a Spring Web MVC with backend and frontend. Spring Web MVC offers you a very well integration with Template Engines and other technologies.

If you are using any template engine you can choose the view prefix and suffix by using the following properties:

```
spring.mvc.view.prefix=/WEB-INF/my-views/  
spring.mvc.view.suffix=.jsp
```

As you can see, Spring Boot can help you with a lot of configuration that normally you need to do extensively if you are doing regular Spring Web apps. This new way help you to accelerate your development. If you need more guidance on how to use Spring MVC and all its features you can take a look at the Reference documentation: <https://docs.spring.io/spring/docs/5.0.5.RELEASE/spring-framework-reference/web.html#mvc>

Using a Different Application Container

By default, Spring Boot uses Tomcat (for web servlet apps) as an application container and sets up an embedded server. If you want to override this default, you can do it by modifying the Maven `pom.xml` or the Gradle `build.gradle` files.

Using Jetty Server

The same changes apply to Undertow or Netty (see Listing 4-12 and Listing 4-13).

Listing 4-12. Maven - pom.xml

```
<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>
```

Listing 4-13. Gradle: build.gradle

```
configurations {
  compile.exclude module: "spring-boot-starter-tomcat"
}

dependencies {
  compile("org.springframework.boot:spring-boot-starter-web")
  compile("org.springframework.boot:spring-boot-starter-jetty")
  // ...
}
```

Spring Boot Web: Client

Another important feature of creating Spring Boot web applications is that the Spring Web MVC comes with a useful `RestTemplate` class that helps create clients.

ToDo Client App

Open your browser and go to <https://start.spring.io> site to create your ToDo client app by using the following values (also see Figure 4-2).

- Group: `com.apress.todo`
- Artifact: `todo-client`
- Name: `todo-client`
- Package Name: `com.apress.todo`
- Dependencies: Web, Lombok

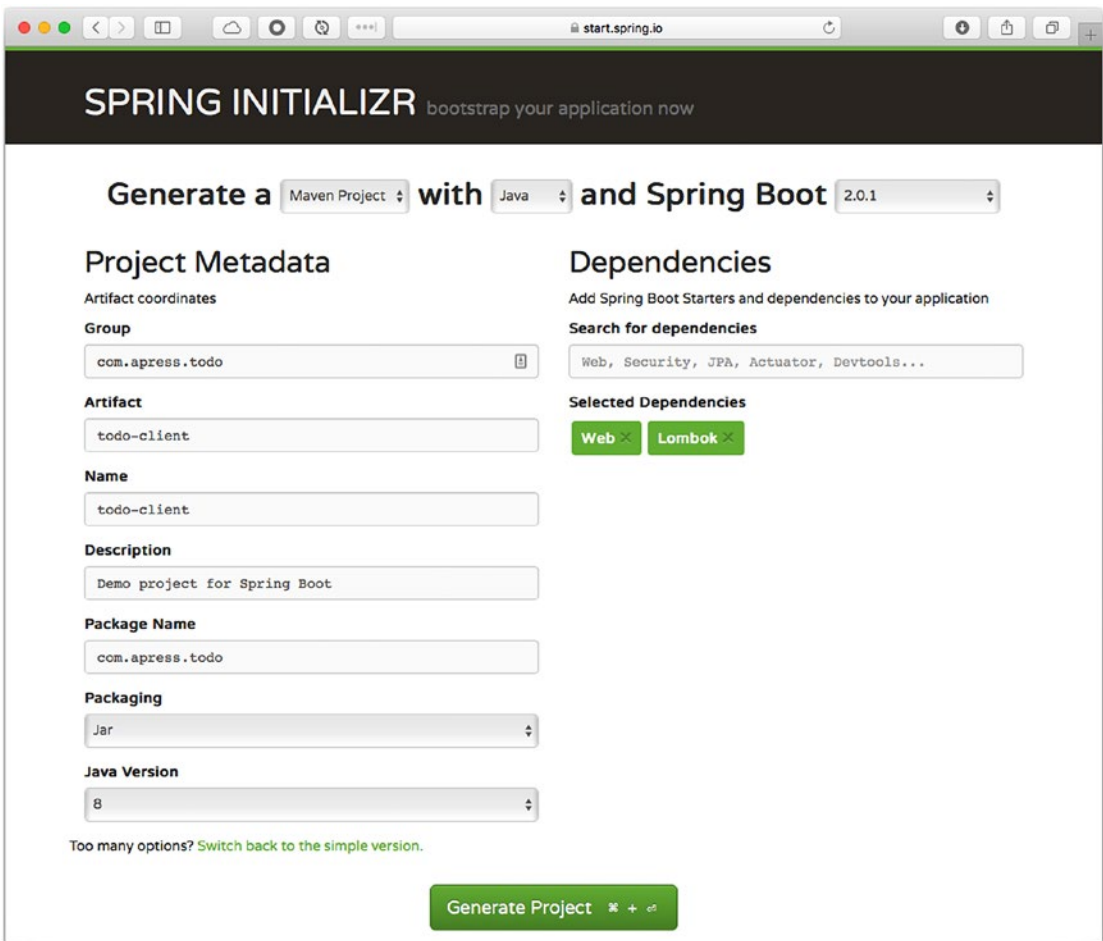


Figure 4-2. ToDo client app

Press the Generate Project button and download the ZIP file. Uncompress it and import it into your favorite IDE.

Domain Model: ToDo

Create the ToDo domain model, which should match the minimum fields from the previous application (see Listing 4-14).

Listing 4-14. `com.apress.todo.client.domain.ToDo.java`

```
package com.apress.todo.client.domain;

import lombok.Data;

import java.time.LocalDateTime;
import java.util.UUID;

@Data
public class ToDo {

    private String id;
    private String description;
    private LocalDateTime created;
    private LocalDateTime modified;
    private boolean completed;

    public ToDo(){
        LocalDateTime date = LocalDateTime.now();
        this.id = UUID.randomUUID().toString();
        this.created = date;
        this.modified = date;
    }

    public ToDo(String description){
        this();
        this.description = description;
    }
}
```

Listing 4-14 shows you the ToDo domain model class. The package name is different; there is no need to match the name of the class. The application doesn't need to know what the package is to serialize or deserialize into JSON.

Error Handler: `ToDoErrorHandler`

Next, let's create an error handler that takes care of any error responses that come from the server. Create the `ToDoErrorHandler` class (see Listing 4-15).

Listing 4-15. `com.apress.todo.client.error.ToDoErrorHandler.java`

```
package com.apress.todo.client.error;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.http.client.ClientHttpResponse;
import org.springframework.util.StreamUtils;
import org.springframework.web.client.DefaultResponseErrorHandler;

import java.io.IOException;
import java.nio.charset.Charset;

public class ToDoErrorHandler extends DefaultResponseErrorHandler {
    private Logger log = LoggerFactory.getLogger(ToDoErrorHandler.class);

    @Override
    public void handleError(ClientHttpResponse response)
throws IOException {
        log.error(response.getStatusCode().toString());
        log.error(StreamUtils.copyToString(
response.getBody(),Charset.defaultCharset()));
    }
}
```

Listing 4-15 shows the `ToDoErrorHandler` class, which is a custom class that extends from the `DefaultResponseErrorHandler`. So if we get a 400 HTTP status (Bad Request), we can catch the error and react to it; but in this case, the class is just logging the error.

Custom Properties: `ToDoRestClientProperties`

It's necessary to know where the `ToDo` app is running and which `basePath` to use, which is why it is necessary to hold that information. One of the best practices is to have this information externally.

Let's create a `ToDoRestClientProperties` class that holds the URL and the `basePath` information. This information can be saved in the `application.properties` file (see Listing 4-16).

Listing 4-16. `com.apress.todo.client.ToDoRestClientProperties.java`

```
package com.apress.todo.client;

import lombok.Data;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.stereotype.Component;

@Component
@ConfigurationProperties(prefix="todo")
@Data
public class ToDoRestClientProperties {

    private String url;
    private String basePath;

}
```

Listing 4-16 shows the class that holds the information about the URL and the `basePath`. Spring Boot allows you to create custom-typed properties that can be accessed and mapped from the `application.properties` file; the only requirement is that you need to mark the class with the `@ConfigurationProperties` annotation. This annotation can accept parameters like `prefix`.

In the `application.properties` file, add the following content.

```
todo.url=http://localhost:8080
todo.base-path=/api/todo
```

Client: ToDoRestClient

Let's create the client that uses the `RestTemplate` class, which helps exchange information between this client and the server. Create the `ToDoRestClient` class (see Listing 4-17).

Listing 4-17. `com.apress.todo.client.ToDoRestClient.java`

```
package com.apress.todo.client;

import com.apress.todo.client.domain.ToDo;
import com.apress.todo.client.error.ToDoErrorHandler;
import org.springframework.core.ParameterizedTypeReference;
import org.springframework.http.*;
import org.springframework.stereotype.Service;
import org.springframework.web.client.RestTemplate;

import java.net.URI;
import java.net.URISyntaxException;
import java.util.HashMap;
import java.util.Map;

@Service
public class ToDoRestClient {

    private RestTemplate restTemplate;
    private ToDoRestClientProperties properties;

    public ToDoRestClient(
        ToDoRestClientProperties properties){
        this.restTemplate = new RestTemplate();
        this.restTemplate.setErrorHandler(
            new ToDoErrorHandler());
        this.properties = properties;
    }
}
```

```

public Iterable<ToDo> findAll() throws URISyntaxException {
    RequestEntity<Iterable<ToDo>> requestEntity = new RequestEntity
    <Iterable<ToDo>>(HttpMethod.GET,new URI(properties.getUrl() +
    properties.getBasePath()));
    ResponseEntity<Iterable<ToDo>> response =
        restTemplate.exchange(requestEntity,new ParameterizedType
        Reference<Iterable<ToDo>>({});

    if(response.getStatusCode() == HttpStatus.OK){
        return response.getBody();
    }

    return null;
}

public ToDo findById(String id){
    Map<String, String> params = new HashMap<String, String>();
    params.put("id", id);
    return restTemplate.getForObject(properties.getUrl() + properties.
    getBasePath() +("/{id}",ToDo.class,params);
}

public ToDo upsert(ToDo toDo) throws URISyntaxException {
    RequestEntity<?> requestEntity = new
    RequestEntity<>(toDo,HttpMethod.POST,new URI(properties.getUrl() +
    properties.getBasePath()));
    ResponseEntity<?> response = restTemplate.exchange(requestEntity,
    new ParameterizedTypeReference<ToDo>() {});

    if(response.getStatusCode() == HttpStatus.CREATED){
        return restTemplate.getForObject(response.getHeaders().
        getLocation(),ToDo.class);
    }

    return null;
}

```

```

public Todo setCompleted(String id) throws URISyntaxException{
    Map<String, String> params = new HashMap<String, String>();
    params.put("id", id);
    restTemplate.postForObject(properties.getUrl() + properties.
        getBasePath() +("/{id}?_method=patch",null, ResponseEntity.class,
        params);

    return findById(id);
}

public void delete(String id){
    Map<String, String> params = new HashMap<String, String>();
    params.put("id", id);
    restTemplate.delete(properties.getUrl() + properties.getBasePath()
        +("/{id}",params);
}
}
}

```

Listing 4-17 shows the client that interacts with the `ToDo` application. This class is using the `RestTemplate` class. `RestTemplate` is Spring's central class for synchronous client-side HTTP access. It simplifies communication with HTTP servers and enforces RESTful principles. It handles HTTP connections, leaving application code to provide URLs (with possible template variables) and extract results. One of the many features allows you to handle your own error response. So take a look at the constructor and see that is setting the `ToDoErrorHandler` class.

Review the class; it contains all the actions that the `ToDo` app (Backend) has.

Note By default, `RestTemplate` relies on standard JDK facilities to establish HTTP connections. You can switch to use a different HTTP library, such as Apache `HttpComponents`, `Netty`, and `OkHttp`, through the `InterceptingHttpAccessor`. `setRequestFactory(org.springframework.http.client.ClientHttpRequestFactory)` property.

Running and Testing the Client

To run and test the client app, modify the `TodoClientApplication` class (see Listing 4-18).

Listing 4-18. `com.apress.todo.TODOClientApplication.java`

```
package com.apress.todo;

import com.apress.todo.client.ToDoRestClient;
import com.apress.todo.client.domain.ToDo;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.WebApplicationType;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
public class TodoClientApplication {

    public static void main(String[] args) {
        SpringApplication app = new
            SpringApplication(TodoClientApplication.class);
        app.setWebApplicationType(WebApplicationType.NONE);
        app.run(args);
    }

    private Logger log = LoggerFactory.getLogger(TodoClientApplication.
class);

    @Bean
    public CommandLineRunner process(TodoRestClient client){
        return args -> {

            Iterable<ToDo> toDos = client.findAll();
            assert toDos != null;
            toDos.forEach( toDo -> log.info(toDo.toString()));
        }
    }
}
```

```

        ToDo newToDo = client.upsert(new ToDo("Drink plenty of Water
        daily!"));
        assert newToDo != null;
        log.info(newToDo.toString());

        ToDo toDo = client.findById(newToDo.getId());
        assert toDos != null;
        log.info(toDo.toString());

        ToDo completed = client.setCompleted(newToDo.getId());
        assert completed.isCompleted();
        log.info(completed.toString());

        client.delete(newToDo.getId());
        assert client.findById(newToDo.getId()) == null;
    };
}
}

```

Listing 4-18 shows you how to test the client. First, turn off the web environment with `WebApplicationType.NONE`. Then use `CommandLineRunner` (as bean) to execute the code before the app starts.

Before running it, analyze the code to see what's going on. You can run using a command line or through your IDE. Make sure the `ToDo` app is up and running.

You will use this client again.

Note Remember that all the source code can be download it from the Apress website or at this GitHub location: <https://github.com/Apress/pro-spring-boot-2>.

Summary

In this chapter, you learned how Spring Boot manages and auto-configures web applications and how it uses the power of the Spring MVC. You also learned that you can override all the sensible defaults that auto-configuration provides.

With a `ToDo` app, you learned some of the Spring Boot features for web applications, such as JSON and XML configurations, the usage of several MVC annotations, such as `@RequestMapping` and `@ExceptionHandler`, and more.

You learned how Spring Boot uses an embedded application container (in this case, Tomcat was the default) to easily deploy or transport.

All the annotations that you used in the `ToDo` app are part of Spring MVC. In the remainder of the book, we implement more. If you want to know more about Spring MVC, take a look at *Pro Spring 5* (Apress, 2017).

In the next chapter, you learn how Spring Boot auto-configures Spring Data modules for persistence engines. You also learn more about Spring JDBC, JPA, REST, and NoSQL modules, such as Mongo.

CHAPTER 5

Data Access with Spring Boot

Data has become the most important part of the IT world, from trying to access, persist, and analyze it, to using a few bytes to petabytes of information. There have been many attempts to create frameworks and libraries to facilitate a way for developers to interact with the data, but sometimes this becomes too complicated.

After version 3.0, the Spring Framework created different teams that specialized in the different technologies. The Spring Data project team was born. This particular project's goal is to simplify uses of data access technologies, from relational and non-relational databases, to map-reduce frameworks and cloud-based data services. This Spring Data project is a collection of subprojects specific to a given database.

This chapter covers data access with Spring Boot using the `ToDo` application from previous chapters. You are going to make the `ToDo` app work with SQL and NoSQL databases. Let's get started.

SQL Databases

Do you remember those days when (in the Java world) you needed to deal with all the JDBC (Java Database Connectivity) tasks? You had to download the correct drivers and connection strings, open and close connections, SQL statements, result sets, and transactions, and convert from result sets to objects. In my opinion, these are all very manual tasks. Then a lot of ORM (object-relational mapping) frameworks started to emerge to manage these tasks—frameworks like Castor XML, ObjectStore, and Hibernate, to mention a few. They allowed you to identify the domain classes and create XML that was related to the database's tables. At some point, you also needed to be an expert to manage those kinds of frameworks.

The Spring Framework helped a lot with those frameworks by following the *template design pattern*. It allowed you create an abstract class that defined ways to execute the methods and created the database abstractions that allowed you to focus only on your business logic. It left all the hard lifting to the Spring Framework, including handling connections (open, close, and pooling), transactions, and the way you interact with the frameworks.

It's worth mentioning that the Spring Framework relies on several interfaces and classes (like the `javax.sql.DataSource` interface) to get information about the database you are going to use, how to connect to it (by providing a connection string), and its credentials. Now, if you have transaction management to do, the `DataSource` interface is essential. Normally, the `DataSource` interface requires the `Driver` class, the JDBC URL, a username, and a password to connect to the database.

Spring Data

The Spring Data team has created some of the amazing data-driven frameworks available today for the Java and Spring community. Their mission is to provide familiar and consistent Spring-based programming for data access and total control of the underlying data store technology that you want to use.

The Spring Data project is the umbrella for several additional libraries and data frameworks, which makes it easy to use data access technologies for relational and non-relation databases (a.k.a. NoSQL).

The following are some of the Spring Data features.

- Support for cross-store persistence
- Repository-based and custom object-mapping abstractions
- Dynamic queries based on method names
- Easy Spring integration via `JavaConfig` and XML
- Support for Spring MVC controllers
- Events for transparent auditing (created, last changes)

There are plenty more features—an entire book would be needed to cover each one of them. This chapter covers just enough to create powerful data-driven applications. Remember that Spring Data is the main umbrella project for everything that is covered.

Spring JDBC

In this section, I show you how to use the `JdbcTemplate` class. This particular class implements the template design pattern, which is a concrete class that exposes defined ways or templates to execute its methods. It hides all the boilerplate algorithms or a set of instructions. In Spring, you can choose different ways to form the basis for your JDBC database access; using the `JdbcTemplate` class is the classic Spring JDBC approach and this is the lowest level.

When you are using the `JdbcTemplate` class, you only need to implement callback interfaces to create an easy way to interact with any database engine. The `JdbcTemplate` class requires a `javax.sql.DataSource` and can be used in any class, by declaring it in `JavaConfig`, XML, or by annotations. The `JdbcTemplate` class takes care of all `SQLExceptions` and are properly handled.

You can use `NamedParameterJdbcTemplate` (a `JdbcTemplate` wrapper) to provide named parameters (`:parameterName`), instead of the traditional JDBC "?" placeholders. This is another option for your SQL queries.

The `JdbcTemplate` class exposes different methods.

- Querying (SELECT). You normally use the `query`, `queryForObject` method calls.
- Updating (INSERT/UPDATE/DELETE). You use the `update` method call.
- Operations (database/table/functions). You use the `execute` and `update` method calls.

With Spring JDBC, you have the ability to call stored procedures by using the `SimpleJdbcCall` class and manipulating the result with a particular `RowMapper` interface. `RowMapper<T>` is used by the `JdbcTemplate` class for mapping rows of `ResultSet` on a per-row basis.

Spring JDBC has support for embedded database engines, such as HSQL, H2 and Derby. It is easy to configure and offers quick startup time and testability.

Another feature is the ability to initialize the database with scripts; you can use embedded support or not. You can add your own schemas and data in a SQL format.

JDBC with Spring Boot

The Spring Framework has the support for working with SQL databases either using JDBC or ORMs (I cover this in the following sections). Spring Boot brings even more to data applications.

Spring Boot uses auto-configuration to set sensible defaults when it finds out that your application has a JDBC JARs. Spring Boot auto-configures the datasource based on the SQL driver in your classpath. If it finds that you have any of the embedded database engines (H2, HSQL or Derby), it is configured by default; in other words, you can have two driver dependencies (e.g., MySQL and H2), and if Spring Boot doesn't find any declared datasource bean, it creates it based on the embedded database engine JAR in your classpath (e.g., H2). Spring Boot also configures HikariCP as connection pool manager by default. Of course, you can override these defaults.

If you want to override the defaults, then you need to provide your own datasource declaration, either JavaConfig, XML, or in the `application.properties` file.

src/main/resources/application.properties

```
# Custom DataSource
spring.datasource.username=springboot
spring.datasource.password=rocks!
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/testdb?autoReconnect=true
&useSSL=false
```

Spring Boot supports JNDI connections if you are deploying your app in an application container. You can set the JNDI name in the `application.properties` file.

```
spring.datasource.jndi-name=java:jboss/ds/todos
```

Another feature that Spring Boot brings to data apps is that if you have a file named `schema.sql`, `data.sql`, `schema-<platform>.sql`, or `data-<platform>.sql` in the classpath, it initializes your database by executing those script files.

So, If you want to use JDBC in your Spring Boot application, you need to add the `spring-boot-starter-jdbc` dependency and your SQL driver.

ToDo App with JDBC

It's time to work with the ToDo app as we did in the previous chapter. You can start from scratch or you can follow along. If you are starting from scratch, then you can go to Spring Initializr (<https://start.spring.io>) and add the following values to the fields.

- Group: `com.apress.todo`
- Artifact: `todo-jdbc`
- Name: `todo-jdbc`
- Package Name: `com.apress.todo`
- Dependencies: Web, Lombok, JDBC, H2, MySQL

You can select either Maven or Gradle as the project type. Then you can press the Generate Project button, which downloads a ZIP file. Uncompress it and import the project in your favorite IDE (see Figure 5-1).

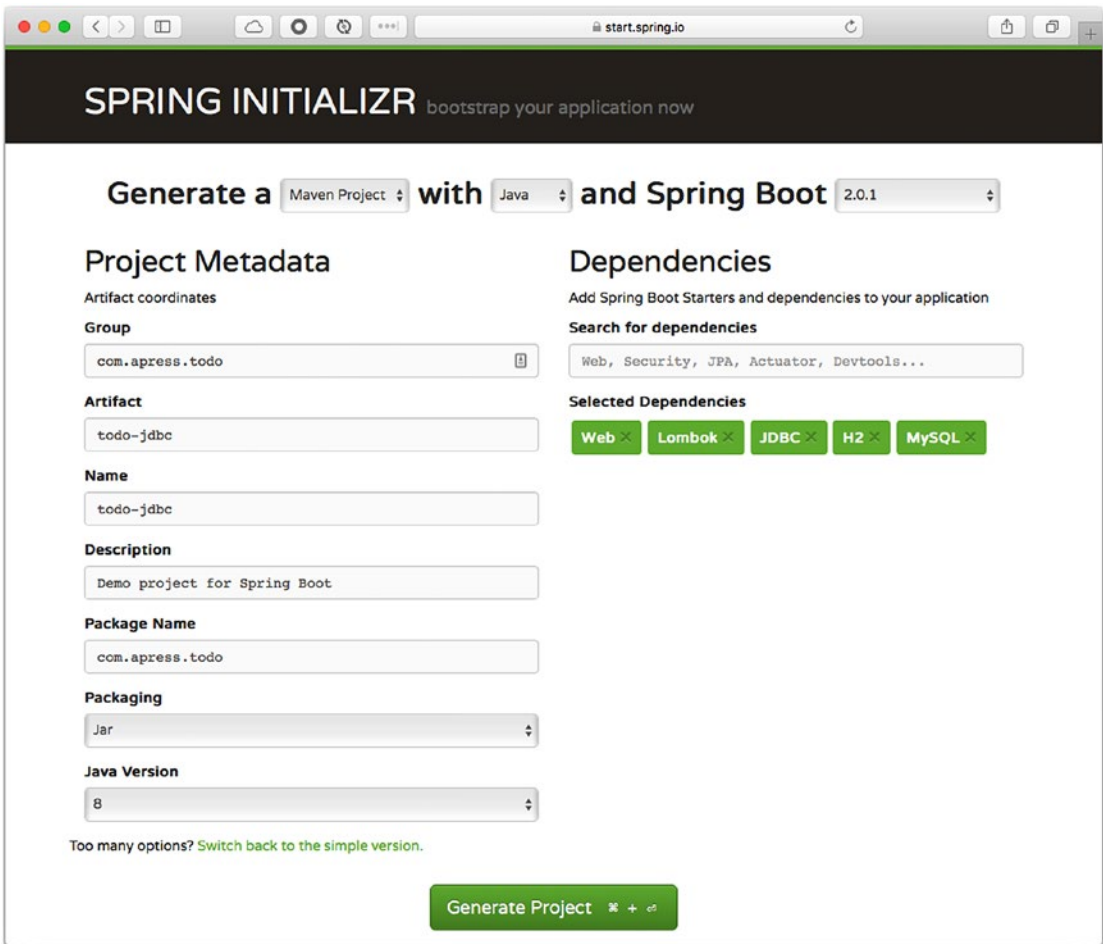


Figure 5-1. Spring Initializr <https://start.spring.io>

You can copy all the classes from the previous chapter, except for the `ToDoRepository` class; this is the only class that is new. Also make sure that in the `pom.xml` or `build.gradle` files, there are two drivers: H2 and MySQL. Based on what I discussed in the previous section, if I don't specify any `datasource` (in the `JavaConfig`, XML or `application.properties`) what does Spring Boot auto-configuration do? Correct! Spring Boot *auto-configures* the H2 embedded database by default.

Repository: ToDoRepository

Create a `ToDoRepository` class that implements the `CommonRepository` interface (see Listing 5-1).

Listing 5-1. `com.apress.todo.repository.ToDoRepository.java`

```
package com.apress.todo.repository;

import com.apress.todo.domain.ToDo;
import org.springframework.dao.EmptyResultDataAccessException;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
import org.springframework.stereotype.Repository;

import java.sql.ResultSet;
import java.time.LocalDateTime;
import java.util.*;

@Repository
public class ToDoRepository implements CommonRepository<ToDo> {

    private static final String SQL_INSERT = "insert into todo (id,
description, created, modified, completed) values (:id,:description,
:created,:modified,:completed)";
    private static final String SQL_QUERY_FIND_ALL = "select id,
description, created, modified, completed from todo";
    private static final String SQL_QUERY_FIND_BY_ID = SQL_QUERY_FIND_ALL +
" where id = :id";
    private static final String SQL_UPDATE = "update todo set description =
:description, modified = :modified, completed = :completed
where id = :id";
    private static final String SQL_DELETE = "delete from todo where id = :id";

    private final NamedParameterJdbcTemplate jdbcTemplate;

    public ToDoRepository(NamedParameterJdbcTemplate jdbcTemplate){
        this.jdbcTemplate = jdbcTemplate;
    }
}
```

```

private RowMapper<ToDo> toDoRowMapper = (ResultSet rs, int rowNum) -> {
    ToDo toDo = new ToDo();
    toDo.setId(rs.getString("id"));
    toDo.setDescription(rs.getString("description"));
    toDo.setModified(rs.getTimestamp("modified").toLocalDateTime());
    toDo.setCreated(rs.getTimestamp("created").toLocalDateTime());
    toDo.setCompleted(rs.getBoolean("completed"));
    return toDo;
};

@Override
public ToDo save(final ToDo domain) {
    ToDo result = findById(domain.getId());
    if(result != null){
        result.setDescription(domain.getDescription());
        result.setCompleted(domain.isCompleted());
        result.setModified(LocalDateTime.now());
        return upsert(result, SQL_UPDATE);
    }
    return upsert(domain,SQL_INSERT);
}

private ToDo upsert(final ToDo toDo, final String sql){
    Map<String, Object> namedParameters = new HashMap<>();
    namedParameters.put("id",toDo.getId());
    namedParameters.put("description",toDo.getDescription());
    namedParameters.put("created",java.sql.Timestamp.valueOf(toDo.
        getCreated()));
    namedParameters.put("modified",java.sql.Timestamp.valueOf(toDo.
        getModified()));
    namedParameters.put("completed",toDo.isCompleted());

    this.jdbcTemplate.update(sql,namedParameters);

    return findById(toDo.getId());
}

```

```

@Override
public Iterable<ToDo> save(Collection<ToDo> domains) {
    domains.forEach( this::save);
    return findAll();
}

@Override
public void delete(final ToDo domain) {
    Map<String, String> namedParameters = Collections.
        singletonMap("id", domain.getId());
    this.jdbcTemplate.update(SQL_DELETE,namedParameters);
}

@Override
public ToDo findById(String id) {
    try {
        Map<String, String> namedParameters = Collections.
            singletonMap("id", id);
        return this.jdbcTemplate.queryForObject(SQL_QUERY_FIND_BY_ID,
            namedParameters, toDoRowMapper);
    } catch (EmptyResultDataAccessException ex) {
        return null;
    }
}

@Override
public Iterable<ToDo> findAll() {
    return this.jdbcTemplate.query(SQL_QUERY_FIND_ALL, toDoRowMapper);
}
}

```

Listing 5-1 shows the `ToDoRepository` class that uses the `JdbcTemplate`, not directly though. This class is using `NamedParameterJdbcTemplate` (a `JdbcTemplate` wrapper) that helps with all the named parameters, which means that instead of using `?` in your SQL statements, you use names like `:id`.

This class is also declaring a `RowMapper`; remember that the `JdbcTemplate` used the `RowMapper` for mapping rows of a `ResultSet` on a per-row basis.

Analyze the code and check out every method implementation that is using plain SQL in every method.

Database Initialization: `schema.sql`

Remember that the Spring Framework allows you to initialize your database—creating or altering any table, or inserting/updating data when your application starts. To initialize a Spring app (not Spring Boot), it is necessary to add configuration (XML or JavaConfig); but the `ToDo` app is Spring Boot. If Spring Boot finds the `schema.sql` and/or `data.sql` files, it executes them automatically. Let's create the `schema.sql` (see Listing 5-2).

Listing 5-2. `src/main/resources/schema.sql`

```
DROP TABLE IF EXISTS todo;
CREATE TABLE todo
(
  id varchar(36) not null primary key,
  description varchar(255) not null,
  created timestamp,
  modified timestamp,
  completed boolean
);
```

Listing 5-2 shows the `schema.sql` that executes when the application is starting, and because H2 is the default datasource that is configured, then this script is executed against the H2 engine.

Running and Testing: `ToDo App`

Now it's time to run and test the `ToDo` app. You can run it within your IDE, or if you are using Maven, execute

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

If you are using Gradle, execute

```
./gradlew bootRun
```

To test the `ToDo` app, you can run your `ToDoClient` app. It should work without any problems.

H2 Console

Now that you ran the ToDo app, how can you make sure that the app is saving the data in the H2 engine? Spring Boot has a property that enables a H2 console so that you can interact with it. It is very useful for development purposes but not for production environments.

Modify the `application.properties` file and add the following property.

src/main/resources/application.properties

```
spring.h2.console.enabled=true
```

Restart the ToDo app, add values with a cURL command, and go to your browser and hit `http://localhost:8080/h2-console`. (see Figure 5-2).

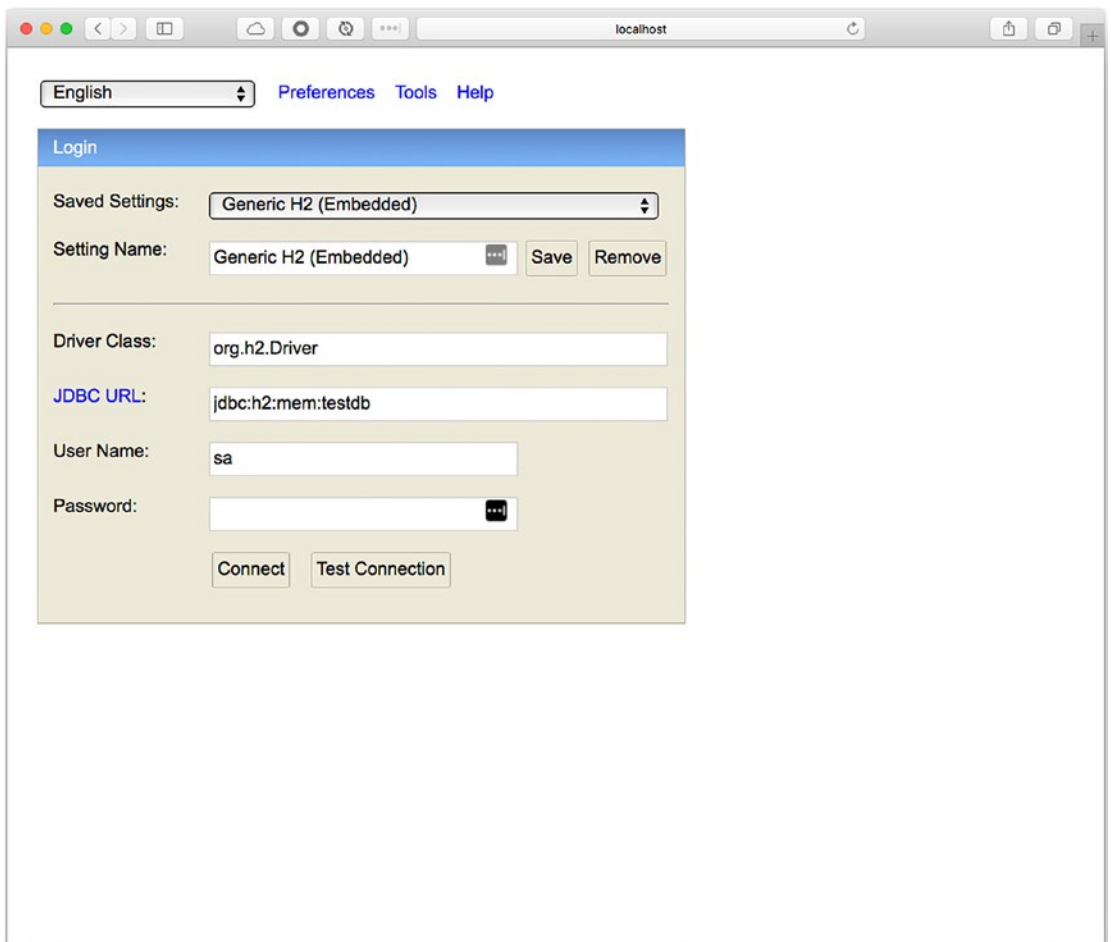


Figure 5-2. `http://localhost:8080/h2-console`

Figure 5-2 shows the H2 console, which you can reach at the /h2-console endpoint. (you can override this endpoint as well). The JDBC URL must be jdbc:h2:mem:testdb (sometimes this is different, so change it to that value). By default, the database name is testdb (but you can override this as well). If you click the Connect button, you get a different view, in which you can see the table and data being created (see Figure 5-3).

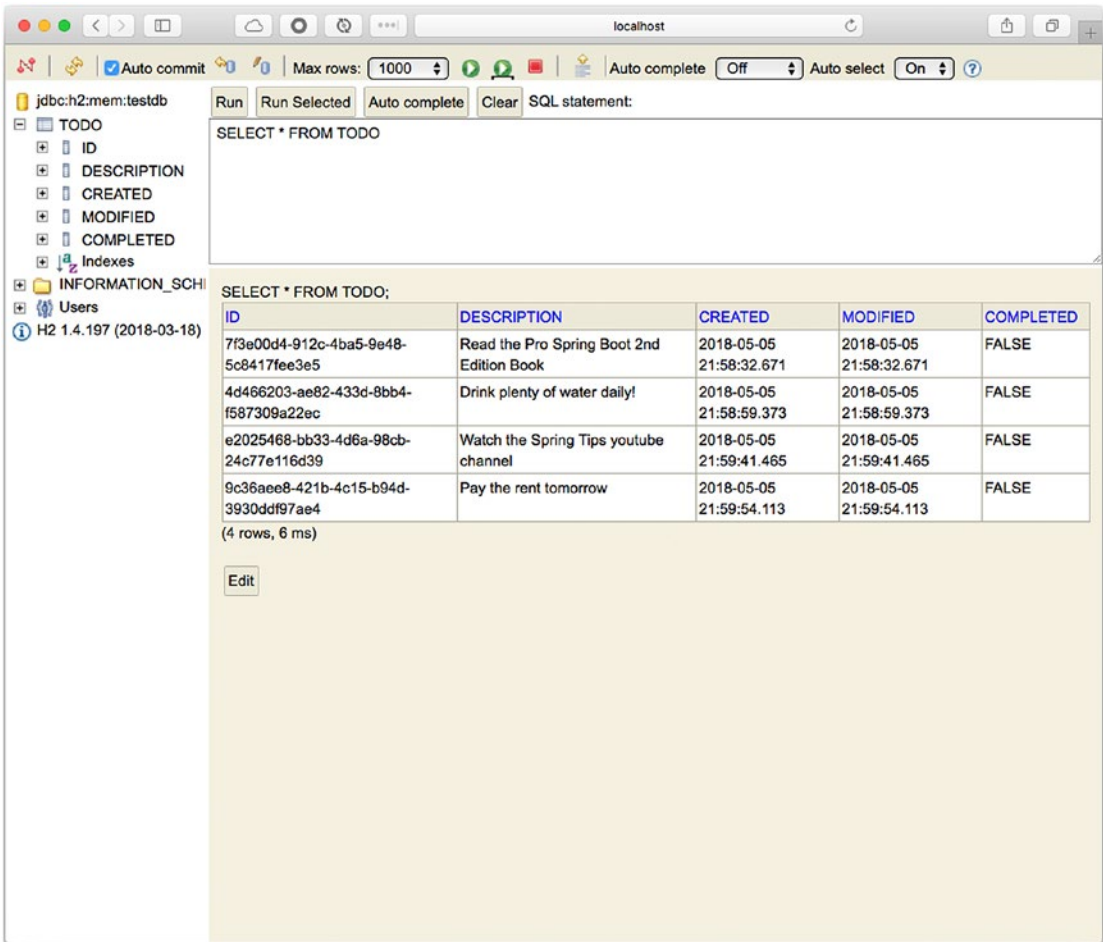


Figure 5-3. <http://localhost:8080/h2-console>

Figure 5-3 shows that you can execute any SQL query and get data back. If you want to see which SQL queries are being executed in the ToDo app, you can add the following properties to the application.properties file.

src/main/resources/application.properties

```
logging.level.org.springframework.data=INFO
logging.level.org.springframework.jdbc.core.JdbcTemplate=DEBUG
```

As you can see, the `JdbcTemplate` class offers you a lot of possibilities to interact with any database engine, but this class is the “lowest level” approach.

At the time of this writing, there was a new way to use the `JdbcTemplate` class in a more uniform way—the Spring Data way (something that I describe in the following sections). The Spring Data team has created the new Spring Data JDBC project, which follows the *aggregate root* concept, as described in the book *Domain-Driven Design* by Eric Evans (Addison-Wesley Professional, 2003). It has many features, such as CRUD operations, support for `@Query` annotations, support for MyBatis queries, events, and more, so keep an eye on this project. It is a new way to do JDBC.

Spring Data JPA

The JPA (Java Persistence API) provides a POJO persistence model for object-relational mapping. Spring Data JPA facilitates persistence with this model.

Implementing data access can be a hassle because we need to deal with connections, sessions, exception handling, and more, even for simple CRUD operations. That’s why the Spring Data JPA provides an additional level of functionality: creating repository implementations directly from interfaces and using conventions to generate queries from method names.

The following are some of the Spring Data JPA features.

- Support of the JPA specification with different providers, such as Hibernate, Eclipse Link, Open JPA, and so forth.
- Support for repositories (a concept from *Domain-Driven Design*).
- Auditing for domain class.
- Support for Quesydsl (<http://www.querydsl.com/>) predicates and type-safe JPA queries.
- Pagination, sort, dynamic query execution support.
- Support for `@Query` annotations.

- Support for XML-based entity mapping.
- JavaConfig based repository configuration by using the `@EnableJpaRepositories` annotation.

Spring Data JPA with Spring Boot

One of the most important benefits from the Spring Data JPA is that we don't need to worry about implementing basic CRUD functionalities, because that's what it does. We only need to create an interface that extends from a `Repository<T, ID>`, `CrudRepository<T, ID>`, or `JpaRepository<T, ID>`. The `JpaRepository` interface offers not only what the `CrudRepository` does, but also extends from the `PagingAndSortingRepository` interface that provides extra functionality. If you review the `CrudRepository<T, ID>` interface (which you use in your `ToDo` app), you can see all the signature methods, as shown in Listing 5-3.

Listing 5-3. `org.springframework.data.repository.CrudRepository.java`

```
@NoRepositoryBean
public interface CrudRepository<T, ID> extends Repository<T, ID> {
    <S extends T> S save(S entity);
    <S extends T> Iterable<S> saveAll(Iterable<S> entities);
    Optional<T> findById(ID id);
    boolean existsById(ID id);
    Iterable<T> findAll();
    Iterable<T> findAllById(Iterable<ID> ids);
    long count();
    void deleteById(ID id);
    void delete(T entity);
    void deleteAll(Iterable<? extends T> entities);
    void deleteAll();
}
```

Listing 5-3 shows the `CrudRepository<T, ID>` interface, where the `T` means the entity (your domain model class) and the `ID`, the primary key that needs to implement `Serializable`.

In a simple Spring app, you are required to use the `@EnableJpaRepositories` annotation that triggers the extra configuration that is applied in the life cycle of the repositories defined within of your application. The good thing is that you don't need this when using Spring Boot because Spring Boot takes care of it. Another feature from Spring Data JPA are the query methods, which are a very powerful way to create SQL statements with the fields of the domain entities.

So, to use Spring Data JPA with Spring Boot, you need `spring-boot-starter-data-jpa` and the SQL driver.

When Spring Boot executes its auto-configuration and finds out that you have the Spring Data JPA JAR, it configures the datasource by default (if there is none defined). It configures the JPA provider (by default it uses Hibernate). It enables the repositories (by using the `@EnableJpaRepositories` configuration). It checks if you have defined any query methods. And more.

ToDo App with Spring Data JPA

You can create your ToDo app from scratch, or take a look the classes you need, as well the necessary dependencies in your `pom.xml` or `build.gradle` files.

Starting from scratch, go to your browser and open Spring Initializr. Add the following values to the fields.

- Group: `com.apress.todo`
- Artifact: `todo-jpa`
- Name: `todo-jpa`
- Package Name: `com.apress.todo`
- Dependencies: Web, Lombok, JPA, H2, MySQL

You can select either Maven or Gradle as the project type. Then you can press the Generate Project button, which downloads a ZIP file. Uncompress it and import the project in your favorite IDE (see Figure 5-4).

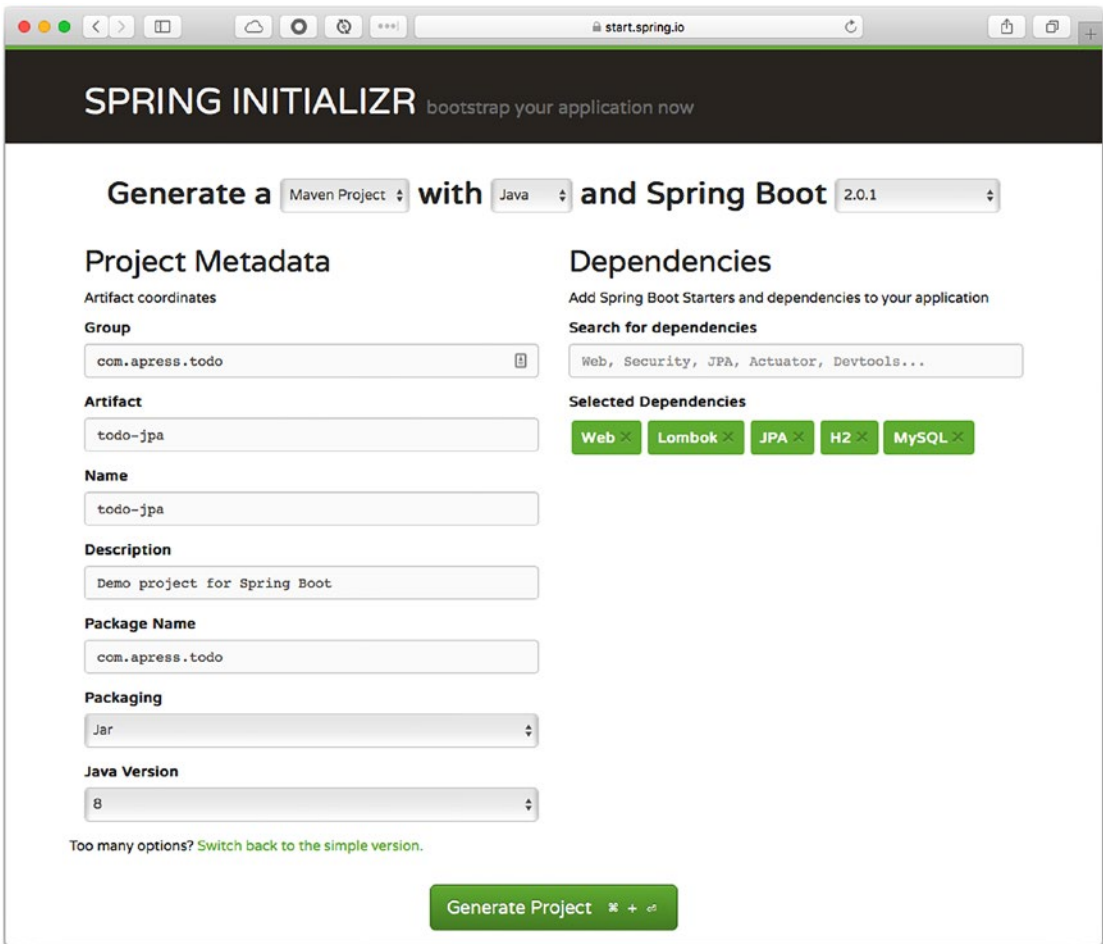


Figure 5-4. Spring Initializr <https://start.spring.io>

You can copy all the classes from the previous chapter, except for the `ToDoRepository` class, which is the only class that is new; you modify the others.

Repository: `ToDoRepository`

Create a `ToDoRepository` interface that extends from `CrudRepository<T, ID>`. The `T` is the `ToDo` class, and the `ID` is a `String` (see Listing 5-4).

Listing 5-4. `com.apress.todo.repository.ToDoRepository.java`

```
package com.apress.todo.repository;

import com.apress.todo.domain.ToDo;
import org.springframework.data.repository.CrudRepository;

public interface ToDoRepository extends
    CrudRepository<ToDo,String> {}
```

Listing 5-4 shows the `ToDoRepository` interface that extends a `CrudRepository`. It is not necessary to create a concrete class or implement anything; the Spring Data JPA does the implementation for us. All the CRUD actions handle anything that we need to persist the data. That's it—there's nothing else that we need to do to use `ToDoRepository` where we need it.

Domain Model: ToDo

To use JPA and be compliant, it is necessary to declare the entity (`@Entity`) and the primary key (`@Id`) from the domain model. Let's modify the `ToDo` class by adding the following annotations and methods (see Listing 5-5).

Listing 5-5. `com.apress.todo.domain.ToDo.java`

```
package com.apress.todo.domain;

import lombok.Data;
import lombok.NoArgsConstructor;
import org.hibernate.annotations.GenericGenerator;

import javax.persistence.*;
import javax.validation.constraints.NotBlank;
import javax.validation.constraints.NotNull;
import java.time.LocalDateTime;
```

@Entity

@Data

@NoArgsConstructor

```

public class ToDo {

    @NotNull
    @Id
    @GeneratedValue(generator = "system-uuid")
    @GenericGenerator(name = "system-uuid", strategy = "uuid")
    private String id;
    @NotNull
    @NotBlank
    private String description;

    @Column(insertable = true, updatable = false)
    private LocalDateTime created;
    private LocalDateTime modified;
    private boolean completed;

    public ToDo(String description){
        this.description = description;
    }

    @PrePersist
    void onCreate() {
        this.setCreated(LocalDateTime.now());
        this.setModified(LocalDateTime.now());
    }

    @PreUpdate
    void onUpdate() {
        this.setModified(LocalDateTime.now());
    }
}

```

Listing 5-5 shows the modified version of the `ToDo` domain model. This class now has additional elements.

- `@NoArgsConstructor`. This annotation belongs to the Lombok library. It creates a class constructor with no arguments. It is required that JPA have a constructor with no arguments.
- `@Entity`. This annotation specifies that the class is an entity and is persisted into the database engine selected.
- `@Id`. This annotation specifies the primary key of an entity. The field annotated should be any Java primitive type and any primitive wrapper type.
- `@GeneratedValue`. This annotation provides the generation strategies for the values of primary keys (simple keys only). Normally, it is used with the `@Id` annotation. There are different strategies (`IDENTITY`, `AUTO`, `SEQUENCE`, and `TABLE`) and a key generator. In this case, the class defined the "system-uuid" (this generates a unique 36-character ID).
- `@GenericGenerator`. This is part of the Hibernate, which allows you to use the strategy to generate a unique ID from the previous annotation.
- `@Column`. This annotation specifies the mapped column for persistent properties; if there is no column annotation in a field, it is the default name of the column in the database. This class is marking the created field to be only for inserts but never for updates.
- `@PrePersist`. This annotation is a callback that is triggered before any persistent action is taken. It sets the new timestamp for the created and modified fields before the record is inserted into the database.
- `@PreUpdate`. This annotation is another callback that is triggered before any update action is taken. It sets the new timestamp for the modified field before it is updated into the database.

The last two annotations (`@PrePersist` and `@PreUpdate`) are a very nice way to deal with dates/timestamps, making it easier for the developer.

Before we continue, analyze the code to see what is different from previous versions of the `ToDo` domain model class.

Controller: ToDoController

Now, it's time to modify the `ToDoController` class (see Listing 5-6).

Listing 5-6. `com.apress.todo.controller.ToDoController.java`

```
package com.apress.todo.controller;

import com.apress.todo.domain.ToDo;
import com.apress.todo.domain.ToDoBuilder;
import com.apress.todo.repository.ToDoRepository;
import com.apress.todo.validation.ToDoValidationError;
import com.apress.todo.validation.ToDoValidationErrorBuilder;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.validation.Errors;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.servlet.support.ServletUriComponentsBuilder;

import javax.validation.Valid;
import java.net.URI;
import java.util.Optional;

@RestController
@RequestMapping("/api")
public class ToDoController {

    private ToDoRepository toDoRepository;

    @Autowired
    public ToDoController(ToDoRepository toDoRepository) {
        this.toDoRepository = toDoRepository;
    }

    @GetMapping("/todo")
    public ResponseEntity<Iterable<ToDo>> getTodos(){
        return ResponseEntity.ok(toDoRepository.findAll());
    }
}
```

```

@GetMapping("/todo/{id}")
public ResponseEntity<ToDo> getToDoById(@PathVariable String id){
    Optional<ToDo> todo = todoRepository.findById(id);
    if(todo.isPresent())
        return ResponseEntity.ok(todo.get());
    return ResponseEntity.notFound().build();
}

@PatchMapping("/todo/{id}")
public ResponseEntity<ToDo> setCompleted(@PathVariable String id){
    Optional<ToDo> todo = todoRepository.findById(id);
    if(!todo.isPresent())
        return ResponseEntity.notFound().build();

    ToDo result = todo.get();
    result.setCompleted(true);
    todoRepository.save(result);

    URI location = ServletUriComponentsBuilder.fromCurrentRequest()
        .buildAndExpand(result.getId()).toUri();

    return ResponseEntity.ok().header("Location",location.toString()).
        build();
}

@RequestMapping(value="/todo", method = {RequestMethod.
POST,RequestMethod.PUT})
public ResponseEntity<?> createToDo(@Valid @RequestBody ToDo todo,
Errors errors){
    if (errors.hasErrors()) {
        return ResponseEntity.badRequest().
            body(ToDoValidationErrorBuilder.fromBindingErrors(errors));
    }

    ToDo result = todoRepository.save(todo);
    URI location = ServletUriComponentsBuilder.fromCurrentRequest().
        path("/{id}")

```



```

        .buildAndExpand(result.getId()).toUri();
    return ResponseEntity.created(location).build();
}

@DeleteMapping("/todo/{id}")
public ResponseEntity<ToDo> deleteToDo(@PathVariable String id){
    toDoRepository.delete(ToDoBuilder.create().withId(id).build());
    return ResponseEntity.noContent().build();
}

@DeleteMapping("/todo")
public ResponseEntity<ToDo> deleteToDo(@RequestBody ToDo toDo){
    toDoRepository.delete(toDo);
    return ResponseEntity.noContent().build();
}

@ExceptionHandler
@ResponseStatus(value = HttpStatus.BAD_REQUEST)
public ToDoValidationError handleException(Exception exception) {
    return new ToDoValidationError(exception.getMessage());
}
}

```

Listing 5-6 shows the modified `ToDoController` class. It now directly uses the `ToDoRepository` interface, and some of the methods, like `findById`, return a Java 8 `Optional` type.

Before we continue, analyze the class to see what is different from previous versions. Most of the code remains the same.

Spring Boot JPA Properties

Spring Boot provides properties that allow you to override defaults when using the Spring Data JPA. One of them is the ability to create the DDL (data definition language), which is turned off by default, but you can enable it to do reverse engineering from your domain model. In other words, this property generates the tables and any other relationships from your domain model classes.

Also you can tell your JPA provider to create, drop, update, or validate your existing DDL/data, which is a useful migration mechanism. Also, you can set a property to show you the SQL statements that are being executed against the database engine.

Add the necessary properties to the `application.properties` file, as shown in Listing 5-7.

Listing 5-7. `src/main/resources/application.properties`

```
# JPA
spring.jpa.generate-ddl=true
spring.jpa.hibernate.ddl-auto=create-drop
spring.jpa.show-sql=true

# H2
spring.h2.console.enabled=true
```

Listing 5-7 shows the `application.properties` and the JPA properties. It generates the table based on the domain model class and it creates the tables every time the application starts. The following are the possible values for the `spring.jpa.hibernate.ddl-auto` property.

- `create` (Creates the schema and destroys previous data).
- `create-drop` (Creates and then destroys the schema at the end of the session).
- `update` (Updates the schema, if necessary).
- `validate` (Validates the schema, makes no changes to the database).
- `none` (Disables DDL handling).

Running and Testing: ToDo App

Now it's time to run and test the ToDo app. You can run it within your IDE. If you are using Maven, execute

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

If you are using Gradle, execute

```
./gradlew bootRun
```

To test the `ToDo` app, you can run your `ToDoClient` app. It should work without any problems. Also you can send `ToDo`'s with the `cURL` command and see the H2 console (<http://localhost:8080/h2-console>).

Spring Data REST

The Spring Data REST project builds on top of the Spring Data repositories. It analyzes your domain model classes, and it exposes hypermedia-driven HTTP resources using HATEOAS (Hypermedia as the Engine of Application State, HAL +JSON). The following are some of the features.

- Exposes a discoverable RESTful API from your domain model classes using HAL as the media type.
- Supports pagination and exposes your domain class as collections.
- Exposes dedicated search resources for query methods defined in the repositories.
- Supports high customization for your own controllers if you want to extend the defaults.
- Allows hooking into the handling of REST requests by handling `Spring ApplicationEvents`.
- Brings a HAL browser to expose all the metadata; very useful for development purposes.
- Supports Spring Data JPA, Spring Data MongoDB, Spring Data Neo4j, Spring Data Solr, Spring Data Cassandra, and Spring Data Gemfire.

Spring Data REST with Spring Boot

If you want to use Spring Data REST in a regular Spring MVC app, you need to trigger its configuration by including the `RepositoryRestMvcConfiguration` class with the `@Import` annotation in your `JavaConfig` class (where you have your `@Configuration` annotation); but you don't need to anything if you are using Spring Boot directly. Spring Boot takes care of this thanks to the `@EnableAutoConfiguration` annotation.

If you want to use the Spring Data REST in a Spring Boot application, you need to include the `spring-boot-starter-data-rest` and `spring-boot-starter-data-*` technology dependencies, and/or the SQL driver if you are going to use SQL database engines.

ToDo App with Spring Data JPA and Spring Data REST

You can create your ToDo app from scratch, or take a look which classes you need, as well the necessary dependencies, in your `pom.xml` or `build.gradle` files.

Starting from scratch, go to your browser and open Spring Initializr . Add the following values to the fields.

- Group: `com.apress.todo`
- Artifact: `todo-rest`
- Name: `todo-rest`
- Package Name: `com.apress.todo`
- Dependencies: Web, Lombok, JPA, REST Repositories, H2, MySQL

You can select either Maven or Gradle as the project type. Then you can press the Generate Project button, which downloads a ZIP file. Uncompress it and import the project in your favorite IDE (see Figure 5-5).

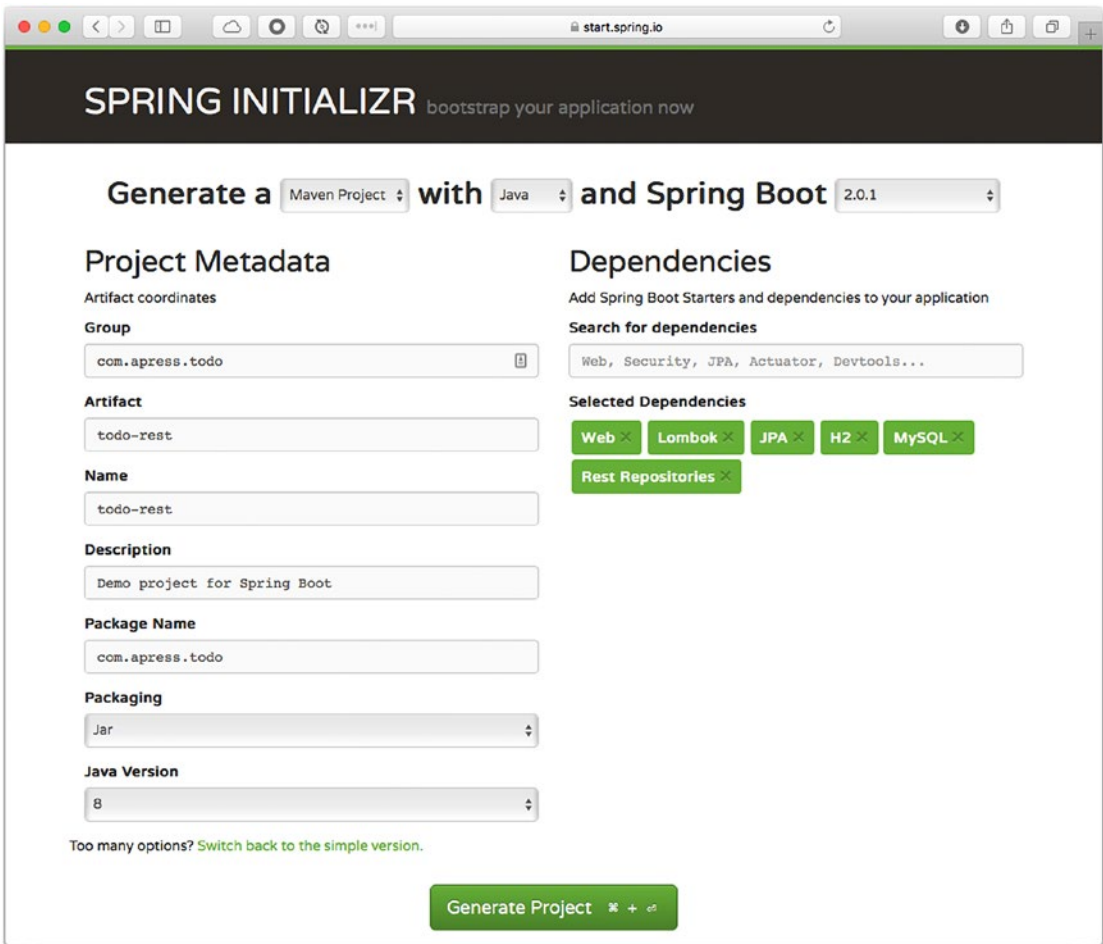


Figure 5-5. <https://start.spring.io>

You can copy only the domain model `ToDo`, `ToDoRepository` classes and the `application.properties` file; yes, only two classes and one properties file.

Running: ToDo App

Now it's time to run and test the `ToDo` app. You can run it within your IDE. If you are using Maven, execute

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

If you are using Gradle, execute

```
./gradlew bootRun
```

One of the important things to see is the output when running the ToDo app.

```

Mapped "{[/{repository}/search],methods=[HEAD],produces ...
Mapped "{[/{repository}/search],methods=[GET],produces= ...
Mapped "{[/{repository}/search],methods=[OPTIONS],produ ...
Mapped "{[/{repository}/search/{search}],methods=[GET], ...
Mapped "{[/{repository}/search/{search}],methods=[GET], ...
Mapped "{[/{repository}/search/{search}],methods=[OPTIO ...
Mapped "{[/{repository}/search/{search}],methods=[HEAD] ...
Mapped "{[/{repository}/{id}/{property}],methods=[GET], ...
Mapped "{[/{repository}/{id}/{property}/{propertyId}],m ...
Mapped "{[/{repository}/{id}/{property}],methods=[DELET ...
Mapped "{[/{repository}/{id}/{property}],methods=[GET], ...
Mapped "{[/{repository}/{id}/{property}],methods=[PATCH ...
Mapped "{[/{repository}/{id}/{property}/{propertyId}],m ...
Mapped "{[/ | | ],methods=[OPTIONS],produces=[applicatio ...
Mapped "{[/ | | ],methods=[HEAD],produces=[application/h ...
Mapped "{[/ | | ],methods=[GET],produces=[application/ha ...
Mapped "{[/{repository}],methods=[OPTIONS],produces=[ap ...
Mapped "{[/{repository}],methods=[HEAD],produces=[appli ...
Mapped "{[/{repository}],methods=[GET],produces=[applic ...
Mapped "{[/{repository}],methods=[GET],produces=[applic ...
Mapped "{[/{repository}],methods=[POST],produces=[appli ...
Mapped "{[/{repository}/{id}],methods=[OPTIONS],produce ...
Mapped "{[/{repository}/{id}],methods=[HEAD],produces=[ ...
Mapped "{[/{repository}/{id}],methods=[GET],produces=[a ...
Mapped "{[/{repository}/{id}],methods=[PUT],produces=[a ...
Mapped "{[/{repository}/{id}],methods=[PATCH],produces= ...
Mapped "{[/{repository}/{id}],methods=[DELETE],produces ...
Mapped "{[/profile/{repository}],methods=[GET],produces ...
Mapped "{[/profile/{repository}],methods=[OPTIONS],prod ...
Mapped "{[/profile/{repository}],methods=[GET],produces ...

```

It defines all the mapping endpoints of the repositories (only one in this app), and all the HTTP methods that you can use.

Testing: ToDo App

To test the ToDo app, we are going to use the cURL command and the browser. The ToDoClient app needs to be modified to accept the media type, HAL+JSON; so in this section, we won't use it. First take a look at your browser. Go to the `http://localhost:8080` you should see something similar to Figure 5-6.

First, take a look at your browser. Go to `http://localhost:8080`. You should see something similar to Figure 5-6.

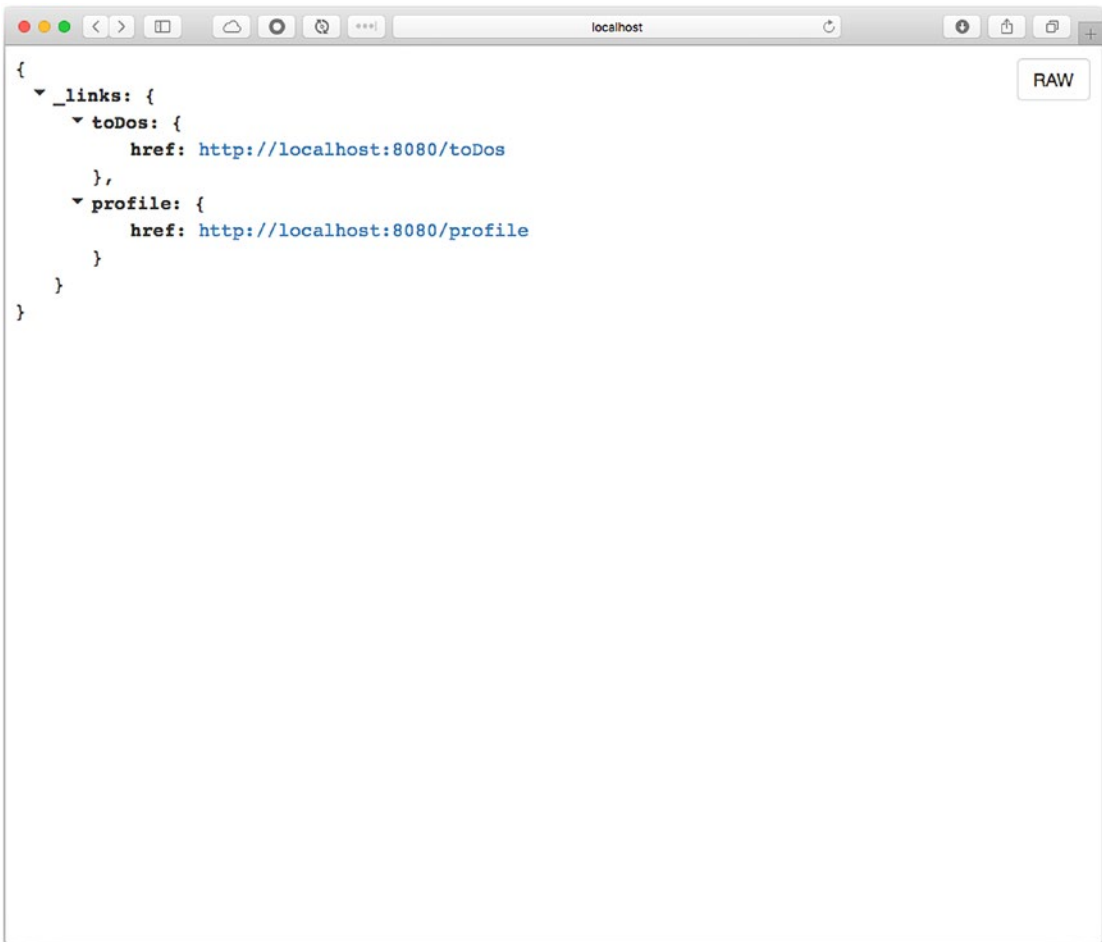


Figure 5-6. `http://localhost:8080`

If you see the same information but in RAW format, try to install the JSON Viewer plugin for your browser and reload the page. It exposes the `http://localhost:8080/todos` URL as the endpoint, which means that you can access and execute all the HTTP methods (from the logs) to this URL.

Let's add a few `ToDo`'s with a `cURL` command (is a single line).

```
curl -i -X POST -H "Content-Type: application/json" -d '{
"description":"Read the Pro Spring Boot 2nd Edition Book"}' http://
localhost:8080/todos
```

```
HTTP/1.1 201
```

```
Location: http://localhost:8080/todos/8a8080876338ae4e016338b2e2ee0000
```


```
Content-Type: application/hal+json;charset=UTF-8
```

```
Transfer-Encoding: chunked
```

```
Date: Mon, 07 May 2018 03:43:57 GMT
```

```
{
  "description" : "Read the Pro Spring Boot 2nd Edition Book",
  "created" : "2018-05-06T21:43:57.676",
  "modified" : "2018-05-06T21:43:57.677",
  "completed" : false,
  "_links" : {
    "self" : {
      "href" : "http://localhost:8080/todos/8a8080876338ae4e016338b2e2ee0000"
    },
    "todo" : {
      "href" : "http://localhost:8080/todos/8a8080876338ae4e016338b2e2ee0000"
    }
  }
}
```

You are getting a HAL+JSON result. After adding a couple more, you can go back to your browser and click the `http://localhost:8080/todos` link. You see something like Figure 5-7.



```

{
  "_embedded": {
    "toDos": [
      {
        "description": "Read the Pro Spring Boot 2nd Edition Book",
        "created": "2018-05-06T21:43:57.676",
        "modified": "2018-05-06T21:43:57.677",
        "completed": false,
        "_links": {
          "self": {
            "href": "http://localhost:8080/toDos/8a8080876338ae4e016338b2e2ee0000"
          },
          "toDo": {
            "href": "http://localhost:8080/toDos/8a8080876338ae4e016338b2e2ee0000"
          }
        }
      },
      {
        "description": "Drink plenty of water daily!",
        "created": "2018-05-06T22:08:32.72",
        "modified": "2018-05-06T22:08:32.72",
        "completed": false,
        "_links": {
          "self": {
            "href": "http://localhost:8080/toDos/8a8080876338ae4e016338c964d00001"
          },
          "toDo": {
            "href": "http://localhost:8080/toDos/8a8080876338ae4e016338c964d00001"
          }
        }
      },
      {
        "description": "Watch the Spring Tips youtube channel",

```

Figure 5-7. *http://localhost:8080/toDos*

Figure 5-7 shows the HAL+JSON response when accessing the /toDos endpoint.

Testing with HAL Browser: ToDo App

The Spring Data REST project has a tool—the HAL browser. It is a web app that helps developers visualize all the endpoints in an interactive way. So, if you don't want to use the endpoints and/or cURL commands directly, you can use the HAL browser.

To use the HAL browser, add the following dependency. If you are using Maven, add the following to your pom.xml file.

Maven pom.xml

```
<dependency>
  <groupId>org.springframework.data</groupId>
  <artifactId>spring-data-rest-hal-browser</artifactId>
</dependency>
```

If you are using Gradle, add the following to your `build.gradle` file.

Gradle build.gradle

```
compile 'org.springframework.data:spring-data-rest-hal-browser'
```

Now, you can restart your `ToDo` app and go directly to `http://localhost:8080` in your browser. You the same as what's shown in Figure 5-8.

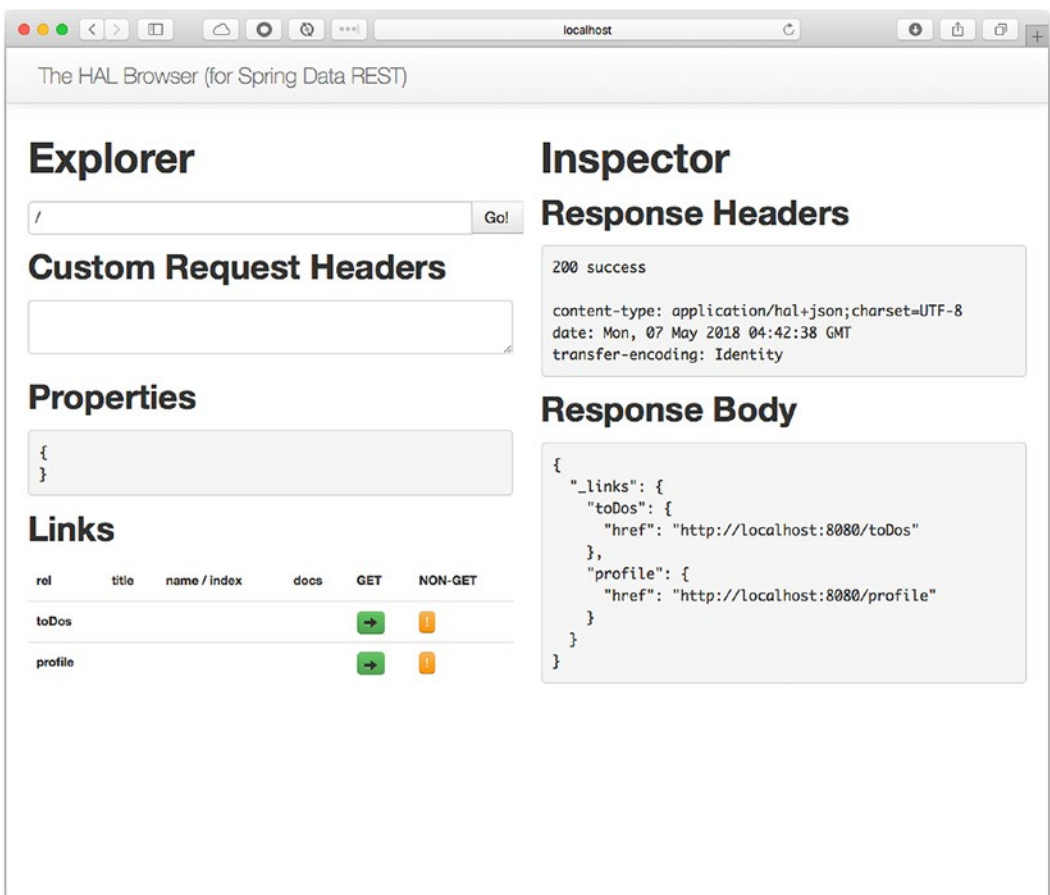


Figure 5-8. `http://localhost:8080`

You can click in the GET and NON-GET columns to interact with every single endpoint and HTTP method. This is a great alternative for a developer.

What I showed you is some of the many features that you can do with the Spring Data REST. You really don't need any controller anymore. Also, you are able to expose any domain class, thanks to the easy override in Spring Boot and Spring Data REST.

No SQL Databases

NoSQL databases are another way to persist data, but in a different way than the tabular relationships of relational databases. There is already a classification system for these emergent NoSQL databases. You can find it based on its data model.

- Column (Cassandra, HBase, etc.)
- Document (CouchDB, MongoDB, etc.)
- Key-Value (Redis, Riak, etc.)
- Graph (Neo4J, Virtuoso, etc.)
- Multimodel (OrientDB, ArangoDB, etc.)

As you can see, you have many options. I think the most important kind of feature finds a database that is scalable and can easily handle millions of records.

Spring Data MongoDB

The Spring Data MongoDB project gives you the necessary interactions with the MongoDB document database. One of the important features is that you still work with domain model classes that use the `@Document` annotation and declare interfaces that use `CrudRepository<T, ID>`. This creates the necessary collection that the MongoDB uses for persistence.

The following are some of the features of this project.

- Spring Data MongoDB offer a `MongoTemplate` helper class (very similar to `JdbcTemplate`) that deals with all the boilerplate interacting with the MongoDB document database.
- Persistence and mapping lifecycle events.

- `MongoTemplate` helper class. It also provides low-level mapping using the `MongoReader/MongoWriter` abstractions.
- Java-based query, criteria, and update DSLs.
- Geospatial and MapReduce integrations and GridFS support.
- Cross-storage persistence support for JPA entities. This means that you can use your classes marked with `@Entity` and other annotations, and use them to persist/retrieve data using the MongoDB document database.

Spring Data MongoDB with Spring Boot

To use MongoDB with Spring Boot, you need to add the `spring-boot-starter-data-mongodb` dependency and have access to a MongoDB server instance.

Spring Boot uses the auto-configuration feature to set up everything to communicate with the MongoDB server instance. By default, Spring Boot tries to connect to the local host and use port 27017 (the MongoDB standard port). If you have a MongoDB remote server, you can connect to it by overriding the defaults. You need to use the `spring.mongodb.*` properties in the `application.properties` file (the easy way), or you can have a bean declaration using XML or in a `JavaConfig` class.

Spring Boot also auto-configures the `MongoTemplate` class (this class is very similar to `JdbcTemplate`), so it's ready for any interaction with the MongoDB server. Another great feature is that you can work with *repositories*, meaning that you can reuse the same interface that you used for JPA.

MongoDB Installation

Before you start, you need to make sure that you have the MongoDB server installed on your computer.

If you are using Mac/Linux with the `brew` command (<http://brew.sh/>), execute the following command.

```
brew install mongodb
```

You can run it with this command.

```
mongod
```

Or you can install MongoDB by downloading it from the website at <https://www.mongodb.org/downloads#production> and following the instructions.

MongoDB Embedded

There is another way to use MongoDB, at least as a development environment. You can use MongoDB Embedded. Normally, you use this in a test environment, but you can easily run it in development mode with a runtime scope.

To use MongoDB Embedded, you need to add the following dependency. If you are using Maven, you can add it into the `pom.xml` file.

Maven pom.xml

```
<dependency>
  <groupId>de.flapdoodle.embed</groupId>
  <artifactId>de.flapdoodle.embed.mongo</artifactId>
  <scope>runtime</scope>
</dependency>
```

If you are using Gradle:

Gradle build.gradle

```
runtime('de.flapdoodle.embed:de.flapdoodle.embed.mongo')
```

Next, you need to configure the Mongo client to use the MongoDB Embedded server (see Listing 5-8).

Listing 5-8. `com.apress.todo.config.ToDoConfig.java`

```
package com.apress.todo.config;

import com.mongodb.MongoClient;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
```

```

import org.springframework.context.annotation.DependsOn;
import org.springframework.core.env.Environment;

@Configuration
public class ToDoConfig {

    private Environment environment;
    public ToDoConfig(Environment environment){
        this.environment = environment;
    }

    @Bean
    @DependsOn("embeddedMongoServer")
    public MongoClient mongoClient() {
        int port =
            this.environment.getProperty("local.mongo.port",
                                       Integer.class);
        return new MongoClient("localhost",port);
    }
}

```

Listing 5-8 shows the configuration of the MongoClient bean. MongoDB Embedded uses a random port when the application starts, that's why it is necessary to also use the Environment bean.

If you take this approach to use a MongoDB server, you don't need to set up any other properties.

ToDo App with Spring Data MongoDB

You can create your ToDo app from scratch, or take a look at which classes you need, as well the necessary dependencies, in your pom.xml or build.gradle files.

Starting from scratch, go to your browser and open Spring Initializr (<https://start.spring.io>). Add the following values to the fields.

- Group: com.apress.todo
- Artifact: todo-mongo
- Name: todo-mongo

- Package Name: `com.apress.todo`
- Dependencies: Web, Lombok, MongoDB, Embedded MongoDB

You can select either Maven or Gradle as the project type. Then you can press the Generate Project button, which downloads a ZIP file. Uncompress it and import the project in your favorite IDE (see Figure 5-9).

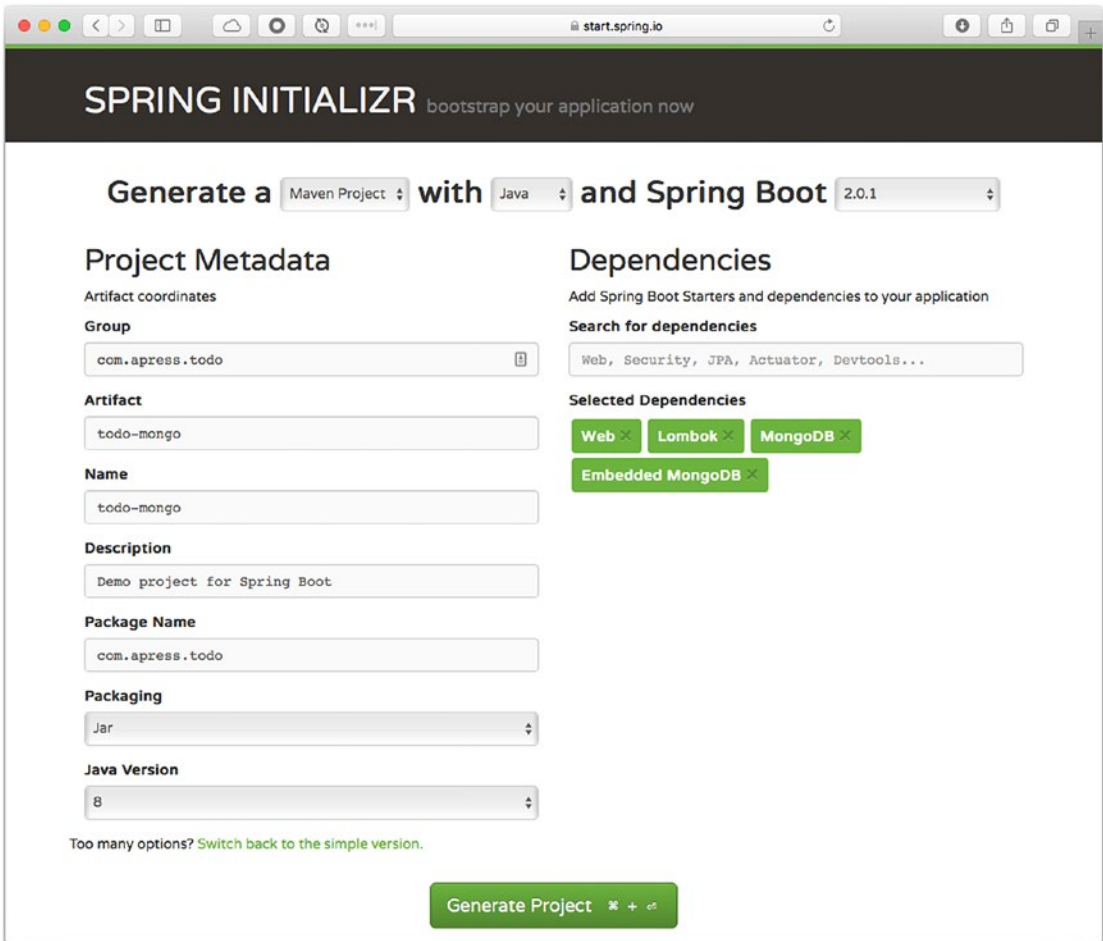


Figure 5-9. <https://start.spring.io>

You can copy all the classes from the `todo-jpa` project. In the next section, you see which classes need to be modified.

Domain Model: ToDo

Open the `ToDo` domain model class and modify it accordingly (see Listing 5-9).

Listing 5-9. `com.apress.todo.domain.ToDo.java`

```
package com.apress.todo.domain;

import lombok.Data;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;
import javax.validation.constraints.NotBlank;
import javax.validation.constraints.NotNull;
import java.time.LocalDateTime;
import java.util.UUID;

@Document
@Data
public class ToDo {

    @NotNull
    @Id
    private String id;
    @NotNull
    @NotBlank
    private String description;

    private LocalDateTime created;
    private LocalDateTime modified;
    private boolean completed;

    public ToDo(){
        LocalDateTime date = LocalDateTime.now();
        this.id = UUID.randomUUID().toString();
        this.created = date;
        this.modified = date;
    }
}
```



```

    public Todo(String description){
        this();
        this.description = description;
    }
}

```

Listing 5-9 shows the modified `Todo` class. The class is using the `@Document` annotation to be marked as persistent; it is also using `@Id` to declare a unique key.

If you have a remote MongoDB server, you can override the defaults that point at the local host. You can go to your `application.properties` file and add the following properties.

```

## MongoDB
spring.data.mongodb.host=my-remote-server
spring.data.mongodb.port=27017
spring.data.mongodb.username=myuser
spring.data.mongodb.password=secretpassword

```

Next, you can review your `ToDoRepository` and `ToDoController` classes, which should not change at all. That's the beauty of using Spring Data: you can reuse your model for cross-store and all the previous classes, making development easier and faster.

Running and Testing: ToDo App

Now it's time to run and test the `ToDo` app. You can run it within your IDE. Or if you are using Maven, execute

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

If you are using Gradle, execute

```
./gradlew bootRun
```

To test the `ToDo` app, you can run your `ToDoClient` app—and that's it. It is very easy to switch persistence engines without too much hassle. Maybe you are wondering what happen if you want to use *map-reduce* or lower-level operations. Well, you can by using the `MongoTemplate` class.

ToDo App with Spring Data MongoDB REST

How can you create a MongoDB REST app? You need to add the `spring-boot-starter-data-rest` dependency to your `pom.xml` or `build.gradle` files—and that's it!! Of course, you need to remove the controller and validation packages and the `ToDoBuilder` class; you only need two classes.

Remember that Spring Data REST exposes the repository endpoints and uses HATEOAS as the media type (HAL+JSON).

Note You can find the solution to this section in the book's source code on the Apress website or on GitHub at <https://github.com/Apress/pro-spring-boot-2>. The name of the project is `todo-mongo-rest`.

Spring Data Redis

Spring Data Redis provides an easy way to configure and access a Redis server. It offers low-level to high-level abstraction to interact with it, and follows the same Spring Data standard, providing a `RedisTemplate` class and repository-based persistence.

The following are some of the features of Spring Data Redis.

- `RedisTemplate` gives the high-level abstraction for all Redis operations.
- Messaging through Pub/Sub.
- Redis Sentinel and Redis Cluster support.
- Uses connection packages as a low level across multiple drivers, like Jedis and Lettuce.
- Repositories, sorting and paging by using `@EnableRedisRepositories`.
- Redis implements for Spring cache, so you can use Redis as your web cache mechanism.

Spring Data Redis with Spring Boot

If you want to use Spring Data Redis, you only need to add the `spring-boot-starter-data-redis` dependency to have access to a Redis server.

Spring Boot uses auto-configuration to set up defaults for using the Redis server. It automatically uses `@EnableRedisRepositories` (you don't need to add it) if you are using the Repository feature.

By default, use the local host and port 6379. Of course, you can override the defaults by changing the `spring.redis.*` properties in the `application.properties` file.

ToDo App with Spring Data Redis

You can create your ToDo app from scratch, or take a look at which classes you need, as well the necessary dependencies, in your `pom.xml` or `build.gradle` files.

Starting from scratch, go to your browser and open Spring Initializr. Add the following values to the fields.

- Group: `com.apress.todo`
- Artifact: `todo-redis`
- Name: `todo-redis`
- Package Name: `com.apress.todo`
- Dependencies: Web, Lombok, Redis

You can select either Maven or Gradle as the project type. Then you can press the Generate Project button, which downloads a ZIP file. Uncompress it and import the project in your favorite IDE (see Figure 5-10).

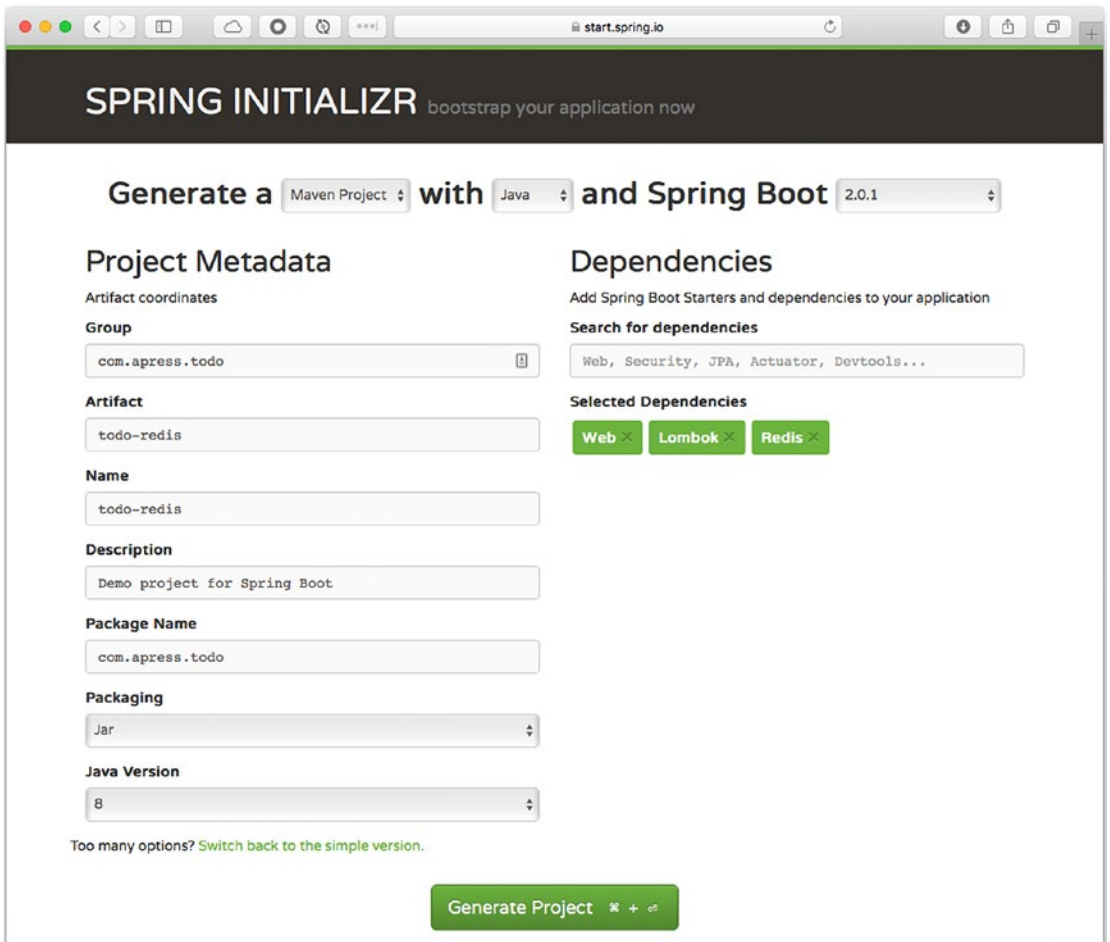


Figure 5-10. <https://start.spring.io>

You can copy all the classes from the todo-mongo project. In the next section, we see which classes need to be modified.

Domain Model: ToDo

Open the ToDo domain model class and modified accordingly (see Listing 5-10).

Listing 5-10. com.apress.todo.ToDo.java

```
package com.apress.todo.domain;

import lombok.Data;
import org.springframework.data.annotation.Id;
import org.springframework.data.redis.core.RedisHash;

import javax.validation.constraints.NotBlank;
import javax.validation.constraints.NotNull;
import java.time.LocalDateTime;
import java.util.UUID;

@Data
@RedisHash
public class ToDo {

    @NotNull
    @Id
    private String id;
    @NotNull
    @NotBlank
    private String description;
    private LocalDateTime created;
    private LocalDateTime modified;
    private boolean completed;

    public ToDo(){
        LocalDateTime date = LocalDateTime.now();
        this.id = UUID.randomUUID().toString();
        this.created = date;
        this.modified = date;
    }
}
```

```

public Todo(String description){
    this();
    this.description = description;
}
}

```

Listing 5-10 shows the only class that you modify. The class is using the `@RedisHash` annotation that marks the class as persistent and also is using the `@Id` annotation as part of a composite key. When inserting a `Todo`, there is a hash that contains a key with the format `class:id`. For this application, the composite key is something like `"com.apress.todo.domain.Todo: bbee6e32-37f3-4d5a-8f29-e2c79a28c301"`.

If you have a remote Redis server, you can override the defaults that point at the local host. You can go to your `application.properties` file and add the following properties.

```

### Redis - Remote
spring.redis.host=my-remote-server
spring.redis.port=6379
spring.redis.password=my-secret

```

You can review your `ToDoRepository` and `ToDoController` classes, which should not change at all; the same as before.

Running and Testing: ToDo App

Now it's time to run and test this `ToDo` app. You can run it within your IDE. Or if you are using Maven, execute

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

If you are using Gradle, execute

```
./gradlew bootRun
```

To test the `ToDo` app, you can run your `ToDoClient` app; and that's it. If you want to use a different structure (like `SET`, `LIST`, `STRING`, `ZSET`) a *low-level* operations you can use the `RedisTemplate` class that is already setup and configured by Spring Boot.

Note Remember that you can get the book's source code from the Apress website or on GitHub at <https://github.com/Apress/pro-spring-boot-2>.

More Data Features with Spring Boot

There are plenty more features and supported engines for manipulating data, from using a DSL with jOOQ (Java Object-Oriented Querying at www.jooq.org), which generates Java code from your database and lets you build SQL queries through its own DSL in a type-safe way.

There are ways to do database migrations, using either Flyway (<https://flywaydb.org/>) or Liquibase (<http://www.liquibase.org/>) that you can run at startup.

Multiple Data Sources

One of the important features in Spring Boot (I believed it is a must-have feature) manipulates multiple `DataSource` instances, regardless of the persistent technology used.

As you already know, Spring Boot provides a default auto-configuration based on the app's classpath, and you can override it without any problems. To use multiple `DataSource` instances, which may be pointing to different databases and/or different engines, you must override the defaults. If you remember from the Chapter 4, we created a complete but simple web app that required setup: `DataSource`, `EntityManager`, `TransactionManager`, `JpaVendor`, and so forth. Well, we need to add the same configuration if we want to use multiple datasources. In other words, we need to add multiple `EntityManagers`, `TransactionManagers`, and so forth.

How can we apply this to the `ToDo` app? Review what you did in Chapter 4. Take a close look at each configuration, and you can imagine what you need to do.

You can get the solution in the book's source code. The name of the project is `todo-rest-2ds`. This project contains all the `User` and the `ToDo` domain classes that persist data into their own databases.

Summary

In this chapter, you learned different data technologies and how they work with Spring Boot. You also learned that Spring Boot uses its auto-configuration feature to apply defaults based on the classpath.

You learned that if you have two or more SQL drivers and one of them is H2, HSQL or Derby, Spring Boot configures the embedded database engine if you haven't defined a `DataSource` instance yet. You saw how to configure and override some of the data defaults. You learned that Spring Data implements a Template pattern to hide all the complex tasks that normally do without the Spring Framework.

In the next chapter, we take the web and data up one level with reactive programming and explore the WebFlux and reactive data.

CHAPTER 6

WebFlux and Reactive Data with Spring Boot

In this chapter, I show you the newest addition to Spring Framework 5 and how to use it with Spring Boot. The new reactive stack built for web applications is Spring WebFlux, which was added in the 5.0 version of the Spring Framework. It is a fully non-blocking framework that relies on Project Reactor, which supports reactive streams back pressure and runs on servers like Netty and Undertow, and Servlet 3.1+ containers.

Before I show you how to use WebFlux with Spring Boot, let's learn about reactive systems and how Project Reactor (<https://projectreactor.io/>) implements them.

Reactive Systems

In the past decade, we have been participants in how software changed, by making it more stable, more robust, more resilient, and more flexible to accept even more modern demands, not only by users (using desktop or web), but by many devices (mobile phones, sensors, etc.). Accepting these new workloads has many challenges; that's why a group of organizations worked together to bring a manifest to cover many aspects of today's data demands.

The Reactive Manifesto

The Reactive Manifesto (<https://www.reactivemanifesto.org/>) was signed on September 16, 2014. It defines how reactive systems should be. Reactive systems are flexible, loosely coupled, and scalable. These systems are more tolerant to failure, and when a failure occurs, they deal with it by applying patterns to avoid disaster. These reactive systems have defined certain characteristics.

Reactive systems are

- *Responsive.* Most systems respond in a timely manner, if possible; they focus on providing fast and consistent response times and be reliable for delivering in consistent quality of service.
- *Resilient.* They apply replication, containment, isolation, and delegation patterns to provide resilient systems. The failures of a system must be contained through isolation; failures should not affect other systems. Recovery must be from another system, so that high availability (HA) is ensured.
- *Elastic.* The systems must be responsive in any kind of workload. Reactive systems can react to changes in input rate by increasing or decreasing the resources allocated to service these inputs. There should not be any bottlenecks, which means that the system has the ability to share or replicate components. Reactive systems must support predictive algorithms, which ensures cost-effect elasticity on commodity hardware.
- *Message driven.* Reactive systems must rely on asynchronous messaging to establish a boundary between components, making sure that the systems are loosely coupled, isolated, and location transparent. It must support load management, elasticity, and flow control by providing a back-pressure pattern when needed. The communication must be non-blocking to allow to consumer resources while active, which leads to lower system overhead.

With the Reactive Manifesto, different initiatives started to emerge and implement frameworks and libraries that help many developers around the world. Reactive Streams (www.reactive-streams.org) is a specification that defines four simple interfaces (Publisher<T>, a provider of an unbounded number sequenced elements, publishing them according to the demand of the subscriber; Subscriber<T> subscribes to the publisher; Subscription represents the one-to-one lifecycle of a subscriber subscribing to a publisher; and Processor, which is the processing stage for both Subscriber and Publisher) and different implementations, such as ReactiveX RXJava (<http://reactivex.io/>), Akka Streams (<https://akka.io/>), Ratpack (<https://ratpack.io/>), Vert.X (<https://vertx.io/>), Slick, Project Reactor, and more.

The Reactive Streams API has its own implementation in the Java 9 SDK release; in other words, as of December 2017, Reactive Streams version 1.0.2 is part of the JDK9.

Project Reactor

Project Reactor 3.x is a library that is built around the Reactive Streams specification, bringing the reactive programming paradigm to JVM. *Reactive programming* is a paradigm that is an event-based model, where data is pushed to the consumer as it becomes available; it deals with asynchronous sequences of events. Offering fully asynchronous and non-blocking patterns, reactive programming is an alternative to the limited ways of doing async code in the JDK (callbacks, APIs, and the *Future<V>* interface).

Reactor is a full, non-blocking reactive programming framework that manages back pressure and integrates interaction with the Java 8 functional APIs (*CompletableFuture*, *Stream*, and *Duration*). Reactor provides two reactive composable asynchronous APIs; *Flux [N]* (for N elements), and *Mono [0|1]* (for 0 or 1 elements). Reactor can be used for developing microservices architectures because it offers IPC (interprocess communication) with *reactor-ipc* components and *backpressure-ready* network engines for HTTP (including WebSockets, TCP, and UDP), and reactive encoding and decoding is fully supported.

Project Reactor provides processors, operators, and timers that can sustain a high throughput rate on tens of millions of messages per second with a low memory footprint.

Note If you want to know more about Project Reactor, visit <https://projectreactor.io/> and its documentation at <http://projectreactor.io/docs/core/release/reference/docs/index.html>.

ToDo App with Reactor

Let's start using Reactor in the ToDo App and experiment with the Flux and Mono APIs. In this section, create a simple Reactor example that takes care of ToDo's.

Open your favorite browser and point to Spring Initializr (<https://start.spring.io>); add the following values to the fields.

- Group: `com.apress.reactor`
- Artifact: `example`
- Dependencies: Lombok

You can select either Maven or Gradle as project type. Then press the Generate Project button, which downloads a ZIP file. Uncompress it and import the project in your favorite IDE (see Figure 6-1).

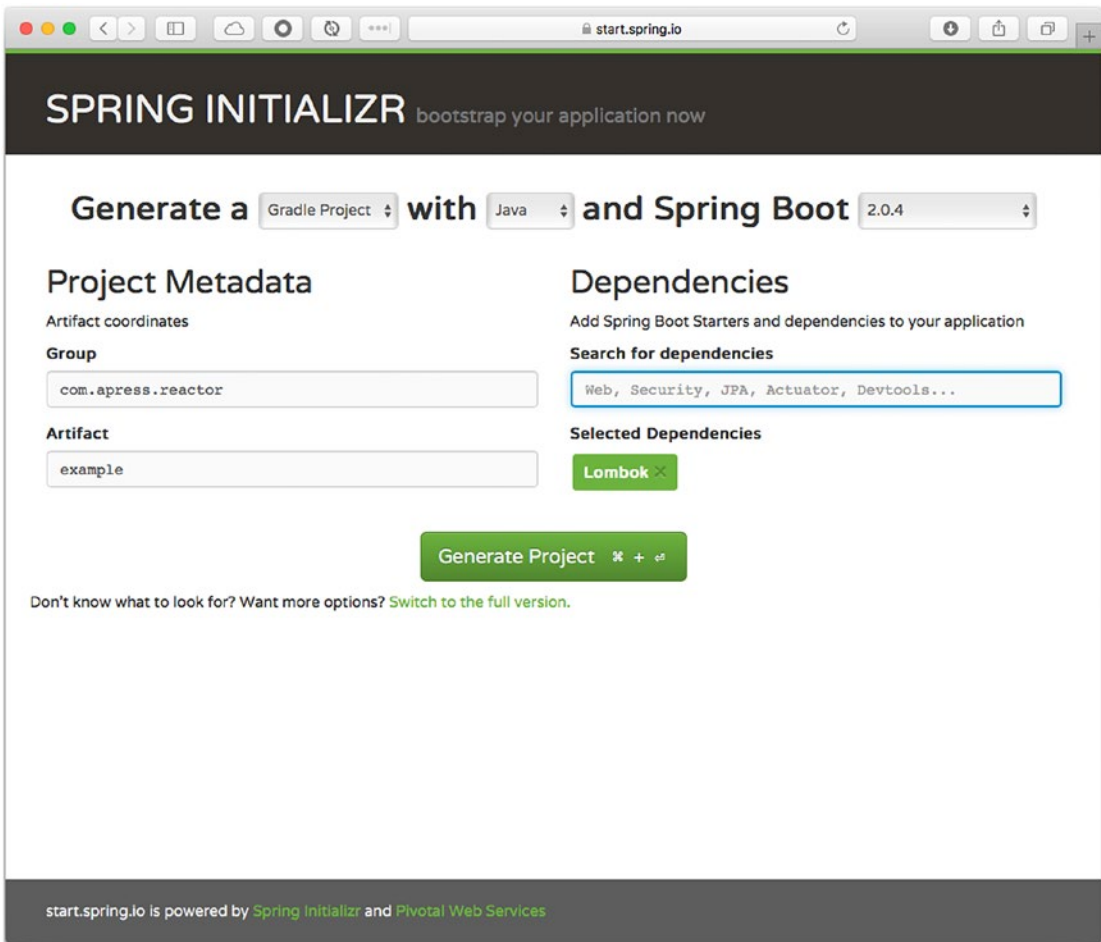


Figure 6-1. Spring Initializr <https://start.spring.io>

As you can see, there are no significant dependencies this time in Lombok (<https://projectlombok.org/>). We are going to make this as simple as possible. This is only a little taste of the Flux and Mono APIs. Later on, you create a web application with the WebFlux framework.

If you are using Maven, open your `pom.xml` and add the following section and dependency.

```

<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>io.projectreactor</groupId>
      <artifactId>reactor-bom</artifactId>
      <version>Bismuth-SR10</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
<dependencies>
  <!-- ... more dependencies here ... -->

  <dependency>
    <groupId>io.projectreactor</groupId>
    <artifactId>reactor-core</artifactId>
  </dependency>
</dependencies>

```

Reactor has a transitive dependency on the reactive-streams jars, so by adding the bill of materials (BOM), Project Reactor provides all the necessary jars.

If you are using Gradle, add the following section and dependency to your build.gradle file.

```

dependencyManagement {
  imports {
    mavenBom "io.projectreactor:reactor-bom:Bismuth-SR10"
  }
}

dependencies {
  // ... More dependencies here ...
  compile('io.projectreactor:reactor-core')
}

```

Next, let's create the `ToDo` domain class, but this time, there is no persistence to it (see Listing 6-1).

Listing 6-1. `com.apress.reactor.example.domain.ToDo.java`

```
package com.apress.reactor.example.domain;

import lombok.Data;
import java.time.LocalDateTime;

@Data
public class ToDo {

    private String id;
    private String description;
    private LocalDateTime created;
    private LocalDateTime modified;
    private boolean completed;

    public ToDo(){
    public ToDo(String description){
        this.description = description;
    }

    public ToDo(String description, boolean completed){
        this.description = description;
        this.completed = completed;
    }
}
```

Listing 6-1 shows the `ToDo` class. There is nothing special about this class, but we had been working with persistent technologies; in this case, you can leave it like that—simple. Let's start by defining the `Mono` and `Flux` reactive APIs and add the necessary code to use the `ToDo` domain class.

`Mono<T>`, an asynchronous [0|1] result

`Mono<T>` is a specialized `Publisher<T>` that emits one item and it can optionally terminate with `onComplete` or `onError` signals. You can apply operators to manipulate the item (see Figure 6-2).

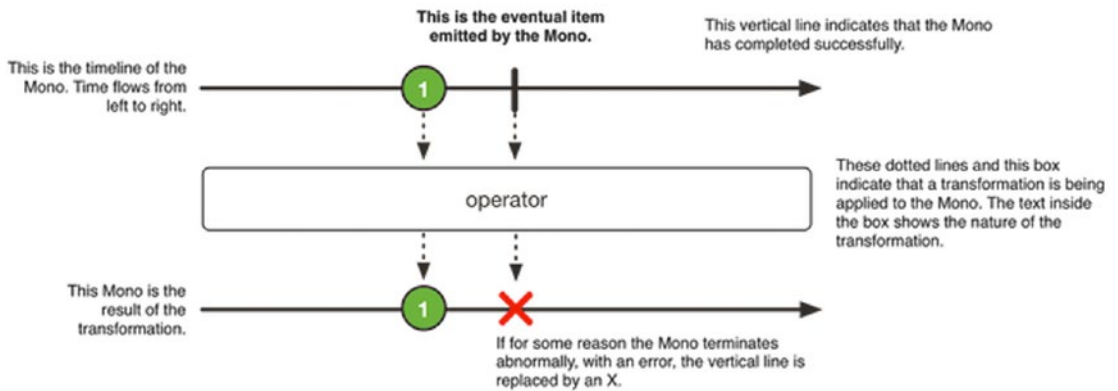


Figure 6-2. Mono [0|1] (Image taken from: <http://projectreactor.io/documentation>).

Next, let's create a `MonoExample` class and learn how to use the Mono API (see Listing 6-2).

Listing 6-2. `com.apress.reactor.example.MonoExample.java`

```
package com.apress.reactor.example;

import com.apress.reactor.example.domain.ToDo;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.CommandLineRunner;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import reactor.core.publisher.Mono;
import reactor.core.publisher.MonoProcessor;
import reactor.core.scheduler.Schedulers;

import java.time.Duration;

@Configuration
public class MonoExample {

    static private Logger LOG = LoggerFactory.getLogger(MonoExample.class);
```

```

@Bean
public CommandLineRunner runMonoExample(){
    return args -> {

        MonoProcessor<ToDo> promise = MonoProcessor.create();
        Mono<ToDo> result = promise
            .doOnSuccess(p -> LOG.info("MONO >> ToDo: {}", p.getDescription()))
            .doOnTerminate( () -> LOG.info("MONO >> Done"))
            .doOnError(t -> LOG.error(t.getMessage(), t))
            .subscribeOn(Schedulers.single());

            promise.onNext(
new ToDo("Buy my ticket for SpringOne Platform 2018"));
            //promise.onError(
new IllegalArgumentException("There is an error processing the ToDo..."));

            result.block(Duration.ofMillis(1000));
        };
    }
}

```

Listing 6-2 shows the MonoExample class; let's analyze it.

- **MonoProcessor.** In Reactor, there are processors that are a kind of publisher that are also a subscriber; this means that you can subscribe to a processor but also call methods to manually inject data into the sequence or terminate it. In this case, you are using the `onNext` method to emit a `ToDo` instance.
- **Mono.** This is a reactive stream publisher with basic operators that complete successfully by emitting an element or with an error.
- **doOnSuccess.** This method is called or triggered when the `Mono` completes successfully.
- **doOnTerminate.** This method is called or triggered when the `Mono` terminates either by completing successfully or with an error.

- `doOnError`. This method is called when the `Mono` type completes with an error.
- `subscribeOn`. Subscribes to the `Mono` type and requests unbounded demand on a specified `Scheduler` worker.
- `onNext`. This method emits a value that can be marked as `@Nullable`.
- `block`. Subscribes to the `Mono` type and block until a next signal is received or a timeout expires.

As you can see a very simple example but remember that now we are talking about reactive programming, where there is no more blocking or polling from the server, but pushing to the consumer until it sends back a completed signal. This make more efficient and more robust our apps. We can say that we never can have slow consumers anymore.

You can now run the application either using command line or using your IDE. You should see the following output:

```
INFO 55588 - [single-1] c.a.r.e.MonoExample : MONO >> ToDo: Buy my ticket
for SpringOne Platform 2018
INFO 55588 - [single-1] c.a.r.e.MonoExample : MONO >> Done
```

Note How did this code run? remember that we had marked the class as `@Configuration` and we had declared a `@Bean` that returns a `CommandLineRunner` interface. Spring Boot execute this bean after it finish wiring up all the Spring beans and before the app starts; so it's a nice way to execute code (like an initialization) before the app runs.

Flux<T>: An asynchronous Sequence of [0..N] Items

Flux is a `Publisher<T>` that represents an asynchronous sequence of 0 to N emitted items that can optionally terminate by using `onComplete` or an `onError` signals (see Figure 6-3).

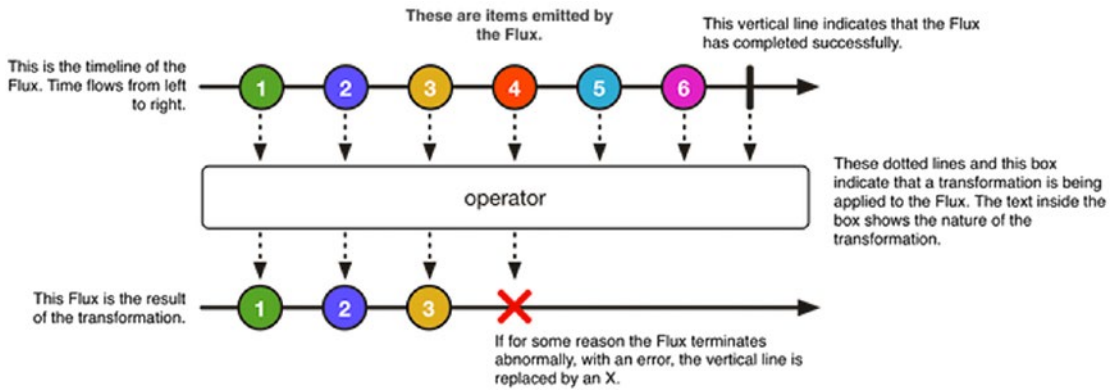


Figure 6-3. Flux [0|N] (Image taken from: <http://projectreactor.io/documentation>).

Create the FluxExample class (see Listing 6-3).

Listing 6-3. com.apress.reactor.example.FluxExample.java

```
package com.apress.reactor.example;

import com.apress.reactor.example.domain.ToDo;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.CommandLineRunner;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import reactor.core.publisher.EmitterProcessor;
import reactor.core.publisher.Mono;
import reactor.core.scheduler.Schedulers;

import java.util.List;

@Configuration
public class FluxExample {

    static private Logger LOG = LoggerFactory.getLogger(FluxExample.class);

    @Bean
    public CommandLineRunner runFluxExample(){
        return args -> {
```

```

EmitterProcessor<ToDo> stream =
    EmitterProcessor.create();

Mono<List<ToDo>> promise = stream
    .filter( s -> s.isCompleted())
    .doOnNext(s -> LOG.info("FLUX >>> ToDo: {}"),
    s.getDescription()))
    .collectList()
    .subscribeOn(Schedulers.single());

stream.onNext(new ToDo("Read a Book",true));
stream.onNext(new ToDo("Listen Classical Music",true));
stream.onNext(new ToDo("Workout in the Mornings"));
stream.onNext(new ToDo("Organize my room", true));
stream.onNext(new ToDo("Go to the Car Wash", true));
stream.onNext(new ToDo("SP1 2018 is coming" , true));

    stream.onComplete();

    promise.block();

};
}
}

```

Listing 6-3 shows the FluxExample class. Let's analyze it.

- **EmitterProcessor.** Remember that a processor is a kind of publisher; in this case, we are using a synchronous processor that can push data both through user action and by subscribing to an upstream publisher and synchronously draining it. This processor creates a flux of ToDo instances; it provided an implementation of a RingBuffer backed *messaging-passing* processor implementing publish-subscribe with synchronous drain loops. If you want to use asynchronous processor, you can use `WorkQueueProcessor` or `aTopicProcessor`.
- **filter.** Remember that you can apply operators to Flux and Mono APIs; in this case, a filter is applied using a predicate that evaluates and emits a value if succeeded (i.e., if the ToDo is completed).

- `doOnNext`. Triggers when a flux emits an item.
- `collectList`. Collects all elements emitted by a flux into a list that is emitted by the resulting mono when this sequence completes.
- `subscribeOn`. Subscribes to this flux based on a scheduler worker.
- `onNext`. Emits a new value to the flux.
- `onComplete`. Finishes the upstream.
- `block`. Subscribes to the Mono type and blocks until a next signal is received or a timeout expires.

Now, you can run the code. You should have something similar to the following output.

```
INFO 61 - [single-1] c.a.r.e.FluxExample : FLUX >>> ToDo: Read a Book
INFO 61 - [single-1] c.a.r.e.FluxExample : FLUX >>> ToDo: Listen Classical
                    Music
INFO 61 - [single-1] c.a.r.e.FluxExample : FLUX >>> ToDo: Organize my room
INFO 61 - [single-1] c.a.r.e.FluxExample : FLUX >>> ToDo: Go to the Car Wash
INFO 61 - [single-1] c.a.r.e.FluxExample : FLUX >>> ToDo: SP1 2018 is coming
```

Again, this is a simple example with the ToDo app. Imagine that you have millions of users accessing your app to post ToDos per account, and you want to keep track of every single one of them—the same as a Twitter feed—and you don't want to block any users. You should be reactive!

WebFlux

Spring MVC has been the primary way to create web applications with the Spring Framework for a long time. Now another player has come into the picture—a reactive stack, the Spring WebFlux framework! Spring WebFlux is a fully asynchronous and non-blocking framework that relies on the Reactive Streams implementations by Project Reactor.

One of the main features of Spring WebFlux is that it provides two programming models.

- *Annotated controllers.* Consistent with the Spring MVC and based on the same annotations from the spring-web module; this means that you can use the same known annotations (`@Controller`, `@Mapping`, `@RequestBody`, etc.) but with all the reactive support from Reactor and RxJava.

```
@RestController
public class TodoController {

    @GetMapping("/todo/{id}")
    public Mono<ToDo> getToDo(@PathVariable Long id) {
        // ...
    }

    @GetMapping("/todo")
    public Flux<ToDo> getTodos() {
        // ...
    }
}
```

- *Functional endpoints.* A functional programming model, where you can use lambda-based calls. You need to provide the route endpoints by declaring RouterFunction beans and the endpoint handler that returns the response with a Mono or a Flux type.

```
@Configuration
public class TodoRoutingConfiguration {

    @Bean
    public RouterFunction<ServerResponse>
        monoRouterFunction(TodoHandler todoHandler) {
        return
            route(GET("/todo/{id}")
                .and(accept(APPLICATION_JSON)), todoHandler::getToDo)
                .andRoute(GET("/todo")
                    .and(accept(APPLICATION_JSON)), todoHandler::getTodos);
    }
}
```

```

@Component
public class ToDoHandler {

    public Mono<ServerResponse> getToDo(ServerRequest request){
        // ...
    }

    public Mono<ServerResponse> getTodos(ServerRequest request){
        // ...
    }
}

```

Filters, exceptions handlers, CORS, view technologies, and web security are handled in the same way as Spring MVC. That's the beauty of using Spring Framework— a consistent ecosystem regardless of new technology.

WebClient

The WebClient module also introduces a reactive, non-blocking client for HTTP requests with a functional-style API client and Reactive Streams support. The WebClient interface has the following characteristics.

- a functional API that takes advantage of the lambda programming style
- non-blocking and reactive
- supports high concurrency with fewer hardware resources
- supports streaming up and down from a server
- supports both synchronous and asynchronous communication

The WebClient is very easy to use. This client has the retrieve or exchange methods to get a response body and decode it.

```

WebClient client = WebClient.create("http://my-to-dos.com");

// [0|1] ToDo
Mono<ToDo> result = client
    .get()

```

```

        .uri("/todo/{id}", id)
        .accept(MediaType.APPLICATION_JSON)
        .retrieve()
        .bodyToMono(ToDo.class);

//[0|N] Todos
Flux<ToDo> result = client
    .get()
    .uri("/todo").accept(MediaType.TEXT_EVENT_STREAM)
    .retrieve()
    .bodyToFlux(ToDo.class);

```

Later, we are going to create a small example for using this client, but if you are interested in learning more about it, check out <https://docs.spring.io/spring/docs/current/spring-framework-reference/web-reactive.html#webflux-client>.

WebFlux and Spring Boot Auto-configuration

With Spring Boot, Spring WebFlux is easier than ever, because Spring Boot offers *auto-configuration* by configuring the necessary codecs for `HttpMessageReader` and `HttpMessageWriter` instances. It has the support for serving static resources, including the support for WebJars. It uses the latest template engine technologies that support WebFlux, such as FreeMarker (<https://freemarker.apache.org/>), Thymeleaf (www.thymeleaf.org), and Mustache (<https://mustache.github.io/>). By default, the auto-configuration set up Netty (<https://netty.io>) as the primary container.

If you need to override the default WebFlux auto-configuration, you can add your own `@Configuration` class of type `WebFluxConfigurer`.

Important Note If you want to have full control over the WebFlux auto-configuration, then you need to add your custom `@Configuration` annotated with `@EnableWebFlux`.

Using WebFlux with Spring Boot

To use WebFlux with Spring Boot, it is necessary to add the `spring-boot-starter-webflux` dependency to your `pom.xml` or `build.gradle` file.

Important Note You can use both `spring-boot-starter-web` and `spring-boot-starter-webflux`, but the Spring Boot auto-configure is the Spring MVC. If you want to use all the WebFlux features, then you need to use `SpringApplication.setWebApplicationType(WebApplicationType.REACTIVE)`

ToDo App with WebFlux

Let's start by creating the ToDo app using the WebFlux module. Let's use the new reactive Flux and Mono APIs.

Open your favorite browser and point to Spring Initializr. Add the following values to the fields.

- Group: `com.apress.todo`
- Artifact: `todo-webflux`
- Name: `todo-webflux`
- Package Name: `com.apress.todo`
- Dependencies: Lombok, Reactive Web

You can select either Maven or Gradle as the project type. Then you can press the Generate Project button to download a ZIP file. Uncompress it and import the project in your favorite IDE (see Figure 6-4).

The screenshot shows the Spring Initializr web interface. At the top, it says "SPRING INITIALIZR bootstrap your application now". Below that, there's a header "Generate a Maven Project with Java and Spring Boot 2.0.4". The interface is divided into two main sections: "Project Metadata" and "Dependencies".

Project Metadata:

- Artifact coordinates:** Group: ; Artifact: ; Name: ; Description: ; Package Name: ; Packaging: ; Java Version: .

Dependencies:

- Search for dependencies:**
- Selected Dependencies:** Lombok, Reactive Web.

At the bottom, there's a green button labeled "Generate Project". A small note says "Too many options? Switch back to the simple version."

Figure 6-4. Spring Initializr <https://start.spring.io>

This time you use the Reactive Web dependency; in this case, the `spring-boot-starter-webflux` starter. Let's create the `ToDo` domain class, which is similar to other projects (see Listing 6-4).

Listing 6-4. `com.apress.todo.domain.ToDo.java`

```
package com.apress.todo.domain;

import lombok.Data;

import java.time.LocalDateTime;
import java.util.UUID;
```

```

@Data
public class ToDo {
    private String id;
    private String description;
    private LocalDateTime created;
    private LocalDateTime modified;
    private boolean completed;

    public ToDo(){
        this.id = UUID.randomUUID().toString();
        this.created = LocalDateTime.now();
        this.modified = LocalDateTime.now();
    }

    public ToDo(String description){
        this();
        this.description = description;
    }

    public ToDo(String description, boolean completed){
        this();
        this.description = description;
        this.completed = completed;
    }
}

```

Listing 6-4 shows the `ToDo` class; as you can see nothing new, except for initialization in the default constructor.

Next, let's create the `ToDoRepository` class, which holds in-memory `ToDo`'s (see Listing 6-5).

Listing 6-5. `com.apress.todo.repository.ToDoRepository.java`

```

package com.apress.todo.repository;

import com.apress.todo.domain.ToDo;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

import java.util.Arrays;

```

```

public class ToDoRepository {
    private Flux<ToDo> todoFlux =
Flux.fromIterable(Arrays.asList(
        new ToDo("Do homework"),
        new ToDo("Workout in the mornings", true),
        new ToDo("Make dinner tonight"),
        new ToDo("Clean the studio", true)));

    public Mono<ToDo> findById(String id){
        return Mono
            .from(
                todoFlux.filter( todo -> todo.getId().equals(id)));
    }

    public Flux<ToDo> findAll(){
        return todoFlux;
    }
}

```

Listing 6-5 shows the `ToDoRepository` class, where the `todoFlux` instance handles a `Flux<ToDo>`. Take a look at the `findById` method that returns a `Mono<ToDo>` from the flux.

Using Annotated Controllers

Let's continue by creating something that you already know about (similar to Spring MVC): a `ToDoController` class that handles `Flux` and `Mono` types (see Listing 6-6).

Listing 6-6. `com.apress.todo.controller.ToDoController.java`

```

package com.apress.todo.controller;

import com.apress.todo.domain.ToDo;
import com.apress.todo.repository.ToDoRepository;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

```

```

@RestController
public class TodoController {

    private TodoRepository repository;

    public TodoController(TodoRepository repository){
        this.repository = repository;
    }

    @GetMapping("/todo/{id}")
    public Mono<Todo> getTodo(@PathVariable String id){
        return this.repository.findById(id);
    }

    @GetMapping("/todo")
    public Flux<Todo> getTodos(){
        return this.repository.findAll();
    }
}

```

As you can see from the code, you are returning a `Mono<Todo>` or a `Flux<Todo>`, which is different from Spring MVC; remember that we are doing async and non-blocking calls.

You can run the application and go to a browser and point to `http://localhost:8080/todo` to see the results—a JSON response of the `Todo`'s; or you can execute the following command in a terminal and see the same output.

```

$ curl http://localhost:8080/todo
[
  {
    "completed": false,
    "created": "2018-08-14T20:46:05.542",
    "description": "Do homework",
    "id": "5520e646-47aa-4be6-802a-ef6df500d6fb",
    "modified": "2018-08-14T20:46:05.542"
  },

```

```

{
    "completed": true,
    "created": "2018-08-14T20:46:05.542",
    "description": "Workout in the mornings",
    "id": "3fe07f8d-64b0-4a39-ab1b-658bde4815d7",
    "modified": "2018-08-14T20:46:05.542"
},
...
]

```

In the console logs, note that the running container is the Netty container—the default from the Spring Boot auto-configuration.

Using Functional Endpoints

Remember that Spring WebFlux use functional programming to create reactive web apps. Let's create the `ToDoRouter` class that declares the routing functions(see Listing 6-7).

Listing 6-7. `com.apress.todo.reactive.ToDoRouter.java`

```

package com.apress.todo.reactive;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.reactive.function.server.RouterFunction;
import org.springframework.web.reactive.function.server.ServerResponse;

import static org.springframework.http.MediaType.APPLICATION_JSON;
import static org.springframework.web.reactive.function.server.
RequestPredicates.GET;
import static org.springframework.web.reactive.function.server.
RequestPredicates.accept;
import static org.springframework.web.reactive.function.server.
RouterFunctions.route;

@Configuration
public class ToDoRouter {

```

```

@Bean
public RouterFunction<ServerResponse> monoRouterFunction(ToDoHandler
todoHandler) {
    return route(GET("/todo/{id}").and(accept(APPLICATION_JSON)),
        todoHandler::getToDo)
        .andRoute(GET("/todo").and(accept(APPLICATION_JSON)),
            todoHandler::getTodos);
}
}

```

Listing 6-7 shows the routes that are used (endpoints) and the handler. Spring WebFlux offers a very nice fluent API to easily build any route.

Next, let's create the `ToDoHandler` class that has the logic to server the requests (see Listing 6-8).

Listing 6-8. `com.apress.todo.reactive.ToDoHandler.java`

```

package com.apress.todo.reactive;

import com.apress.todo.domain.ToDo;
import com.apress.todo.repository.ToDoRepository;
import org.springframework.stereotype.Component;
import org.springframework.web.reactive.function.server.ServerRequest;
import org.springframework.web.reactive.function.server.ServerResponse;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

import static org.springframework.http.MediaType.APPLICATION_JSON;
import static org.springframework.web.reactive.function.BodyInserters.
fromObject;

@Component
public class ToDoHandler {

    private ToDoRepository repository;

    public ToDoHandler(ToDoRepository repository){
        this.repository = repository;
    }
}

```

```

public Mono<ServerResponse> getToDo(ServerRequest request) {
    String todoId = request.pathVariable("id");
    Mono<ServerResponse> notFound =
ServerResponse.notFound().build();
    Mono<ToDo> todo = this.repository.findById(todoId);
    return todo
        .flatMap(t ->
            ServerResponse
                .ok()
                .contentType(APPLICATION_JSON)
                .body(fromObject(t)))
        .switchIfEmpty(notFound);
}

public Mono<ServerResponse> getTodos(
    ServerRequest request) {
    Flux<ToDo> todos = this.repository.findAll();
    return ServerResponse
        .ok()
        .contentType(APPLICATION_JSON)
        .body(todos, ToDo.class);
}
}

```

Listing 6-8 shows you the handler. Let's analyze it.

- `Mono<ServerResponse>`. This response type is used on both methods, and it uses the `ServerResponse` interface with a `BodyBuilder` fluent API that adds a body to the response. The `BodyBuilder` interface provides useful methods that can help you build the response, such as adding status with the `ok` method. You can add headers with the `headers` method, and so forth.

Before you run the application, it's important to remove the `ToDoController` class from the Spring container. You can do this by commenting out the `@RestController` annotation.

Running the application, you get the same results as before. I know that this is a very simple example, because this app is doing everything in-memory, right? Well, let's add reactive persistence!

Reactive Data

The Spring Data team created reactive repositories with dynamic APIs implemented using RxJava and Project Reactor. This abstraction defines several wrapper types. Spring Data converts reactive wrapper types behind the scenes so that you can stick to your favorite composition library, making it easier for the developer.

- `ReactiveCrudRepository`
- `ReactiveSortingRepository`
- `RxJava2CrudRepository`
- `RxJava2SortingRepository`

Spring Data offers different reactive modules: MongoDB, Redis, and Cassandra Reactive Streams, giving you all the flexibility and all the goodies from the Reactive Streams initiative.

MongoDB Reactive Streams

Spring Data MongoDB is built on top of MongoDB Reactive Streams, and it provides the maximum interoperability of Reactive Streams. It provides the `ReactiveMongoOperations` helper interface to use the Flux and Mono types.

To use MongoDB Reactive Streams it is necessary to include the `spring-boot-starter-data-mongodb-reactive` dependency to your `pom.xml` or `build.gradle` file. MongoDB Reactive Streams also provide a dedicated interface for a repository declaration, the `ReactiveMongoRepository<T, ID>` interface. Following the same repository pattern, you can declare your own *query named method* that returns a Flux or Mono types.

ToDo App with Reactive Data

Let's complete the ToDo app by adding reactive data with MongoDB and still use WebFlux for any requests and responses, using the functional programming mode.

You can open your favorite browser and point to Spring Initializr(<https://start.spring.io>); add the following values to the fields.

- Group: `com.apress.todo`
- Artifact: `todo-reactive-data`

- Name: todo-reactive-data
- Package Name: com.apress.todo
- Dependencies: Lombok, Reactive Web, Reactive MongoDB

You can select either Maven or Gradle as the project type. Then you can press the Generate Project button to download the ZIP file. Uncompress it and import the project in your favorite IDE (see Figure 6-5).

The screenshot shows the Spring Initializr web application interface. At the top, it says "SPRING INITIALIZR bootstrap your application now". Below that, there's a heading "Generate a Maven Project with Java and Spring Boot 2.0.4". The interface is divided into two main sections: "Project Metadata" and "Dependencies".

Project Metadata:

- Artifact coordinates:**
 - Group:** com.apress.todo
 - Artifact:** todo-reactive-data
 - Name:** todo-reactive-data
 - Description:** Demo project for Spring Boot
 - Package Name:** com.apress.todo
 - Packaging:** Jar
 - Java Version:** 8

Dependencies:

- Search for dependencies:** Web, Security, JPA, Actuator, Devtools...
- Selected Dependencies:** Lombok, Reactive Web, Reactive MongoDB

At the bottom, there's a green button labeled "Generate Project". A note at the bottom left says "Too many options? Switch back to the simple version."

Figure 6-5. Spring Initializr <https://start.spring.io>

As you can see, we now have the Reactive Web and the Reactive MongoDB dependencies. Because we are using MongoDB, there is no need to have a server. You use an embedded Mongo; normally, it is for testing but we are going to use it for this app.

Let's start by adding the embedded Mongo dependency. If you are using Maven, open your `pom.xml` file and add the following dependency.

```
<dependency>
  <groupId>de.flapdoodle.embed</groupId>
  <artifactId>de.flapdoodle.embed.mongo</artifactId>
  <scope>runtime</scope>
</dependency>
```

If you are using Gradle, you can open the `build.gradle` file and add the following dependency.

```
runtime('de.flapdoodle.embed:de.flapdoodle.embed.mongo')
```

Next, let's create the `ToDo` domain class (see Listing 6-9).

Listing 6-9. `com.apress.todo.domain.ToDo.java`

```
package com.apress.todo.domain;

import lombok.Data;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

import java.time.LocalDateTime;
import java.util.UUID;

@Document
@Data
public class ToDo {

    @Id
    private String id;
    private String description;
    private LocalDateTime created;
    private LocalDateTime modified;
    private boolean completed;
```

```

public Todo(){
    this.id = UUID.randomUUID().toString();
    this.created = LocalDateTime.now();
    this.modified = LocalDateTime.now();
}

public Todo(String description){
    this();
    this.description = description;
}

public Todo(String description, boolean completed){
    this();
    this.description = description;
    this.completed = completed;
}
}

```

This class is using the `@Document` and `@Id` annotation that marks them as a persistent class for MongoDB.

Next, let's create the `ToDoRepository` interface (see Listing 6-10).

Listing 6-10. `com.apress.todo.repository.ToDoRepository.java`

```

package com.apress.todo.repository;

import com.apress.todo.domain.ToDo;
import org.springframework.data.mongodb.repository.ReactiveMongoRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface ToDoRepository extends
    ReactiveMongoRepository<ToDo, String> {
}

```

This interface extends from `ReactiveMongoRepository<T, ID>`. The interface provides the same `Repository` functionality that you already know, but now it returns `Flux` and `Mono` types. Remember that is the same as previous chapters. Here you define your custom named queries that return the reactive types.

For this `ToDo` app, we are going to use functional programming. Let's create the router and handler in the `ToDoRouter` class (see Listing 6-11).

Listing 6-11. `com.apress.todo.reactive.ToDoRouter.java`

```
package com.apress.todo.reactive;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.reactive.function.server.RouterFunction;
import org.springframework.web.reactive.function.server.ServerResponse;

import static org.springframework.http.MediaType.APPLICATION_JSON;
import static org.springframework.web.reactive.function.server.
RequestPredicates.*;
import static org.springframework.web.reactive.function.server.
RouterFunctions.route;

@Configuration
public class ToDoRouter {

    @Bean
    public RouterFunction<ServerResponse>
        monoRouterFunction(ToDoHandler toDoHandler) {
        return
            route(GET("/todo/{id}").and(accept(APPLICATION_JSON)),
                toDoHandler::getToDo)

        .andRoute(GET("/todo").and(accept(APPLICATION_JSON)),
            toDoHandler::getTodos)

        .andRoute(POST("/todo").and(accept(APPLICATION_JSON)),
            toDoHandler::newToDo);
    }
}
```

Listing 6-11 shows the endpoint declarations. There is a new method, a `POST`. Next, let's create the handler (see Listing 6-12).

Listing 6-12. com.apress.todo.reactive.ToDoHandler.java

```

package com.apress.todo.reactive;

import com.apress.todo.domain.ToDo;
import com.apress.todo.repository.ToDoRepository;
import org.springframework.stereotype.Component;
import org.springframework.web.reactive.function.server.ServerRequest;
import org.springframework.web.reactive.function.server.ServerResponse;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

import static org.springframework.http.MediaType.APPLICATION_JSON;
import static org.springframework.web.reactive.function.BodyInserters.
    fromObject;
import static org.springframework.web.reactive.function.BodyInserters.
    fromPublisher;
@Component
public class ToDoHandler {

    private ToDoRepository repository;

    public ToDoHandler(ToDoRepository repository){
        this.repository = repository;
    }

    public Mono<ServerResponse> getToDo(ServerRequest request) {
        return findById(request.pathVariable("id"));
    }

    public Mono<ServerResponse> getTodos(ServerRequest request) {
        Flux<ToDo> todos = this.repository.findAll();
        return ServerResponse
            .ok()
            .contentType(APPLICATION_JSON)
            .body(todos, ToDo.class);
    }
}

```

```

public Mono<ServerResponse> newToDo(ServerRequest request) {
    Mono<ToDo> toDo = request.bodyToMono(ToDo.class);

    return ServerResponse
        .ok()
        .contentType(APPLICATION_JSON)
        .body(fromPublisher(toDo.flatMap(this::save),ToDo.class));
}

private Mono<ServerResponse> findById(String id){
    Mono<ToDo> toDo = this.repository.findById(id);

    Mono<ServerResponse> notFound = ServerResponse.notFound().build();

    return toDo
        .flatMap(t -> ServerResponse
            .ok()
            .contentType(APPLICATION_JSON)
            .body(fromObject(t)))
        .switchIfEmpty(notFound);
}

private Mono<ToDo> save(ToDo toDo) {
    return Mono.fromSupplier(
        () -> {
            repository
                .save(toDo)
                .subscribe();
            return toDo;
        });
}
}

```

Listing 6-12 shows the handler that takes care of the response to the endpoints. Every method returns a `Mono<ServerResponse>`. Take a look at the `Mono` operators that make it possible save to the MongoDB server, and how the `BodyBuilder` creates the response.

Next, create the configuration that sets up the connection to the embedded MongoDB server. Create the `ToDoConfig` class (see Listing 6-13).

Listing 6-13. `com.apress.todo.config.ToDoConfig.java`

```

package com.apress.todo.config;

import com.apress.todo.domain.ToDo;
import com.apress.todo.repository.ToDoRepository;
import com.mongodb.reactivestreams.client.MongoClient;
import com.mongodb.reactivestreams.client.MongoClients;
import org.springframework.boot.CommandLineRunner;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.DependsOn;
import org.springframework.core.annotation.Order;
import org.springframework.core.env.Environment;
import org.springframework.data.mongodb.config.
AbstractReactiveMongoConfiguration;
import org.springframework.data.mongodb.repository.config.
EnableReactiveMongoRepositories;

import java.util.Arrays;
import java.util.Optional;
import java.util.stream.Collectors;

@Configuration
@EnableReactiveMongoRepositories(basePackages = "com.apress.todo.repository")
public class ToDoConfig extends AbstractReactiveMongoConfiguration {

    private final Environment environment;

    public ToDoConfig(Environment environment){
        this.environment = environment;
    }

    @Override
    protected String getDatabaseName() {
        return "todos";
    }
}

```

```

@Override
@Bean
@DependsOn("embeddedMongoServer")
public MongoClient reactiveMongoClient() {
    int port = environment.getProperty("local.mongo.port", Integer.class);
    return MongoClients.create(String.format("mongodb://localhost:%d",
        port));
}

@Bean
public CommandLineRunner insertAndView(TodoRepository repository,
ApplicationContext context){
    return args -> {

        repository.save(new Todo("Do homework")).subscribe();
        repository.save(new Todo("Workout in the mornings", true)).
            subscribe();
        repository.save(new Todo("Make dinner tonight")).subscribe();
        repository.save(new Todo("Clean the studio", true)).subscribe();

        repository.findAll().subscribe(System.out::println);
    };
}
}

```

Listing 6-13 shows the configuration required to use the MongoDB Reactive Stream embedded server. Let's analyze this class.

- `@EnableReactiveMongoRepositories`. This annotation is required to set up all the necessary infrastructure for the reactive stream APIs. It's also important to tell this annotation where the repositories are (this is not necessary if the repos are part of the package).
- `AbstractReactiveMongoConfiguration`. To set up the embedded Mongo, it is necessary to extend from this abstract class and implement the `reactiveMongoClient` and the `getDatabaseName` methods. `reactiveMongoClient` creates the `MongoClient` instance that connects to wherever the port of the embedded MongoDB is set (thanks to the environment property `local.mongo.port`).

- `@DependsOn`. This annotation is a helper that creates `reactiveMongoClient` after the `embeddedMongoServer` bean.

The class is also running code where the `ToDo`'s are saved to the MongoDB.

Now, you are all set to run the `ToDo` app. After running the app, you can go to the browser or execute the following command.

```
$ curl http://localhost:8080/todo
[
  {
    "completed": false,
    "created": "2018-08-14T20:46:05.542",
    "description": "Do homework",
    "id": "5520e646-47aa-4be6-802a-ef6df500d6fb",
    "modified": "2018-08-14T20:46:05.542"
  },
  {
    "completed": true,
    "created": "2018-08-14T20:46:05.542",
    "description": "Workout in the mornings",
    "id": "3fe07f8d-64b0-4a39-ab1b-658bde4815d7",
    "modified": "2018-08-14T20:46:05.542"
  },
  ...
]
```

You can add a new `ToDo` by executing the following.

```
$ curl -i -X POST -H "Content-Type: application/json" -d '{
"description":"Read a book"}' http://localhost:8080/todo
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 164
{
"id":"a3133b8d-1d8b-4b2e-b7d9-48a73999a104",
"description":"Read a book",
"created":"2018-08-14T22:51:19.734",
```

```
"modified": "2018-08-14T22:51:19.734",  
"completed": false  
}
```

Congratulations! Now, you know how to create a reactive, async, and a non-blocking application using the power of Spring WebFlux and Project Reactor!

Note All the code is available from the Apress website. You can also get the latest at <https://github.com/felipeg48/pro-spring-boot-2nd> repository.

Summary

This chapter discussed how to use WebFlux, the new addition to the Spring Framework. You learned that WebFlux is a non-blocking, asynchronous, and reactive framework that implements the Reactive Streams.

You also learned that WebFlux provides two ways for programming with Spring Boot: annotated controllers and functional endpoints, where you can define Flux and Mono response types. You learned that WebFlux also provides a `WebClient` interface for using these new reactive APIs.

You learned that Spring Boot provides an auto-configuration for the reactive stack by using the `spring-boot-starter-webflux`, using Netty by default as a server container.

The next chapter tests the apps and shows how Spring Boot makes it easier to apply TDD.

CHAPTER 7

Testing with Spring Boot

Software testing is a process of executing a program or system to find errors or defects (normally call bugs) and make sure every program or system really works.

This is one of the many definitions that you can find on the Internet.

Many companies put every effort into finding the right and easy way to do testing by creating amazing frameworks, a practice called TDD (test-driven development).

TDD is a process that is based on repetition in a very short development cycle; here, feedback plays an important part of the process, because the developer writes code to pass a use case with a minimum amount of code. And thanks to the feedback, the code can be refactored until it passed and acceptable for the final user.

Spring Testing Framework

One of the main ideas of the Spring Framework is to encourage developers to create simple and loosely coupled classes, program to interfaces, making the software more robust and extensible. The Spring Framework provides the tools for making unit and integration testing easy (actually, you don't need Spring to test the functionality of your system if you really program interfaces); in other words, your application should be testable using the JUnit or TestNG test engines with objects (by simple instantiated using the new operator)—without Spring or any other container.

The Spring Framework has several testing packages that help create unit and integration testing for applications. It offers unit testing by providing several mock objects (Environment, PropertySource, JNDI, Servlet; Reactive: ServerHttpRequest and ServerHttpResponse test utilities) that help test your code in isolation.

One of the most commonly used testing features of the Spring Framework is integration testing. Its primary's goals are

- managing the Spring IoC container caching between test execution
- transaction management

- dependency injection of test fixture instances
- Spring-specific base classes

The Spring Framework provides an easy way to do testing by integrating the `ApplicationContext` in the tests. The Spring testing module offers several ways to use the `ApplicationContext`, programmatically and through annotations:

- `BootstrapWith`. A class-level annotation to configure how the Spring TestContext Framework is bootstrapped.
- `@ContextConfiguration`. Defines class-level metadata to determine how to load and configure an `ApplicationContext` for integration tests. This is a must-have annotation for your classes, because that's where the `ApplicationContext` loads all your bean definitions.
- `@WebAppConfiguration`. A class-level annotation to declare that the `ApplicationContext` loads for an integration test should be a `WebApplicationContext`.
- `@ActiveProfile`. A class-level annotation to declare which bean definition profile(s) should be active when loading an `ApplicationContext` for an integration test.
- `@TestPropertySource`. A class-level annotation to configure the locations of properties files and inline properties to be added to the set of `PropertySources` in the `Environment` for an `ApplicationContext` loaded for an integration test.
- `@DirtiesContext`. Indicates that the underlying Spring `ApplicationContext` has been dirtied during the execution of a test (modified or corrupted for example, by changing the state of a singleton bean) and should be closed.

There a lot more (`@TestExecutionListeners`, `@Commit`, `@Rollback`, `@BeforeTransaction`, `@AfterTransaction`, `@Sql`, `@SqlGroup`, `@SqlConfig`, `@Timed`, `@Repeat`, `@IfProfileValue`, and so forth).

As you can see, there are a lot of choices when you test with the Spring Framework. Normally, you always use the `@RunWith` annotation that wires up all the test framework goodies. For example,

```

@RunWith(SpringRunner.class)
@ContextConfiguration({"app-config.xml", "test-data-access-config.xml"})
@ActiveProfiles("dev")
@Transactional
public class ToDoRepositoryTests {

    @Test
    public void ToDoPersistenceTest(){
        //...
    }
}

```

Now, let's see how the Spring Test Framework is used and all the features that Spring Boot offers.

Spring Boot Testing Framework

Spring Boot uses the power of the Spring Testing Framework by enhancing and adding new annotations and features that make testing easier for developers.

If you want to start using all the testing features by Spring Boot, you only need to add the `spring-boot-starter-test` dependency with scope `test` to your application. This dependency is already in place in the Spring Initializr service.

The `spring-boot-starter-test` dependency provides several test frameworks that play along very well with all the Spring Boot testing features: JUnit, AssertJ, Hamcrest, Mockito, JSONassert, and JsonPath. Of course, there are other test frameworks that play very nicely with the Spring Boot Test module; you only need to include those dependencies manually.

Spring Boot provides the `@SpringBootTest` annotation that simplifies the way you can test Spring apps. Normally, with Spring testing, you are required to add several annotations to test a particular feature or functionality of your app, but not in Spring Boot—although you are still required to use the `@RunWith(SpringRunner.class)` annotation to do your testing; if you do not, any new Spring Boot test annotation will be ignored. The `@SpringBootTest` has parameters that are useful when testing a web app, such as defining a `RANDOM_PORT` or `DEFINED_PORT`.

The following code snippet is the skeleton of what a Spring Boot test looks like.

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class MyTests {

    @Test
    public void exampleTest() {
        ...
    }
}
```

Testing Web Endpoints

Spring Boot offers a way to test endpoints: a mocking environment called the `MockMvc` class.

```
@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureMockMvc
public class MockMvcToDoTests {

    @Autowired
    private MockMvc mvc;

    @Test
    public void toDoTest() throws Exception {
        this.mvc
            .perform(get("/todo"))
            .andExpect(status().isOk())
            .andExpect(content()
                .contentType(MediaType.APPLICATION_JSON_UTF8));
    }
}
```

You can also use the `TestRestTemplate` class.

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

```

@Autowired
private TestRestTemplate restTemplate;

@Test
public void todoTest() {
    String body = this.restTemplate.getForObject("/todo", String.class);
    assertThat(body).contains("Read a Book");
}
}

```

This code shows a test that is running a full server and using the `TestRestTemplate` instance to do a call to the `/todo` endpoint. Here we are assuming a `String` return. (This is not the best way to test a JSON return. Don't worry, you see the proper way to use the `TestRestTemplate` class later on).

Mocking Beans

The Spring Boot testing module offers a `@MockBean` annotation that defines a *Mockito* mock for a bean inside the `ApplicationContext`. In other words, you can mock a new Spring bean or replace an existing definition by adding this annotation. Remember, this is happening inside the `ApplicationContext`.

```

@RunWith(SpringRunner.class)
@SpringBootTest
public class ToDoSimpleMockBeanTests {

    @MockBean
    private ToDoRepository repository;

    @Test
    public void todoTest() {
        given(this.repository.findById("my-id"))
            .Return(new ToDo("Read a Book"));
        assertThat(
            this.repository.findById("my-id").getDescription())
            .isEqualTo("Read a Book");
    }
}

```

Spring Boot Testing Slices

One of the most important features that Spring Boot offers is a way to execute test without the need for a certain infrastructure. The Spring Boot testing module includes *slices* to test specific parts of the application without the need of a server or a database engine.

@JsonTest

The Spring Boot testing module provides the `@JsonTest` annotation, which helps with object JSON serialization and deserialization, and verifies that everything works as expected. `@JsonTest` auto-configures the supported JSON mapper, depending of which library is in the classpath: Jackson, GSON, or JSONB.

```
@RunWith(SpringRunner.class)
```

```
@JsonTest
```

```
public class ToDoJsonTests {
```

```
    @Autowired
```

```
    private JacksonTester<ToDo> json;
```

```
    @Test
```

```
    public void todoSerializeTest() throws Exception {
```

```
        ToDo todo = new ToDo("Read a Book");
```

```
        assertThat(this.json.write(todo))
```

```
            .isEqualToJson("todo.json");
```

```
        assertThat(this.json.write(todo))
```

```
            .hasJsonPathStringValue("@.description");
```

```
        assertThat(this.json.write(todo))
```

```
            .extractingJsonPathStringValue("@.description")
```

```
            .isEqualTo("Read a Book");
```

```
    }
```

```
    @Test
```

```
    public void todoDeserializeTest() throws Exception {
```

```
        String content = "{\"description\":\"Read a Book\",\"completed\":";
```

```
        true }";
```

```
        assertThat(this.json.parse(content))
```



```

        .isEqualTo(new Todo("Read a Book", true));
    assertThat(
        this.json.parseObject(content).getDescription())
        .isEqualTo("Read a Book");
    }
}

```

@WebMvcTest

If you need to test your controllers without using a complete server, Spring Boot provides the `@WebMvcTest` annotation that auto-configures the Spring MVC infrastructure and limits scanned beans to `@Controller`, `@ControllerAdvice`, `@JsonComponent`, `Converter`, `GenericConverter`, `Filter`, `WebMvcConfigurer`, and `HandlerMethodArgumentResolver`; so you know if your controllers are working as expected.

It's important to know that beans marked as `@Component` are not scanned when using this annotation, but you can still use the `@MockBean` if needed.

```
@RunWith(SpringRunner.class)
```

```
@WebMvcTest(ToDoController.class)
```

```
public class ToDoWebMvcTest {
```

```
    @Autowired
```

```
    private MockMvc mvc;
```

```
    @MockBean
```

```
    private ToDoRepository toDoRepository;
```

```
    @Test
```

```
    public void toDoControllerTest() throws Exception {
```

```
        given(this.toDoRepository.findById("my-id"))
```

```
            .Return(new Todo("Do Homework", true));
```

```
        this.mvc.perform(get("/todo/my-id").accept(MediaType.APPLICATION_
JSON_UTF8))
```

```
            .andExpect(status().isOk()).andExpect(content().
```

```
                string("{\"id\":\"my-id\", \"description\":\"Do Homework\",
                \"completed\":true}"));
```

```
    }
```

```
}
```

@WebFluxTest

Spring Boot provides the `@WebFluxTest` annotation for any reactive controller. This annotation auto-configures the Spring WebFlux module infrastructure and scans only for `@Controller`, `@ControllerAdvice`, `@JsonComponent`, `Converter`, `GenericConverter`, and `WebFluxConfigurer`.

It's important to know that beans marked as `@Component` are not scanned when using this annotation, but you can still use the `@MockBean` if needed.

```
@RunWith(SpringRunner.class)
@WebFluxTest(ToDoFluxController.class)
public class ToDoWebFluxTest {

    @Autowired
    private WebClient webClient;

    @MockBean
    private ToDoRepository todoRepository;

    @Test
    public void testExample() throws Exception {
        given(this.todoRepository.findAll())
            .Return(Arrays.asList(new ToDo("Read a Book"), new
                ToDo("Buy Milk")));
        this.webClient.get().uri("/todo-flux").accept(MediaType.
            APPLICATION_JSON_UTF8)
            .exchange()
            .expectStatus().isOk()
            .expectBody(List.class);
    }
}
```

@DataJpaTest

If you need to test your JPA application, the Spring Boot testing module offers `@DataJpaTest`, which auto-configures *in-memory* embedded databases. It scans `@Entity`. This annotation won't load any `@Component` bean. It also provides the `TestEntityManager` helper class that is very similar to the JPA `EntityManager` class, which specializes in testing.

```

@RunWith(SpringRunner.class)
@DataJpaTest
public class TodoDataJpaTests {

    @Autowired
    private TestEntityManager entityManager;

    @Autowired
    private TodoRepository repository;

    @Test
    public void todoDataTest() throws Exception {
        this.entityManager.persist(new Todo("Read a Book"));
        Iterable<Todo> todos = this.repository.
            findByDescriptionContains("Read a Book");
        assertThat(todos.iterator().next()).toString().contains("Read a Book");
    }
}

```

Remember that using `@DataJpaTest` uses embedded in-memory database engines (H2, Derby, HSQL), but if you want to test with a real database, you need to add the following `@AutoConfigureTestDatabase(replace=Replace.NONE)` annotation as a marker for the test class.

```

@RunWith(SpringRunner.class)
@DataJpaTest
@AutoConfigureTestDatabase(replace=Replace.NONE)
public class TodoDataJpaTests {
    //...
}

```

@JdbcTest

This annotation is very similar to `@DataJpaTest`; the only difference is that it does pure JDBC-related tests. It auto-configures the in-memory embedded database engine and it configures the `JdbcTemplate` class. It omits every class marked as `@Component`.

```

@RunWith(SpringRunner.class)
@JdbcTest
@Transactional(propagation = Propagation.NOT_SUPPORTED)
public class TodoJdbcTests {

    @Autowired
    private NamedParameterJdbcTemplate jdbcTemplate;

    private CommonRepository<ToDo> repository;

    @Test
    public void todoJdbcTest() {
        ToDo todo = new ToDo("Read a Book");

        this.repository = new ToDoRepository(jdbcTemplate);
        this.repository.save(todo);

        ToDo result = this.repository.findById(todo.getId());
        assertThat(result.getId()).isEqualTo(todo.getId());
    }
}

```

@DataMongoTest

The Spring Boot testing module provides the `@DataMongoTest` annotation to test Mongo applications. It auto-configures the in-memory embedded Mongo server (if available), if it doesn't, you need to add the right `spring.data.mongodb.*` properties. It configures the `MongoTemplate` class and it scans for `@Document` annotations. `@Component` beans aren't scanned.

```

@RunWith(SpringRunner.class)
@DataMongoTest
public class TodoMongoTests {

    @Autowired
    private MongoTemplate mongoTemplate;

```

```

@Test
public void toDoMongoTest() {
    ToDo toDo = new ToDo("Read a Book");
    this.mongoTemplate.save(toDo);

    ToDo result = this.mongoTemplate.findById(toDo.getId(),ToDo.class);
    assertThat(result.getId()).isEqualTo(toDo.getId());
}
}

```

If you require an external MongoDB server (not in-memory embedded), add the `excludeAutoConfiguration = EmbeddedMongoAutoConfiguration.class` parameter to the `@DataMongoTest` annotation.

```

@RunWith(SpringRunner.class)
@DataMongoTest(excludeAutoConfiguration = EmbeddedMongoAutoConfiguration.class)
public class ToDoMongoTests {
    // ...
}

```

@RestClientTest

Another important annotation is the `@RestClientTest`, which tests your REST clients. This annotation auto-configures Jackson, GSON, and JSONB support. It configures the `RestTemplateBuilder` and adds support for `MockRestServiceServer`.

```

@RunWith(SpringRunner.class)
@RestClientTest(ToDoService.class)
public class ToDoRestClientTests {

    @Autowired
    private ToDoService service;

    @Autowired
    private MockRestServiceServer server;
}

```

```

@Test
public void todoRestClientTest()
    throws Exception {
    String content = "{\"description\": \"Read a Book\", \"completed\":
    true }";
    this.server.expect(requestTo("/todo/my-id"))
        .andRespond(withSuccess(content, MediaType.APPLICATION_JSON_
        UTF8));
    ToDo result = this.service.findById("my-id");
    assertThat(result).isNotNull();
    assertThat(result.getDescription()).contains("Read a Book");
}
}

```

There are many other slices that you can check out. The important note to take is that you aren't required to have a whole infrastructure or to have servers running to do testing. Slices facilitate more testing for Spring Boot apps.

Summary

In this chapter, you learned different ways to test apps with Spring Boot. Even though this chapter is short, I showed you some of the important features, such as slices.

In the next chapter, we cover security and learn how Spring Boot can secure our applications.

CHAPTER 8

Security with Spring Boot

This chapter shows you how to use security in your Spring Boot applications to secure your web application. You learn everything from using basic security to using OAuth. Security has become a primary and important factor for desktop, web, and mobile applications in the last decade. But security is a little hard to implement because you need to think about everything—cross-site scripting, authorization, and authentication, secure sessions, identification, encryption, and a lot more. There is still a lot to do to implement simple security in your applications.

The Spring security team has worked hard to make it easier for developers to bring security to their applications, from securing service methods to entire web applications. Spring security is centered around `AuthenticationProvider`, `AuthenticationManager`, and specialized `UserDetailsService`; it also provides integration with identity provider systems, such as LDAP, Active Directory, Kerberos, PAM, OAuth, and so on. You are going to review a few of them in the examples in this chapter.

Spring Security

Spring Security is highly customizable and powerful framework that helps with authentication and authorization (or access control); it is the default module for securing Spring applications. The following are some of the important features.

- Servlet API integration
- Integration with Spring Web MVC and WebFlux
- Protection against attacks such as session fixation, clickjacking, CSRF (cross-site request forgery), CORS (cross-origin resource sharing), and so forth

- Extensible and comprehensive support for both authentication and authorization
- Integration with these technologies: HTTP Basic, HTTP Digest, X.509, LDAP, Form-based, OpenID, CAS, RMI, Kerberos, JAAS, Java EE, and more
- Integration with third-party technologies: AppFuse, DWR, Grails, Tapestry, JOSSO, AndroMDA, Roller, and many more

Spring Security has become the de facto way to use security on many Java and Spring projects because it integrates and customizes with minimal effort, creating robust and secure apps.

Security with Spring Boot

Spring Boot uses the power of the Spring Security Framework to secure applications. To use Spring Security it is necessary to add the `spring-boot-starter-security` dependency. This dependency provides all the `spring-security-core` jars and it auto-configures the strategy to determine whether to use `httpBasic` or `formLogin` authentication mechanisms. It defaults to `UserDetailsService` with a single user. This username is `user` and the password is printed (RANDOM string) as a log with INFO level when the application starts.

In other words, by adding the `spring-boot-starter-security` dependency, your application is already secured.

ToDo App with Basic Security

Let's start with the ToDo app. Here, you use the same code as the JPA REST project; but I'll review the class once again. So let's begin. Starting from scratch, go to your browser and open Spring Initializr (<https://start.spring.io>). Add the following values to the fields:

- Group: `com.apress.todo`
- Artifact: `todo-simple-security`
- Name: `todo-simple-security`

- Package Name: `com.apress.todo`
- Dependencies: Web, Security, Lombok, JPA, REST Repositories, H2, MySQL, Mustache

You can select either Maven or Gradle as the project type. Then you can press the Generate Project button; this downloads a ZIP file. Uncompress it and import the project in your favorite IDE (see Figure 8-1).

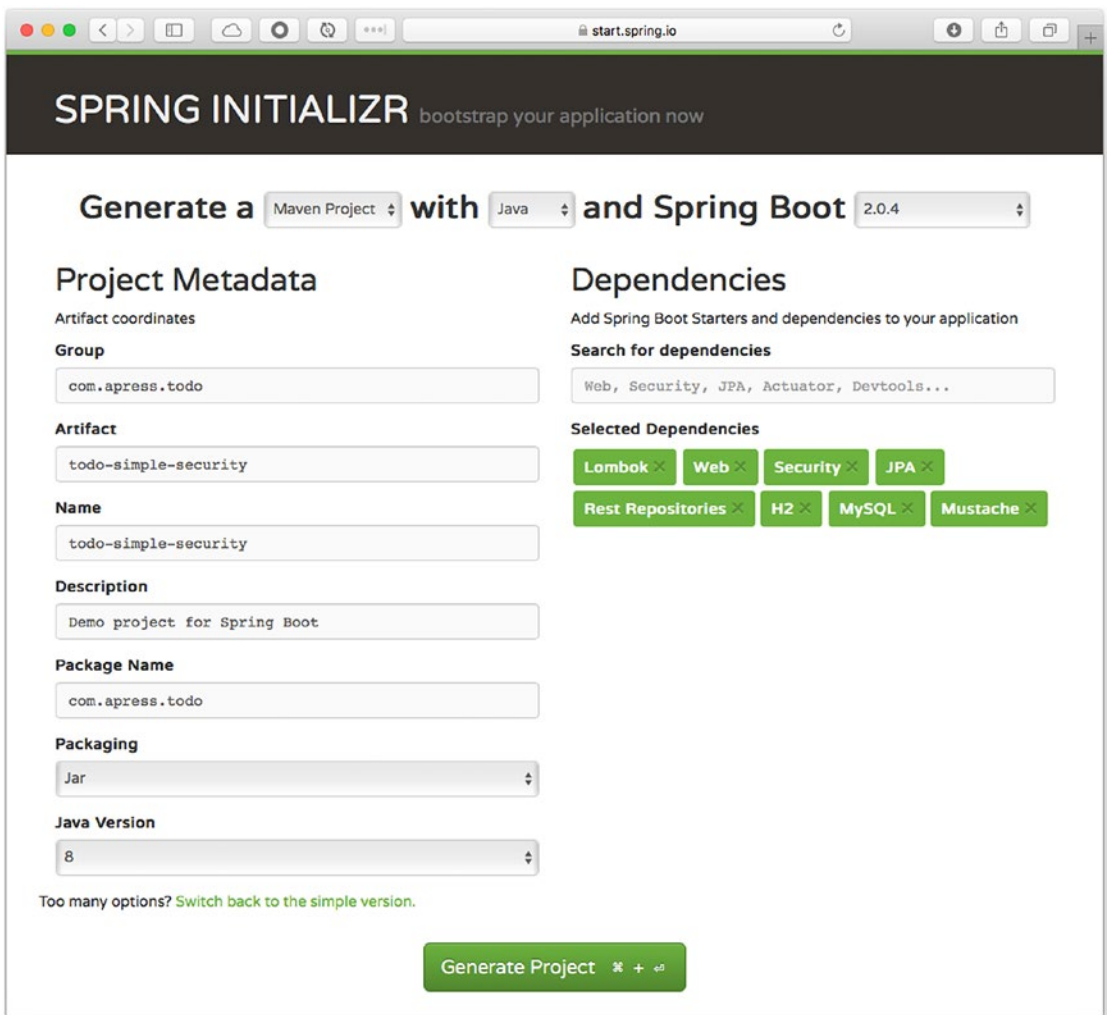


Figure 8-1. Spring Initializr

This project now has the Security module and a template engine, Mustache. Very soon you see how to use it.

Let's start with the `ToDo` domain class (see Listing 8-1).

Listing 8-1. `com.apress.todo.domain.ToDo.java`

```
package com.apress.todo.domain;

import lombok.Data;
import org.hibernate.annotations.GenericGenerator;

import javax.persistence.*;
import javax.validation.constraints.NotBlank;
import javax.validation.constraints.NotNull;
import java.time.LocalDateTime;

@Entity
@Data
public class ToDo {

    @Id
    @GeneratedValue(generator = "system-uuid")
    @GenericGenerator(name = "system-uuid", strategy = "uuid")
    private String id;
    @NotNull
    @NotBlank
    private String description;

    @Column(insertable = true, updatable = false)
    private LocalDateTime created;
    private LocalDateTime modified;
    private boolean completed;

    public ToDo(){ }
    public ToDo(String description){
        this.description = description;
    }
}
```

```

@PrePersist
void onCreate() {
    this.setCreated(LocalDateTime.now());
    this.setModified(LocalDateTime.now());
}

@PreUpdate
void onUpdate() {
    this.setModified(LocalDateTime.now());
}
}

```

Listing 8-1 shows the `ToDo` domain class. You already know about it. It's marked with `@Entity` and it's using `@Id` for a primary key. This class is from the *todo-rest* project.

Next, let's review the `ToDoRepository` interface (see Listing 8-2).

Listing 8-2. `com.apress.todo.repository.ToDoRepository.java`

```

package com.apress.todo.repository;

import com.apress.todo.domain.ToDo;
import org.springframework.data.repository.CrudRepository;

public interface ToDoRepository extends CrudRepository<ToDo,String> {

}

```

Listing 8-2 shows the `ToDoRepository`, and of course, you already know about it. Defining the interface that extends from the `CrudRepository<T, ID>` that has not only the CRUD methods, but also the Spring Data REST, creates all the necessary REST APIs to support the domain class.

Next, let's review the `application.properties` and see what is new (see Listing 8-3).

Listing 8-3. `src/main/resources/application.properties`

```

# JPA
spring.jpa.generate-ddl=true
spring.jpa.hibernate.ddl-auto=create-drop
spring.jpa.show-sql=true

```

```
# H2-Console: http://localhost:8080/h2-console
# jdbc:h2:mem:testdb
spring.h2.console.enabled=true

# REST API
spring.data.rest.base-path=/api
```

Listing 8-3 shows you the `application.properties` file. You've seen already some of the properties, except for the last one, right? The `spring.data.rest.base-path` tells the RestController (of the Spring Data REST configuration) that uses the `/api` as the root to expose all the REST API endpoints. So if we want to get `ToDo`'s, we need to access the endpoint at `http://localhost:8080/api/todos`.

Before running the app, let's add the endpoint in the form of a script. Create the `src/main/resources/data.sql` file with the following SQL statements.

```
insert into to_do (id,description,created,modified,completed)
values ('8a8080a365481fb00165481fbca90000', 'Read a Book', '2018-08-17
07:42:44.136', '2018-08-17 07:42:44.137', true);

insert into to_do (id,description,created,modified,completed)
values ('ebcf1850563c4de3b56813a52a95e930', 'Buy Movie Tickets', '2018-08-17
09:50:10.126', '2018-08-17 09:50:10.126', false);

insert into to_do (id,description,created,modified,completed)
values ('78269087206d472c894f3075031d8d6b', 'Clean my Room', '2018-08-17
07:42:44.136', '2018-08-17 07:42:44.137', false);
```

Now, if you run your application, you should see in the logs this output:

```
Using generated security password: 2a569843-122a-4559-a245-60f5ab2b6c51
```

This is your password. You can now go to your browser and open `https://localhost:8080/api/todos`. When you hit Enter to access that URL, you get something similar to Figure 8-2.

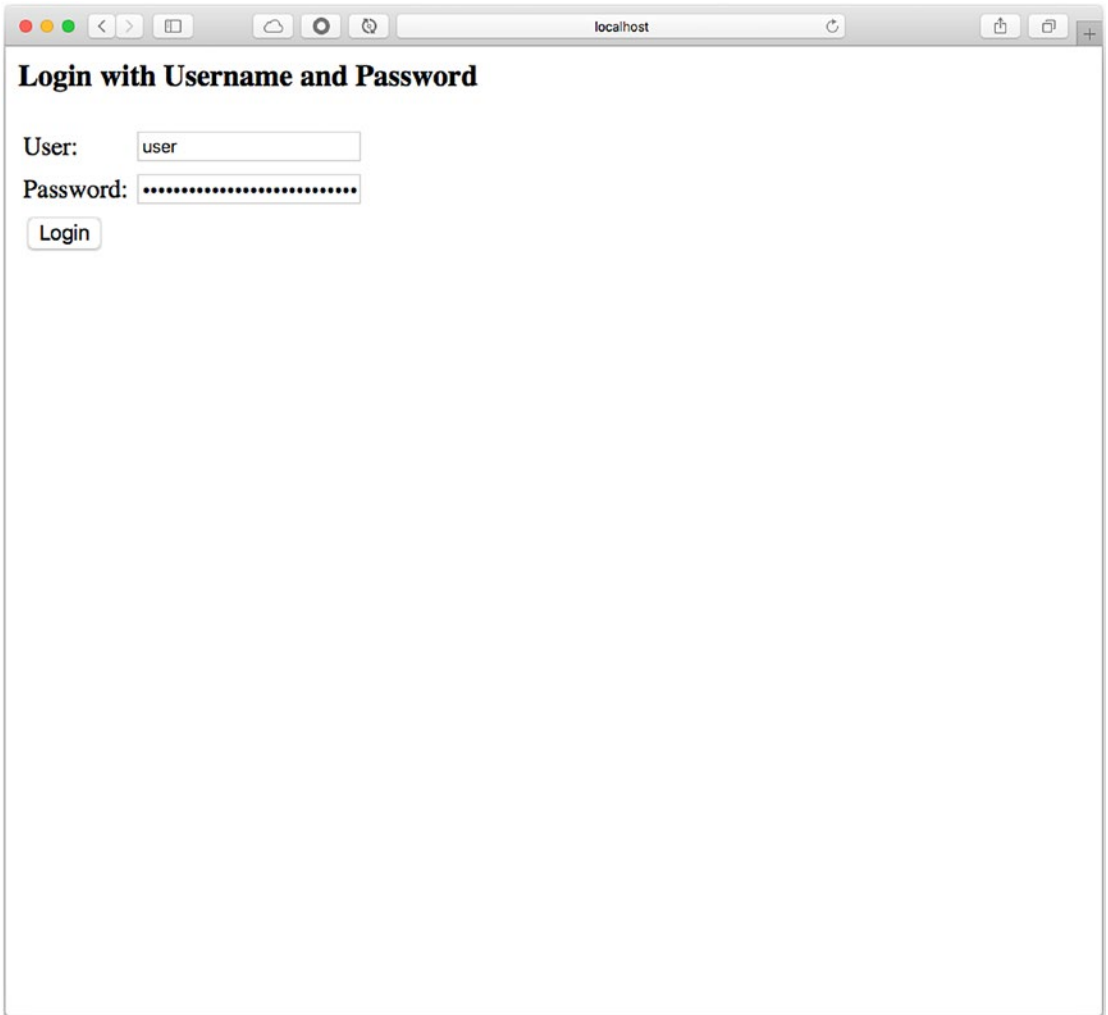


Figure 8-2. *ToDo App: <http://localhost:8080/login> page*

Figure 8-2 shows a login page, which is the default behavior when you add the `spring-boot-starter-security` dependency. By default, Security is on—so simple!! So, what is the user and password? Well, I mentioned this earlier, the user is `user`, and the password is the random one that was printed in the logs (in this example, `2a569843-122a-4559-a245-60f5ab2b6c51`). So, go ahead and enter the username and password; then you should get the `ToDo`'s list (see Figure 8-3).



Figure 8-3. `http://localhost:8080/api/todos`

If you want to try using the command line, you can execute the following command in a terminal window.

```
$ curl localhost:8080/api/todos
```

```
{"timestamp":"2018-08-19T21:25:47.224+0000","status":401,"error":"Unauthorized",
"message":"Unauthorized","path":"/api/todos"}
```

As you can see from the output, you are not authorized to get into that endpoint. Authentication is needed, right? You can execute the following command.

```
$ curl localhost:8080/api/todos -u user:2a569843-122a-4559-a245-60f5ab2b6c51
{
  "_embedded" : {
    "todos" : [ {
      "description" : "Read a Book",
      "created" : "2018-08-17T07:42:44.136",
      "modified" : "2018-08-17T07:42:44.137",
      "completed" : true,
      ...
    }
  ]
}
```

As you can see now, you are passing the username and the random password, and you are getting the response with the list of `ToDo`'s.

As probably you already know, every time you restart this app, the security auto-configuration generates another random password, and that's not optimal; maybe just for development.

Overriding Simple Security

Random passwords don't do the trick in a production environment. Spring Boot Security allows you to override the defaults in multiple ways. The simplest is to override it with the `application.properties` file by adding the following `spring.security.*` properties.

```
spring.security.user.name=apress
spring.security.user.password=springboot2
spring.security.user.roles=ADMIN,USER
```

If you run the app again, the username is `apress` and the password is `springboot2` (the same as in a command line). Also notice that in the logs, the random password is no longer printed.

Another way is to provide authentication programmatically. Create a `ToDoSecurityConfig` class that extends from `WebSecurityConfigureAdapter`. Take a look at Listing 8-4.

Listing 8-4. com.apress.todo.config.ToDoSecurityConfig.java

```

package com.apress.todo.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.authentication.
builders.AuthenticationManagerBuilder;
import org.springframework.security.config.annotation.web.configuration.
WebSecurityConfigurerAdapter;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

@Configuration
public class ToDoSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(
        AuthenticationManagerBuilder auth) throws Exception {
        auth.inMemoryAuthentication()
            .passwordEncoder(passwordEncoder())
            .withUser("apress")
            .password(passwordEncoder().encode("springboot2"))
            .roles("ADMIN", "USER");
    }

    @Bean
    public BCryptPasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
}

```

Listing 8-4 shows the necessary configuration for programmatically building the security, in this case, with one user (you can add more, of course). Let's analyze the code.

- **WebSecurityConfigurerAdapter**. Extending this class is one way to override security because it allows you to override the methods that you really need. In this case, the code overrides the `configure(AuthenticationManagerBuilder)` signature.

- `AuthenticationManagerBuilder`. This class creates an `AuthenticationManager` that allows you to easily build in memory, LDAP, JDBC authentications, `UserDetailsService` and add `AutheticationProviders`. In this case, you are building an in-memory authentication. It's necessary to add a `PasswordEncoder` and a new and more secure way to use and encrypt/decrypt the password.
- `BCryptPasswordEncoder`. In this code you are using the `BCryptPasswordEncoder` (returns a `PasswordEncoder` implementation) that uses the BCrypt strong hashing function. You can use also `Pbkdf2PasswordEncoder` (uses PBKDF2 with a configurable number of iterations and a random 8-byte random salt value), or `SCryptPasswordEncoder` (uses the SCrypt hashing function). Even better, use `DelegatingPasswordEncoder`, which supports password upgrades.

Before you run the application, comment out the `spring.security.*` properties that you added to the `application.properties` file. If you run the app, it should work as expected. You need to provide the username, `apress`, and the password, `springboot2`.

Overriding the Default Login Page

Spring Security allows you to override the default login page in several ways. One way is to configure `HttpSecurity`. The `HttpSecurity` class allows you to configure web-based security for specific HTTP requests. By default, it is applied to all requests, but can be restricted using `requestMatcher` (`RequestMatcher`) or similar methods.

Let's look at a modification of the `ToDoSecurityConfig` class (see Listing 8-5).

Listing 8-5. `com.apress.todo.config.ToDoSecurityConfig.java` - v2

```
package com.apress.todo.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.authentication.
builders.AuthenticationManagerBuilder;
```

```

import org.springframework.security.config.annotation.web.builders.
HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.
WebSecurityConfigurerAdapter;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

@Configuration
public class ToDoSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws
    Exception {
        auth.inMemoryAuthentication()
            .passwordEncoder(passwordEncoder())
            .withUser("apress")
            .password(passwordEncoder().encode("springboot2"))
            .roles("ADMIN", "USER");
    }

    @Bean
    public BCryptPasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .anyRequest().fullyAuthenticated()
            .and()
            .httpBasic();
    }
}

```

Listing 8-5 shows version 2 of the `ToDoSecurityConfig` class. If you run the app and go to the browser (<http://localhost:8080/api/todos>), you now get a pop-up for the basic authentication (see Figure 8-4).

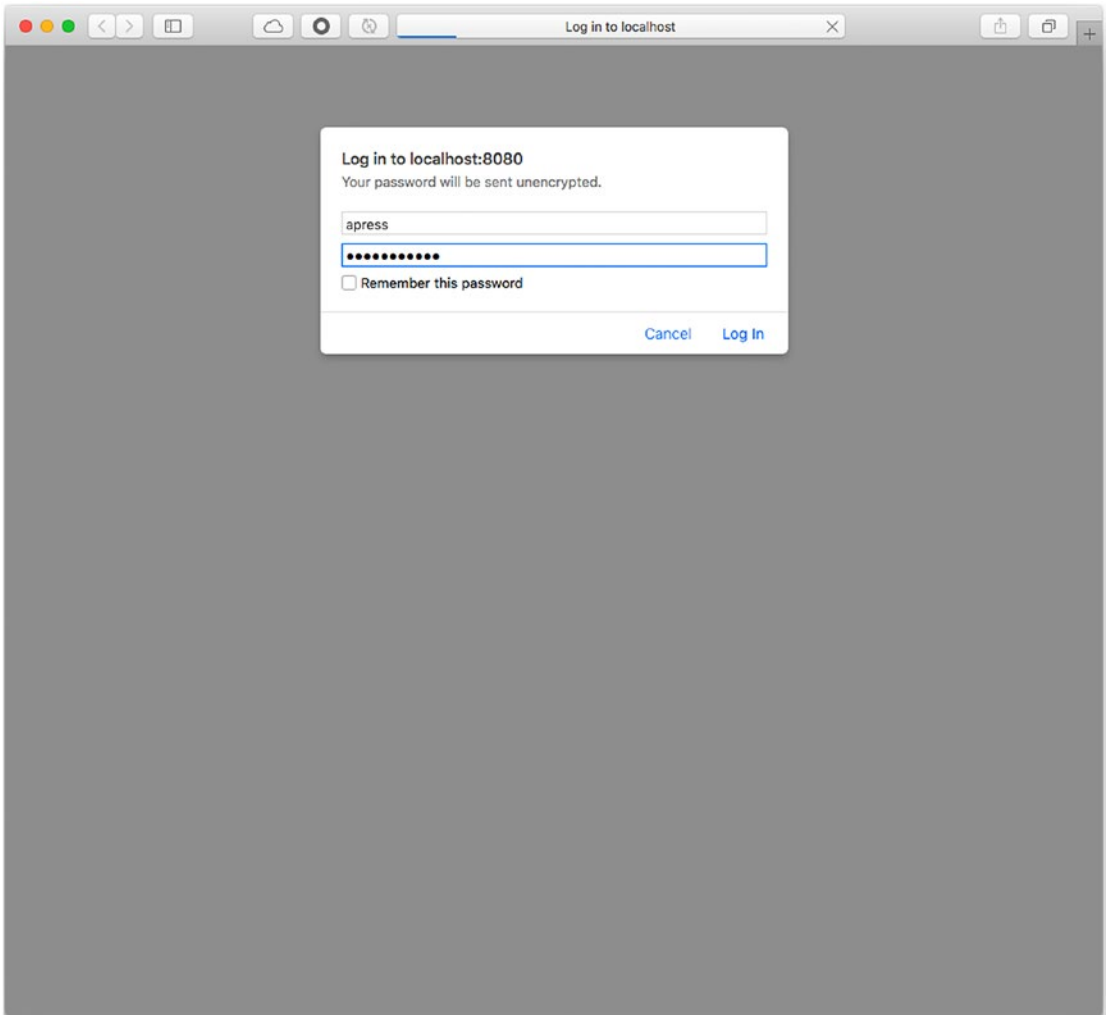


Figure 8-4. *http://localhost:8080/api/todos—Http Basic Authentication*

You can use the username and password that you already know, and you should get the ToDo's list. It is the same for the command line. You need to authenticate

```
$ curl localhost:8080/api/todos -u apress:springboot2
```

Custom Login Page

Normally in applications, you never see a page like that; typically, there is a very nice and well-designed login page, right? Spring Security allows you to create and customize your login page.

Let's prepare the ToDo app with a login page. First, we are going to add some CSS and the well-known jQuery library. Nowadays in a Spring Boot app, we can use WebJars dependencies. This new way avoids manually downloading the files; instead, you can use them as resources. Spring Boot web auto-configuration creates the necessary access for them.

If you are using Maven, open `pom.xml` and add the following dependencies.

```
<dependency>
  <groupId>org.webjars</groupId>
  <artifactId>bootstrap</artifactId>
  <version>3.3.7</version>
</dependency>

<dependency>
  <groupId>org.webjars</groupId>
  <artifactId>jquery</artifactId>
  <version>3.2.1</version>
</dependency>
```

If you are using Gradle, open your `build.gradle` file and add the following dependencies.

```
compile ('org.webjars:bootstrap:3.3.7')
compile ('org.webjars:jquery:3.2.1')
```

Next, let's create the login page, which has the `.mustache` extension (`login.mustache`). It must be created in the `src/main/resources/templates` folder (see Listing 8-6).

Listing 8-6. `src/main/resources/templates/login.mustache`

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
```

```

<meta http-equiv="X-UA-Compatible" content="IE=edge">
<meta name="viewport" content="width=device-width, initial-scale=1">
<title>ToDo's API Login Page</title>
<link href="webjars/bootstrap/3.3.7/css/bootstrap.min.css"
rel="stylesheet">
<link href="css/signin.css" rel="stylesheet">
</head>

<body>

<div class="container">
  <form class="form-signin" action="/login" method="POST">
    <h2 class="form-signin-heading">Please sign in</h2>

    <label for="username" class="sr-only">Username</label>
    <input type="text"      name="username" class="form-control"
placeholder="Username" required autofocus>

    <label for="inputPassword" class="sr-only">Password</label>
    <input type="password" name="password" class="form-control"
placeholder="Password" required>

    <button class="btn btn-lg btn-primary btn-block" id="login"
type="submit">Sign in</button>
    <input type="hidden" name="_csrf" value="{{_csrf.token}}" />
  </form>
</div>
</body>
</html>

```

Listing 8-6 shows the HTML login page. This page is using CSS from Bootstrap (<https://getbootstrap.com>) through the WebJars (www.webjars.org) dependencies. These files are taken as file resources from those jars. HTML-FORM is using *username* and *password* as names (a must for Spring Security). We need to include the CSRF token to avoid any attacks. The Mustache engine provides this with the `{{_csrf.token}}` value. Spring Security uses the *synchronizer token pattern* to avoid any attacks in requests. Later on, we are going to see how we get this value.

Next, let's create an index page that lets you see the homepage and log out. Create the `index.mustache` page in the `src/main/resources/templates` folder (see Listing 8-7).

Listing 8-7. `src/main/resources/templates/index.mustache`

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>ToDo's API</title>
  <link href="webjars/bootstrap/3.3.7/css/bootstrap.min.css"
    rel="stylesheet">
  <script src="webjars/jquery/3.2.1/jquery.min.js"></script>
</head>
<body>
<div class="container">
  <div class="header clearfix">
    <nav>
      <a href="#" id="logoutLink">Logout</a>
    </nav>
  </div>
  <div class="jumbotron">
    <h1>ToDo's Rest API</h1>
    <p class="lead">Welcome to the ToDo App. A Spring Boot
      application!</p>
  </div>
</div>
<form id="logout" action="/logout" method="POST">
  <input type="hidden" name="_csrf" value="{{_csrf.token}}" />
</form>
<script>
  $(function(){
```

```

        $('#logoutLink').click(function(){
            $('#logout').submit();
        });
    });
</script>
</body>
</html>

```

Listing 8-7 shows the index page. We are still using Bootstrap and the jQuery resources, and the most important part, the `{{_csrf.token}}`, for logout.

Next, let's start with the configuration. First, it is necessary to modify the `ToDoSecurityConfig` class (see Listing 8-8).

Listing 8-8. `com.apress.todo.config.ToDoSecurityConfig.java - v3`

```

package com.apress.todo.config;

import org.springframework.boot.autoconfigure.security.servlet.PathRequest;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.authentication.
builders.AuthenticationManagerBuilder;
import org.springframework.security.config.annotation.web.builders.
HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.
WebSecurityConfigurerAdapter;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.web.util.matcher.AntPathRequestMatcher;

@Configuration
public class ToDoSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws
    Exception {
        auth.inMemoryAuthentication()
            .passwordEncoder(passwordEncoder())
            .withUser("apress")

```

```

        .password(passwordEncoder().encode("springboot2"))
        .roles("ADMIN", "USER");
    }

    @Bean
    public BCryptPasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests()
                .requestMatchers(
                    PathRequest
                        .toStaticResources()
                        .atCommonLocations()).permitAll()
                .anyRequest().fullyAuthenticated()
                .and()
                .formLogin().loginPage("/login").permitAll()
                .and()
                .logout()
                    .logoutRequestMatcher(
                        new AntPathRequestMatcher("/logout"))
                    .logoutSuccessUrl("/login");
    }
}

```

Listing 8-8 shows version 3 of the `ToDoSecurityConfig` class. The new modification show how `HttpSecurity` is being configured. First, its adding `requestMatchers`, which point to common locations, such as the static resources (`static/*`). This is where CSS, JS, or any other simple HTML can live and doesn't need any security. Then it uses `anyRequest`, which should be `fullyAuthenticated`. this means that the `/api/*` will be. Then, it uses `formLogin` to specify with `loginPage("/login")` that it is the endpoint for finding the login page. Next, declare the `logout` and its endpoint (`/logout`); if the `logout` is successful, it redirects to the `/login` endpoint/page.

Now it is necessary to tell Spring MVC how to locate the login page. Create the `ToDoWebConfig` class (see Listing 8-9).

Listing 8-9. `com.apress.todo.config.ToDoWebConfig.java`

```
package com.apress.todo.config;

import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.
ViewControllerRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration
public class ToDoWebConfig implements WebMvcConfigurer {

    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController("/login").setViewName("login");
    }
}
```

Listing 8-9 shows a different way of configuring a web controller in Spring MVC. You can still use a class annotated with `@Controller` and create the mapping for the login page; but this is the `JavaConfig` way.

Here the class is implementing the `WebMvcConfigure` interface. It's implementing the `addViewControllers` method and registering the `/login` endpoint by telling the controller where the view is. This locates the `templates/login.mustache` page.

Finally, it is necessary to update the `application.properties` file by adding the following property.

```
spring.mustache.expose-request-attributes=true
```

Remember the `{{_csrf.token}}`? This is how it gets its value—by adding the `spring.mustache.expose-request-attributes` property.

Now, you can run the application. If you go to `http://localhost:8080`, you get something similar to Figure 8-5.

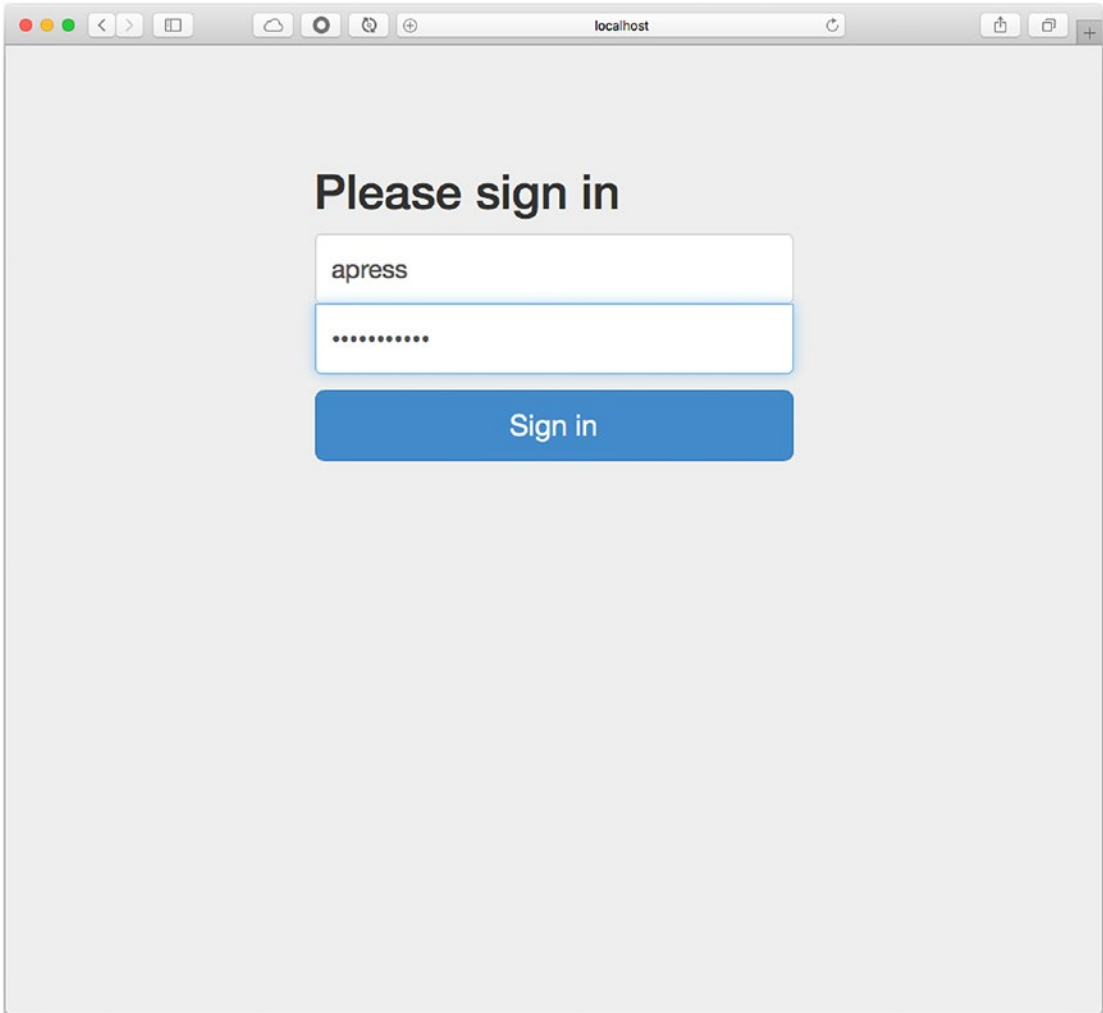


Figure 8-5. `http://localhost:8080/login`

You get the custom login page. Perfect!! Now you can enter the credentials, and it returns the index page (see Figure 8-6).

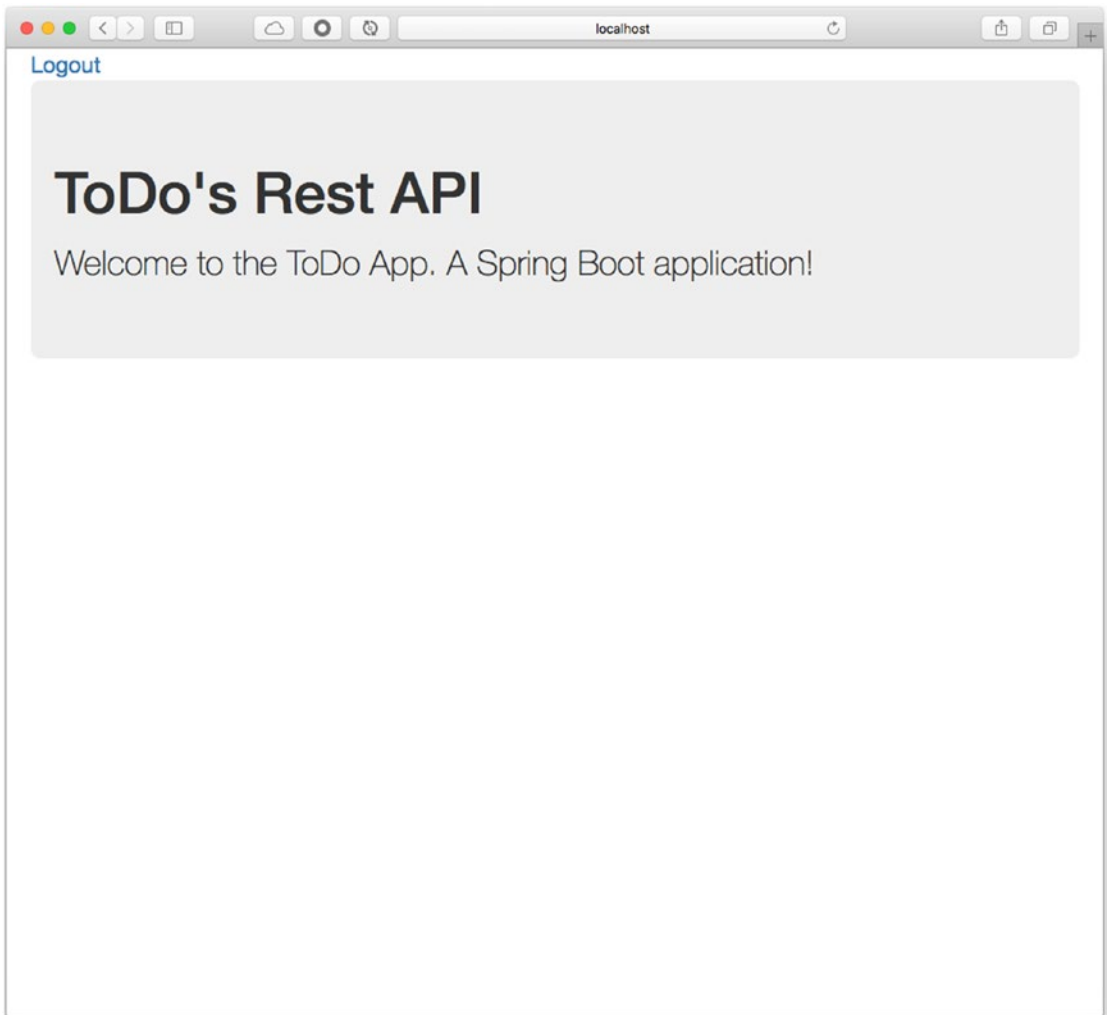


Figure 8-6. *http://localhost:8080 after login*

Once you have the homepage, you can visit `http://localhost:8080/api/toDos`. You should be fully authenticated, and you can go back to the ToDo's list. You can go back to the homepage and press the Logout link, which redirects you to the `/login` endpoint again.

Now, what happens if you try to execute the following command line in a terminal window?

```
$ curl localhost:8080/api/toDos -u apress:springboot2
```

It won't return anything. It is an empty line. If you use the `-i` flag, it tells you that you are being redirected to `http://localhost:8080/login`. But there is no way to interact from the command line, right? So what can we do to fix this? In reality, there are clients that never use web interfaces. Most of the clients are apps and programmatically need to use the REST API, but with this solution, there is no way to do authentication to interact with a form.

Open the `ToDoSecurityConfig` class and modify the `configure(HttpSecurity)` method. It should look like the following snippet.

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
        .requestMatchers(PathRequest.toStaticResources().
            atCommonLocations()).permitAll()
        .anyRequest().fullyAuthenticated()
        .and()
        .formLogin().loginPage("/login").permitAll()
        .and()
        .logout()
        .logoutRequestMatcher(
            new AntPathRequestMatcher("/logout"))
        .logoutSuccessUrl("/login")
        .and()
        .httpBasic();
}
```

The last two lines of the method add the `httpBasic` call, which allows clients (like `cURL`) to use the basic authentication mechanisms. You can re-run the `ToDo` app and see that the executing the command line works now.

Using Security with JDBC

Imagine for a moment that your company already has an employee database, and you want to reuse it for authentication and authorization for the `ToDo` app. It is nice to integrate something like that, right?

Spring Security allows you to use `AuthenticationManager` with in-memory, LDAP and JDBC mechanisms. In this section, we are going to modify the `ToDo` app to run with JDBC.

Directory App with JDBC Security

In this section, you create a new app—a directory application where all the personnel are. The Directory app is integrated with the `ToDo` app to do the authentication and authorization. So, if a client needs to add a new `ToDo`, it needs to be authenticated with a `USER` role.

So let's begin. Starting from scratch, go to your browser and open Spring Initializr. Add the following values to the fields.

- Group: `com.apress.directory`
- Artifact: `directory`
- Name: `directory`
- Package Name: `com.apress.directory`
- Dependencies: `Web`, `Security`, `Lombok`, `JPA`, `REST Repositories`, `H2`, `MySQL`

You can select either Maven or Gradle as the project type. Then you can press the `Generate Project` button, which downloads a ZIP file. Uncompress it and import the project in your favorite IDE (see Figure 8-7).

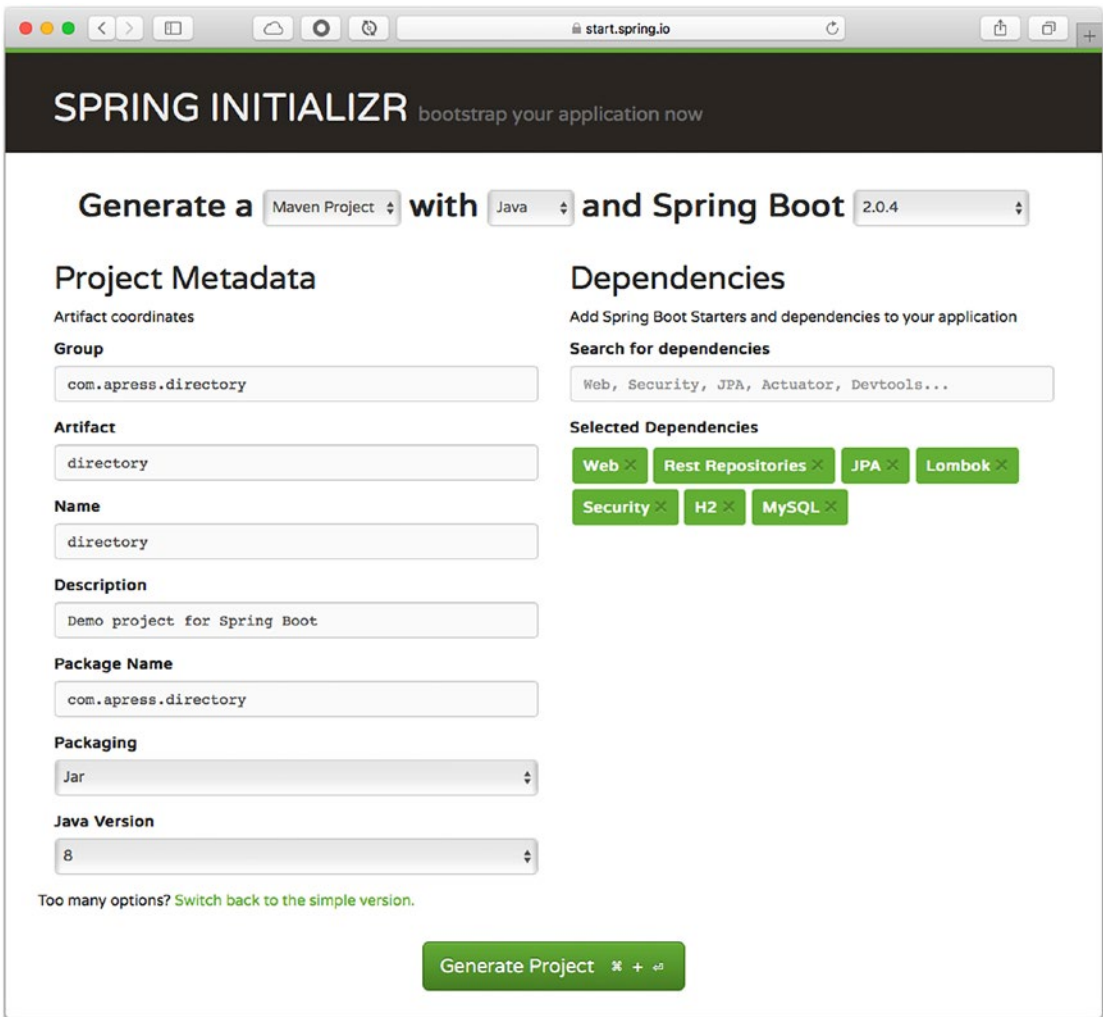


Figure 8-7. Spring Initializr

As you can see, the dependencies are very similar to other projects. We are going to use the power of Spring Data, Security, and REST. Let’s start by adding a new class that holds a person’s information. Create the Person class (see Listing 8-10).

Listing 8-10. com.apress.directory.domain.Person.java

```
package com.apress.directory.domain;

import lombok.Data;
import org.hibernate.annotations.GenericGenerator;
```

```
import javax.persistence.*;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
@Data
@Entity
public class Person {

    @Id
    @GeneratedValue(generator = "system-uuid")
    @GenericGenerator(name = "system-uuid", strategy = "uuid")
    private String id;
    @Column(unique = true)
    private String email;
    private String name;
    private String password;
    private String role = "USER";
    private boolean enabled = true;
    private LocalDate birthday;

    @Column(insertable = true, updatable = false)
    private LocalDateTime created;
    private LocalDateTime modified;

    public Person() {
    }

    public Person(String email, String name, String password, String
    birthday) {
        this.email = email;
        this.name = name;
        this.password = password;
        this.birthday = LocalDate.parse(birthday, DateTimeFormatter.
        ofPattern("yyyy-MM-dd"));
    }
}
```

```

public Person(String email, String name, String password, LocalDate
birthday) {
    this.email = email;
    this.name = name;
    this.password = password;
    this.birthday = birthday;
}

public Person(String email, String name, String password, String
birthday, String role, boolean enabled) {
    this(email, name, password, birthday);
    this.role = role;
    this.enabled = enabled;
}

@PrePersist
void onCreate() {
    this.setCreated(LocalDateTime.now());
    this.setModified(LocalDateTime.now());
}

@PreUpdate
void onUpdate() {
    this.setModified(LocalDateTime.now());
}
}

```

Listing 8-10 shows the Person class; very simple. It holds enough information about a person. Next, let's create the repository—the PersonRepository interface (see Listing 8-11).

Listing 8-11. com.apress.directory.repository.PersonRepository.java

```

package com.apress.directory.repository;

import com.apress.directory.domain.Person;
import org.springframework.data.repository.CrudRepository;
import org.springframework.data.repository.query.Param;

```



```
public interface PersonRepository extends CrudRepository<Person,String> {
    public Person findByEmailIgnoreCase(@Param("email") String email);
}
```

Listing 8-11 shows the `PersonRepository` interface; but what is different from the others? It declared a *query-method* `findByEmailIgnoreCase` with an email as the parameter (annotated by `@Param`). This syntax tells the Spring Data REST that it needs to implement these methods and create the SQL statement accordingly (this is based on the name and the fields in the domain class, in this case, the email field).

Note If you want to learn more about how to define your own query-method, take a look at the Spring Data JPA Reference at <https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods>.

Next, create the `DirectorySecurityConfig` class that extends from the `WebSecurityConfigurerAdapter` class. Remember that by extending from this class, we can customize the way Spring Security is set for this app (see Listing 8-12).

Listing 8-12. `com.apress.directory.config.DirectorySecurityConfig.java`

```
package com.apress.directory.config;

import com.apress.directory.repository.PersonRepository;
import com.apress.directory.security.DirectoryUserDetailsService;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;

@Configuration
public class DirectorySecurityConfig extends WebSecurityConfigurerAdapter {
    private PersonRepository personRepository;
```

```

public DirectorySecurityConfig(PersonRepository personRepository){
    this.personRepository = personRepository;
}

@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        .authorizeRequests()
        .antMatchers("/**").hasRole("ADMIN")
        .and()
        .httpBasic();
}

@Override
public void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.userDetailsService(
        new DirectoryUserDetailsService(this.personRepository));
}
}

```

Listing 8-12 shows the `DirectorySecurityConfig` class. This class is configuring `HttpSecurity` by allowing only users with an ADMIN role to any endpoint (`/**`) using basic authentication.

What else is different from other security configs? You are right! The `AuthenticationManager` is configuring a `UserDetailsService` implementation. This is the key to using any other third-party security app and integrating them with Spring Security.

As you can see, the `userDetailsService` method is using the `DirectoryUserDetailsService` class. Let's create it (see Listing 8-13).

Listing 8-13. `com.apress.directory.security.DirectoryUserDetailsService.java`

```

package com.apress.directory.security;

import com.apress.directory.domain.Person;
import com.apress.directory.repository.PersonRepository;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetails;

```

```

import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.
UsernameNotFoundException;
import org.springframework.security.crypto.factory.
PasswordEncoderFactories;
import org.springframework.security.crypto.password.PasswordEncoder;

public class DirectoryUserDetailsService implements UserDetailsService {

    private PersonRepository repo;

    public DirectoryUserDetailsService(PersonRepository repo) {
        this.repo = repo;
    }

    @Override
    public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
        try {
            final Person person = this.repo.findByEmailIgnoreCase(username);

            if (person != null) {
                PasswordEncoder encoder = PasswordEncoderFactories.
createDelegatingPasswordEncoder();
                String password = encoder.encode(person.getPassword());

                return User.withUsername(person.getEmail()).
accountLocked(!person.isEnabled()).password(password).
roles(person.getRole()).build();
            }
        }catch(Exception ex){
            ex.printStackTrace();
        }

        throw new UsernameNotFoundException(username);
    }
}

```

Listing 8-13 shows the `DirectoryUserDetailsService` class. This class implements the `UserDetailsService` interface and needs to implement `loadUserByUsername` and return a `UserDetails` instance. In this implementation, the code is showing how the `PersonRepository` is being used. In this case, it uses `findByEmailIgnoreCase`; so, if a person is found with the email provided at the time the user wants to access `/**` (any endpoint), it compares the email vs. the password provided, the role, and if the account is locked or not, by creating a `UserDetails` instance.

This is amazing! This app is using JDBC as a mechanism for authentication. Again, you can plug in any other security system/app that can implement `UserDetailsService` and return a `UserDetails` instance; that's it.

Next, let's quickly review the `application.properties` file and see its properties.

```
# Server
server.port=${port:8181}

# JPA
spring.jpa.generate-ddl=true
spring.jpa.hibernate.ddl-auto=create-drop
spring.jpa.show-sql=true

# H2
spring.h2.console.enabled=true
```

The only difference is that it has the `server.port` property, which says: *If you provide the variable port (either command line, environment) I will use it; if not, I will use port 8181.* That's the `:`. This is part of the SpEL (Spring Expression Language).

Before running the Directory app, let's add some data. Create the `data.sql` file in the `src/main/resources` folder.

```
insert into person (id,name,email,password,role,enabled,birthday,created,
modified)
values ('dc952d19ccfc4164b5eb0338d14a6619','Mark','mark@example.com','
secret','USER',true,'1960-03-29','2018-08-17 07:42:44.136','2018-08-17
07:42:44.137');

insert into person (id,name,email,password,role,enabled,birthday,created,
modified)
```

```

values ('02288a3b194e49ceb1803f27be5df457','Matt','matt@example.com','
secret','USER',true,'1980-07-03','2018-08-17 07:42:44.136','2018-08-17
07:42:44.137');

insert into person (id,name,email,password,role,enabled,birthday,created,
modified)
values ('4fe22e358d0e4e38b680eab91787f041','Mike','mike@example.com','s
ecret','ADMIN',true,'19820-08-05','2018-08-17 07:42:44.136','2018-08-17
07:42:44.137');

insert into person (id,name,email,password,role,enabled,birthday,created,
modified)
values ('84e6c4776dcc42369510c2692f129644','Dan','dan@example.com','se
cret','ADMIN',false,'1976-10-11','2018-08-17 07:42:44.136','2018-08-17
07:42:44.137');

insert into person (id,name,email,password,role,enabled,birthday,created,
modified)
values ('03a0c396acee4f6cb52e3964c0274495','Administrator','admin@exampl
e.com','admin','ADMIN',true,'1978-12-22','2018-08-17 07:42:44.136','2018-08-
17 07:42:44.137');

```

Now we are ready to use this application as an authentication and authorization mechanism. Run the Directory application. This app starts in port 8181. You can test it using either the browser and/or cURL command.

```

$ curl localhost:8181/persons/search/findByEmailIgnoreCase?email=mark@
example.com -u admin@example.com:admin
{
  "email" : "mark@example.com",
  "name" : "Mark",
  "password" : "secret",
  "role" : "USER",
  "enabled" : true,
  "birthday" : "1960-03-29",
  "created" : "2018-08-17T07:42:44.136",
  "modified" : "2018-08-17T07:42:44.137",
  "_links" : {

```

```

    "self" : {
      "href" : "http://localhost:8181/persons/dc952d19ccfc4164b5eb0338d14a6619"
    },
    "person" : {
      "href" : "http://localhost:8181/persons/dc952d19ccfc4164b5eb0338d14a6619"
    }
  }
}
}

```

From the command, you are getting the user, Mark, by providing the username/password of a person with an ADMIN role; in this case, using the `-u admin@example.com:admin` parameter.

Great! You are using JDBC to look up users by using Spring Data REST and Spring Security! You can leave this project running.

Using the Directory App within the ToDo App

It's time to integrate this Directory app with the ToDo app. And it is very easy.

Open your ToDo app and let's create a Person class. Yes, we are going to need a Person class that holds just enough information for authentication and authorization purposes. There is no need to have birth dates or any other information (see Listing 8-14).

Listing 8-14. `com.apress.todo.directory.Person.java`

```

package com.apress.todo.directory;

import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
import lombok.Data;

@Data
@JsonIgnoreProperties(ignoreUnknown = true)
public class Person {

    private String email;
    private String password;
    private String role;
    private boolean enabled;
}

```

Listing 8-14 shows the `Person` class. This class only has the necessary fields for the authentication and authorization process. It is important to mention that calling the `Directory` app returns a more complete JSON object. It must match to do the deserialization (from JSON to object using the Jackson library), but because there is no need for extra information, this class is using the `@JsonIgnoreProperties(ignoreUnknown=true)` annotation that helps match the fields needed. I think this is a nice way to decouple classes.

Note Some serialization tools in Java require the same class in the same package and implementing the `java.io.Serializable`, making it more difficult for developers and clients to manage and extend.

Next, create the `ToDoProperties` class that holds the information about the `Directory` app, like `Uri` (what is the address and base Uri), `Username`, and `Password` of the person that has the `ADMIN` role and has access to the REST API (see Listing 8-15).

Listing 8-15. `com.apress.todo.config.ToDoProperties.java`

```
package com.apress.todo.config;

import lombok.Data;
import org.springframework.boot.context.properties.ConfigurationProperties;

@Data
@ConfigurationProperties(prefix = "todo.authentication")
public class ToDoProperties {

    private String findByEmailUri;
    private String username;
    private String password;

}
```

Listing 8-15 shows the `ToDoProperties` class; note that the prefix is `todo.authentication.*`. Next, modify the `ToDoSecurityConfig` class. You can comment the whole class and copy the code in Listing 8-16.

Listing 8-16. com.apress.todo.config.ToDoSecurityConfig.java

```
package com.apress.todo.config;

import com.apress.todo.directory.Person;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.autoconfigure.security.servlet.PathRequest;
import org.springframework.boot.context.properties.
EnableConfigurationProperties;
import org.springframework.boot.web.client.RestTemplateBuilder;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.ParameterizedTypeReference;
import org.springframework.hateoas.MediaType;
import org.springframework.hateoas.Resource;
import org.springframework.http.HttpStatus;
import org.springframework.http.RequestEntity;
import org.springframework.http.ResponseEntity;
import org.springframework.security.config.annotation.authentication.
builders.AuthenticationManagerBuilder;
import org.springframework.security.config.annotation.web.builders.
HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.
WebSecurityConfigurerAdapter;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.
UsernameNotFoundException;
import org.springframework.security.crypto.factory.
PasswordEncoderFactories;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.web.util.matcher.AntPathRequestMatcher;
import org.springframework.web.client.RestTemplate;
import org.springframework.web.util.UriComponentsBuilder;
import java.net.URI;
```



```

@EnableConfigurationProperties(ToDoProperties.class)
@Configuration
public class ToDoSecurityConfig extends WebSecurityConfigurerAdapter {

    private final Logger log = LoggerFactory.getLogger(ToDoSecurityConfig.class);

    //Use this to connect to the Directory App
    private RestTemplate restTemplate;
    private ToDoProperties toDoProperties;
    private UriComponentsBuilder builder;

    public ToDoSecurityConfig(RestTemplateBuilder restTemplateBuilder,
        ToDoProperties toDoProperties){
        this.toDoProperties = toDoProperties;
        this.restTemplate = restTemplateBuilder.basicAuthorization(toDoProperties.getUsername(), toDoProperties.getPassword()).build();
    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.userDetailsService(new UserDetailsServiceImpl(){

            @Override
            public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

                try {
                    builder = UriComponentsBuilder
                    .fromUriString(toDoProperties.getFindByEmailUri())
                    .queryParams("email", username);

                    log.info("Querying: " + builder.toUriString());

                    ResponseEntity<Resource<Person>> responseEntity =
                    restTemplate.exchange(
                        RequestEntity.get(URI.create(builder.toUriString()))
                }
            }
        });
    }
}

```

```

        .accept(MediaType.HAL_JSON)
        .build()
        , new ParameterizedTypeReference<Resource<Person>>() {
        });

    if (responseEntity.getStatusCode() == HttpStatus.OK) {
        Resource<Person> resource = responseEntity.getBody();
        Person person = resource.getContent();

        PasswordEncoder encoder =
        PasswordEncoderFactories.createDelegatingPasswordEncoder();
        String password = encoder.encode(person.
        getPassword());

        return User
        .withUsername(person.getEmail())
        .password(password)
        .accountLocked(!person.isEnabled())
        .roles(person.getRole()).build();
    }

    }catch(Exception ex) {
        ex.printStackTrace();
    }
    throw new UsernameNotFoundException(username);
}
});
}

@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
        .requestMatchers(PathRequest.toStaticResources().
        atCommonLocations()).permitAll()
        .antMatchers("/", "/api/**").hasRole("USER")
        .and()

```

```

        .formLogin().loginPage("/login").permitAll()
        .and()
        .logout()
        .logoutRequestMatcher(new AntPathRequestMatcher("/logout"))
        .logoutSuccessUrl("/login")
        .and()
        .httpBasic();
    }
}

```

Listing 8-16 shows the new `ToDoSecurityConfig` class. Let's analyze it.

- `WebSecurityConfigurerAdapter`. This class overrides what we need to customize the security for the app; but you already knew that, right?
- `RestTemplate`. This helper class makes a REST call to the Directory app endpoint, in particular `/persons/search/findByEmailIgnoreCase` Uri.
- `UriComponentsBuilder`. Remember that the `/persons/search/findByEmailIgnoreCase` endpoint needs a parameter (`email`); that's the one provided by the `loadUserByUsername` method (`username`).
- `AuthenticationBuilder`. The authentication provides `userDetailsService`. In this code, there is an anonymous implementation of the `UserDetailsService` and the implementation of the `loadUserByUsername` method. This is where the `RestTemplate` is being used to make a call to the Directory app and the endpoint.
- `ResponseEntity`. Because the Directory app response is HAL+JSON, it is necessary to use a `ResponseEntity` that manages all the resources from the protocol. If there is `HttpStatus.OK`, it is easy to get the content as a `Person` instance and use it to create `UserDetails`.
- `antMatchers`. This class is configuring `HttpSecurity` as before, but this time it is including an `antMatchers` method that exposes the endpoints that are accessed by a valid person with a `USER` role.

We are reusing the same technique from the Directory app. `AuthenticationManager` is configured to provide a `UserDetails` instance by calling the directory service using `RestTemplate`. The Directory app responded with a HAL+JSON protocol, which is why it is necessary to use `ResponseEntity` to get the person as a resource.

Next, append the following `todo.authentication.*` properties in the application.properties file.

```
# ToDo - Directory integration
todo.authentication.find-by-email-uri=http://localhost:8181/persons/search/
findByEmailIgnoreCase
todo.authentication.username=admin@example.com
todo.authentication.password=admin
```

It is necessary to specify the complete Uri that searches for the email endpoint, and the person that has the ADMIN role.

Now you are ready to use the `ToDo` app. You can use the browser or the command line. Make sure that the Directory app is up and running. Run the `ToDo` app that runs in port 8080.

You can execute the following command in a terminal window.

```
$ curl localhost:8080/api/todos -u mark@example.com:secret
{
  "_embedded" : {
    "todos" : [ {
      "description" : "Read a Book",
      "created" : "2018-08-17T07:42:44.136",
      "modified" : "2018-08-17T07:42:44.137",
      "completed" : true,
      ...
      ...
    }
  ]
  "profile" : {
    "href" : "http://localhost:8080/api/profile/todos"
  }
}
```

Now you are authenticating and authorizing with Mark, who has the USER role. Congrats!! You integrated your own JDBC service with the `ToDo` application.

WebFlux Security

To add security to a WebFlux application, nothing changes. You need to add the `spring-boot-starter-security` dependency, and Spring Boot takes care of the rest with its auto-configuration. If you want to customize it as we did before, the only thing you need to do is use `ReactiveUserDetailsService` (instead of `UserDetailsService`) or use `ReactiveAuthenticationManager` (instead of `AuthenticationManager`). Remember that now you are working with Mono and Flux reactive stream types.

ToDo App with OAuth2

With Spring Boot and Spring Security, OAuth2 is easier than ever. In this section of this chapter, we are going to enter directly into the ToDo app with OAuth2. I assume that you know about OAuth2 and all the benefits of using it as a mechanism for authentication with third-party providers—like Google, Facebook, and GitHub—directly into your app.

So let's begin. Starting from scratch, go to your browser and open Spring Initializr. Add the following values to the fields.

- Group: `com.apress.todo`
- Artifact: `todo-oauth2`
- Name: `todo-oauth2`
- Package Name: `com.apress.todo`
- Dependencies: Web, Security, Lombok, JPA, REST Repositories, H2, MySQL

You can select either Maven or Gradle as the project type. Then you can press the Generate Project button; this downloads a ZIP file. Uncompress it and import the project in your favorite IDE (see Figure 8-8).

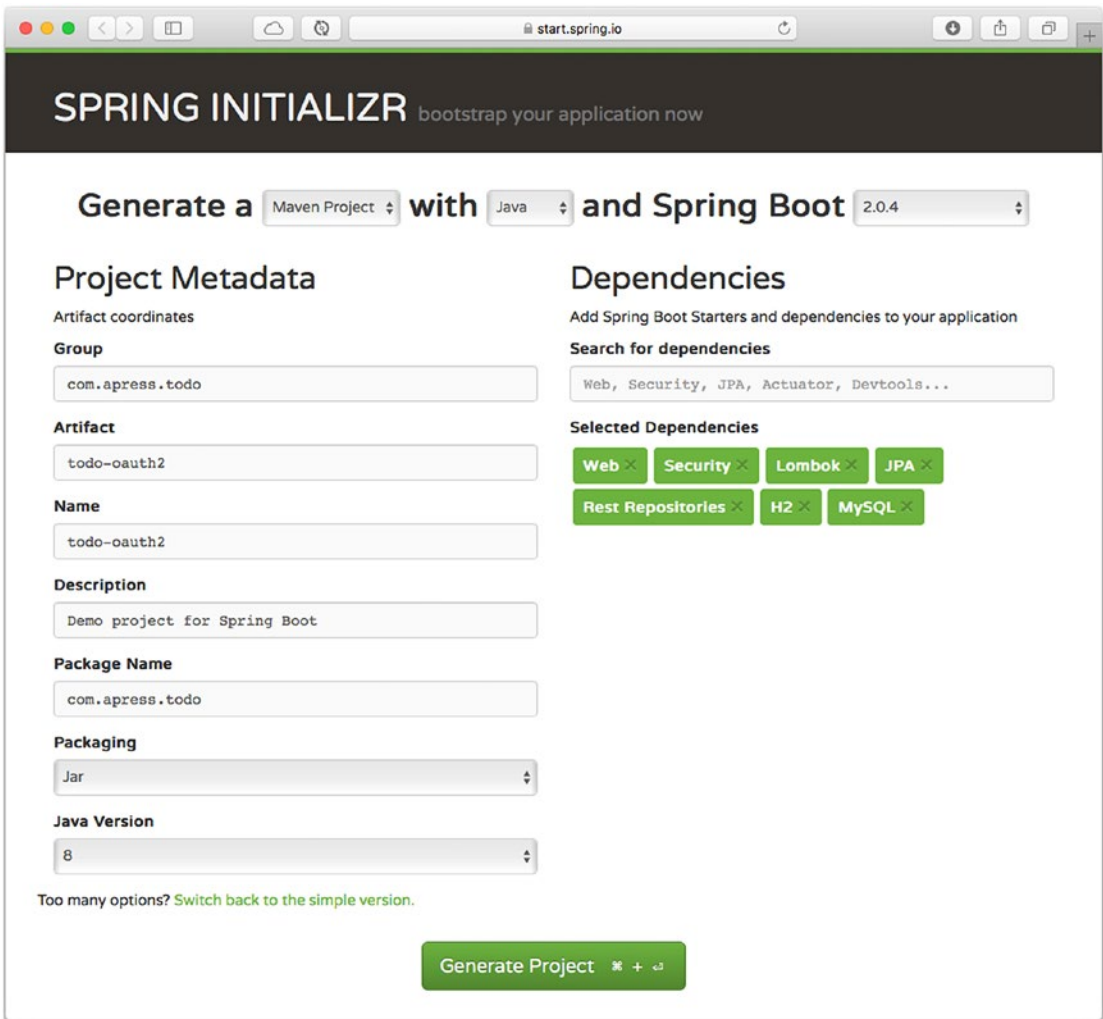


Figure 8-8. Spring Initializr

If you are using Maven, add the following dependencies to your pom.xml file.

```
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-oauth2-client</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-oauth2-jose</artifactId>
</dependency>
```

If you are using Gradle, add the following dependencies to your `build.gradle`:

```
compile('org.springframework.security:spring-security-oauth2-client')
compile('org.springframework.security:spring-security-oauth2-jose')
```

As you can imagine, when Spring Boot sees the `spring-security-oauth2-client`, it auto-configures all the necessary beans to use the OAuth2 security for the app. It's important to mention the need for the `spring-security-oauth2-jose` that contains the Spring Security's support for JOSE (JavaScript Object Signing and Encryption) framework. The JOSE framework is intended to provide a method to securely transfer claims between parties. It is built from a collection of specifications: JSON Web Token (JWT), JSON Web Signature (JWS), JSON Web Encryption (JWE), and JSON Web Key (JWK).

Next, you can reuse the `ToDo` class and the `ToDoRepository` interface (see Listings 8-17 and 8-18).

Listing 8-17. `com.apress.todo.domain.ToDo.java`

```
package com.apress.todo.domain;

import lombok.Data;
import org.hibernate.annotations.GenericGenerator;

import javax.persistence.*;
import javax.validation.constraints.NotBlank;
import javax.validation.constraints.NotNull;
import java.time.LocalDateTime;

@Entity
@Data
public class ToDo {

    @Id
    @GeneratedValue(generator = "system-uuid")
    @GenericGenerator(name = "system-uuid", strategy = "uuid")
    private String id;
    @NotNull
    @NotBlank
    private String description;

    @Column(insertable = true, updatable = false)
```

```

private LocalDateTime created;
private LocalDateTime modified;
private boolean completed;

public Todo(){
public Todo(String description){
    this.description = description;
}

@PrePersist
void onCreate() {
    this.setCreated(LocalDateTime.now());
    this.setModified(LocalDateTime.now());
}

@PreUpdate
void onUpdate() {
    this.setModified(LocalDateTime.now());
}
}

```

As you can see nothing changed. It remains the same.

Listing 8-18. `com.apress.todo.repository.ToDoRepository.java`

```

package com.apress.todo.repository;

import com.apress.todo.domain.ToDo;
import org.springframework.data.repository.CrudRepository;

public interface ToDoRepository extends CrudRepository<ToDo,String> { }

```

The same for this interface—nothing changed. Let’s review the application properties.

```

# JPA
spring.jpa.generate-ddl=true
spring.jpa.hibernate.ddl-auto=create-drop
spring.jpa.show-sql=true

```



```
# H2-Console: http://localhost:8080/h2-console
# jdbc:h2:mem:testdb
spring.h2.console.enabled=true
```

Nothing changed. Well, we are going to add more properties very soon.

Now comes the important part. You are going to use GitHub for OAuth2 authentication for the ToDo app.

Creating the ToDo App in GitHub

I'm assuming that you probably already have a GitHub account; if not, you can open a new one very easily at <https://github.com>. You can log in to your account and then open <https://github.com/settings/applications/new>. That's where you create the application. You can use the following values.

- Application name: todo-app
- Homepage URL: <http://localhost:8080>
- Application description: ToDo App
- Authorization callback URL: <http://localhost:8080/login/oauth2/code/github>

It's important to the authorization callback URL because this is how Spring Security's `OAuth2LoginAuthenticationFilter` expects to work with this endpoint pattern: `/login/oauth2/code/*`; of course, it is customizable by using the `redirect-uri-template` property (see Figure 8-9).

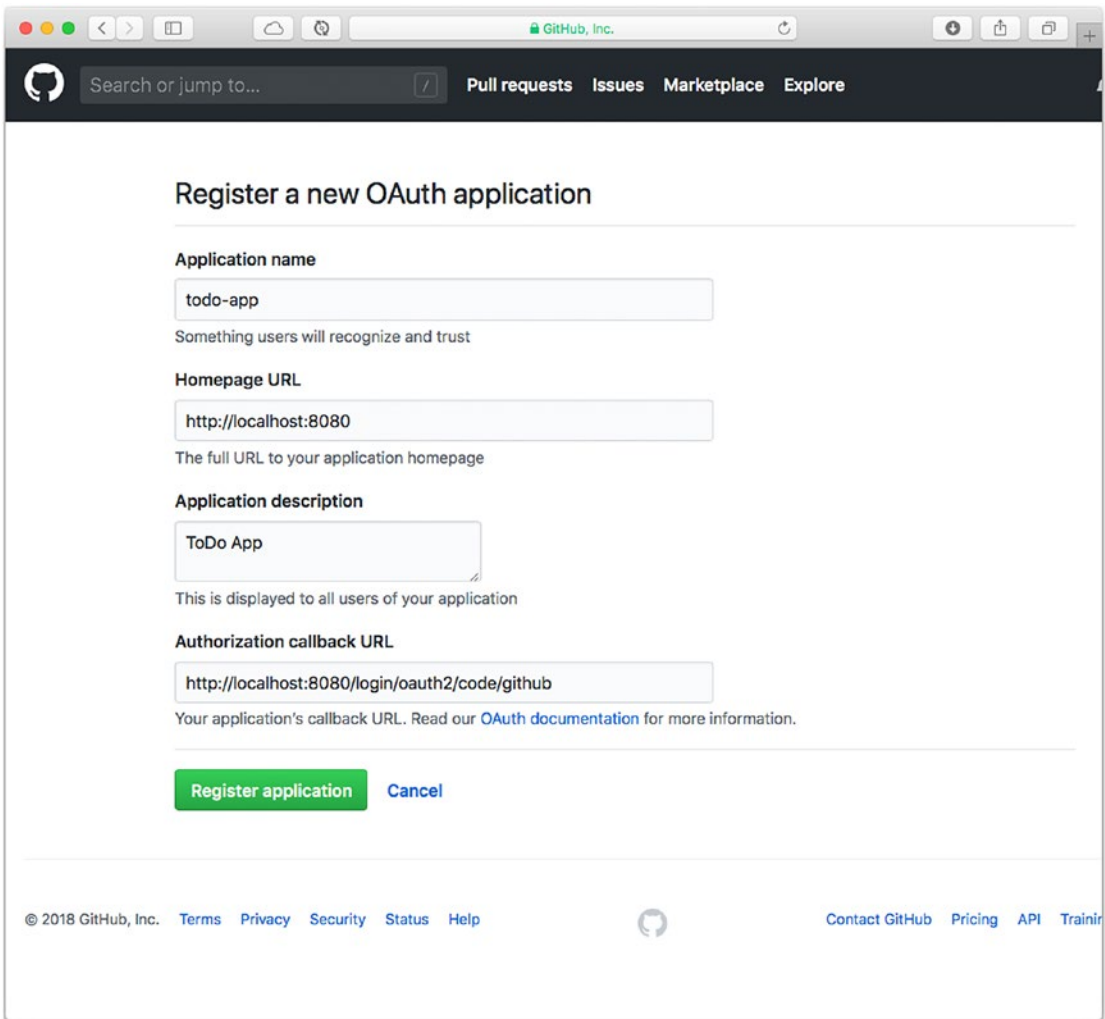


Figure 8-9. GitHub new app: <https://github.com/settings/applications/new>

You can click the Register application button. After this, GitHub creates the keys you need in your application (see Figure 8-10).

Application created successfully

Settings / Developer settings

OAuth Apps
GitHub Apps
Personal access tokens

todo-app

felipeg48 owns this application. [Transfer ownership](#)

You can list your application in the [GitHub Marketplace](#) so that other users can discover it. [List this application in the Marketplace](#)

0 users

Client ID
ac5b347117eb11705b70

Client Secret
44abe272a15834a5390423e53b58f57c35647a98

[Revoke all user tokens](#) [Reset client secret](#)

Figure 8-10. Client ID and client secret keys

Once you have this, copy the client id and client secret keys and append them to the application.properties with the `spring.security.oauth2.client.registration.*` keys.

```
# OAuth2
spring.security.oauth2.client.registration.todo.client-
id=ac5b347117eb11705b70
spring.security.oauth2.client.registration.todo.client-secret=44abe272a1583
4a5390423e53b58f57c35647a98
spring.security.oauth2.client.registration.todo.client-name=ToDo App with
GitHub Authentication
spring.security.oauth2.client.registration.todo.provider=github
spring.security.oauth2.client.registration.todo.scope=user
spring.security.oauth2.client.registration.todo.redirect-uri-
template=http://localhost:8080/login/oauth2/code/github
```

The `spring.security.oauth2.client.registration` accepts a map that contains the necessary keys like the `client-id` and `client-secret`.

That's it!! You don't need anything else. You can now run your application. Open the browser and point to `http://localhost:8080`. You get a link that redirects you to GitHub (see Figure 8-11).

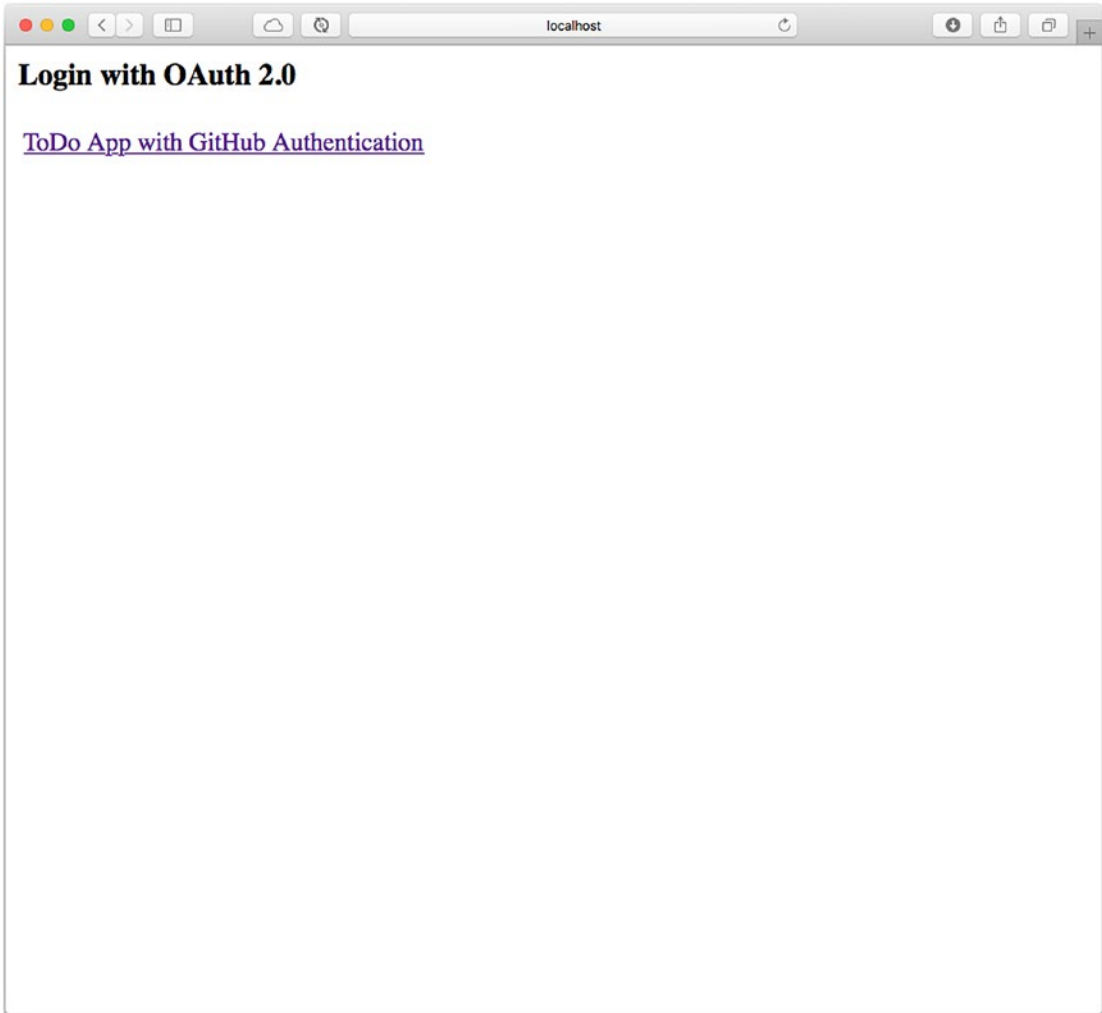


Figure 8-11. `http://localhost:8080`

You can click the link, which gets you through the login process but using a GitHub authentication mechanism (see Figure 8-12).

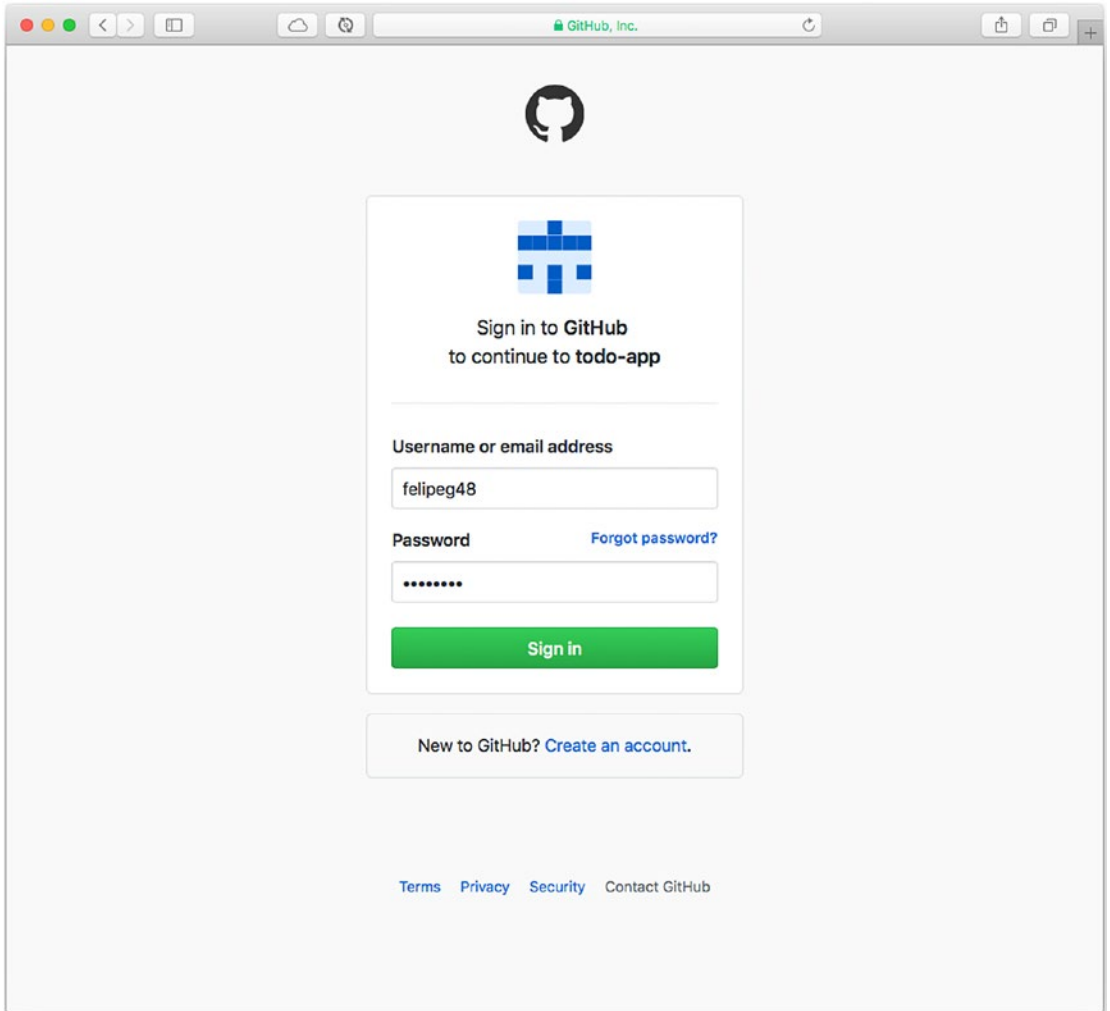


Figure 8-12. *GitHub authentication*

You can log in now with your credentials. Next, you are redirected to another page where you need to give permissions to the *todo-app* to use contact information (see Figure 8-13).

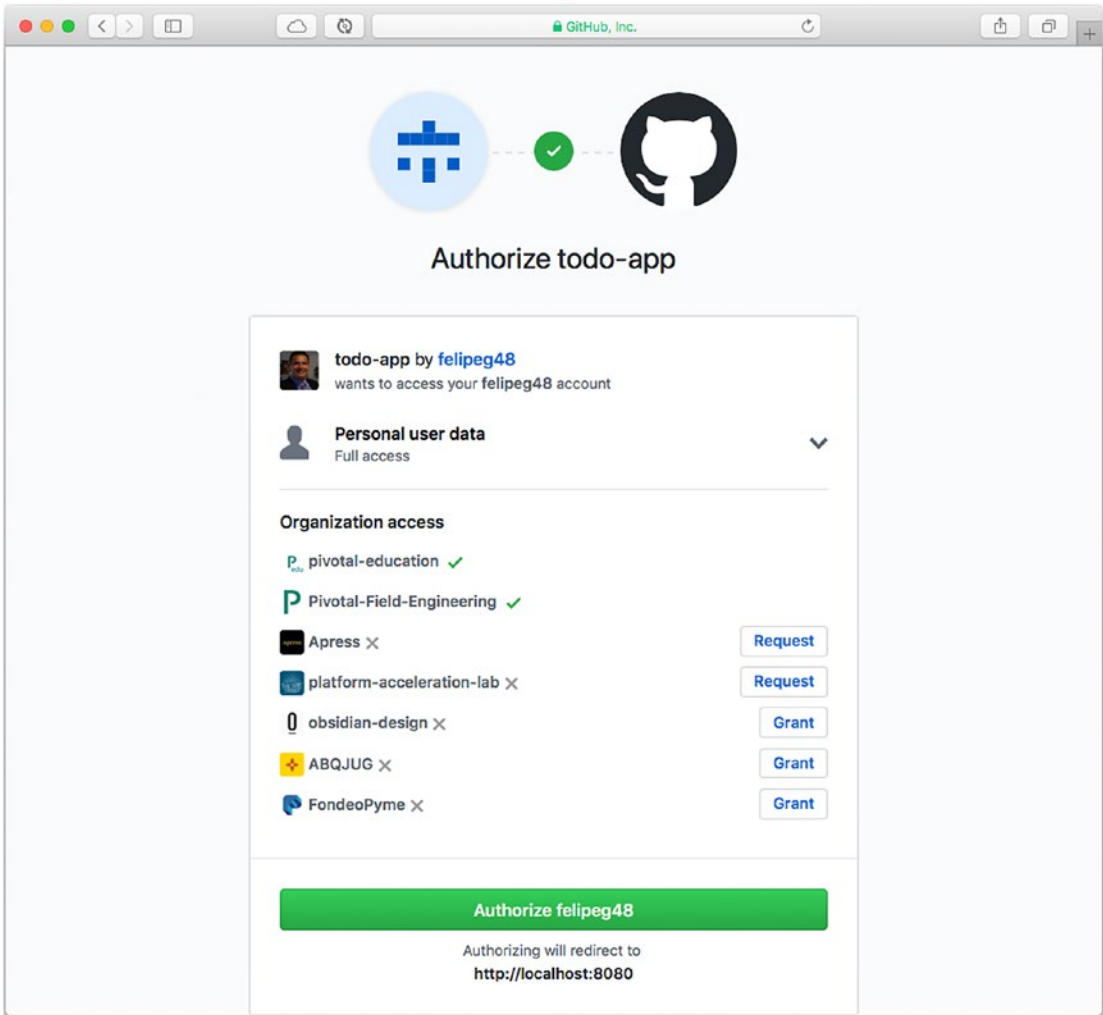


Figure 8-13. *GitHub authorization process*

You can then click the Authorize button to get back to your app with the ToDo REST API (see Figure 8-14).

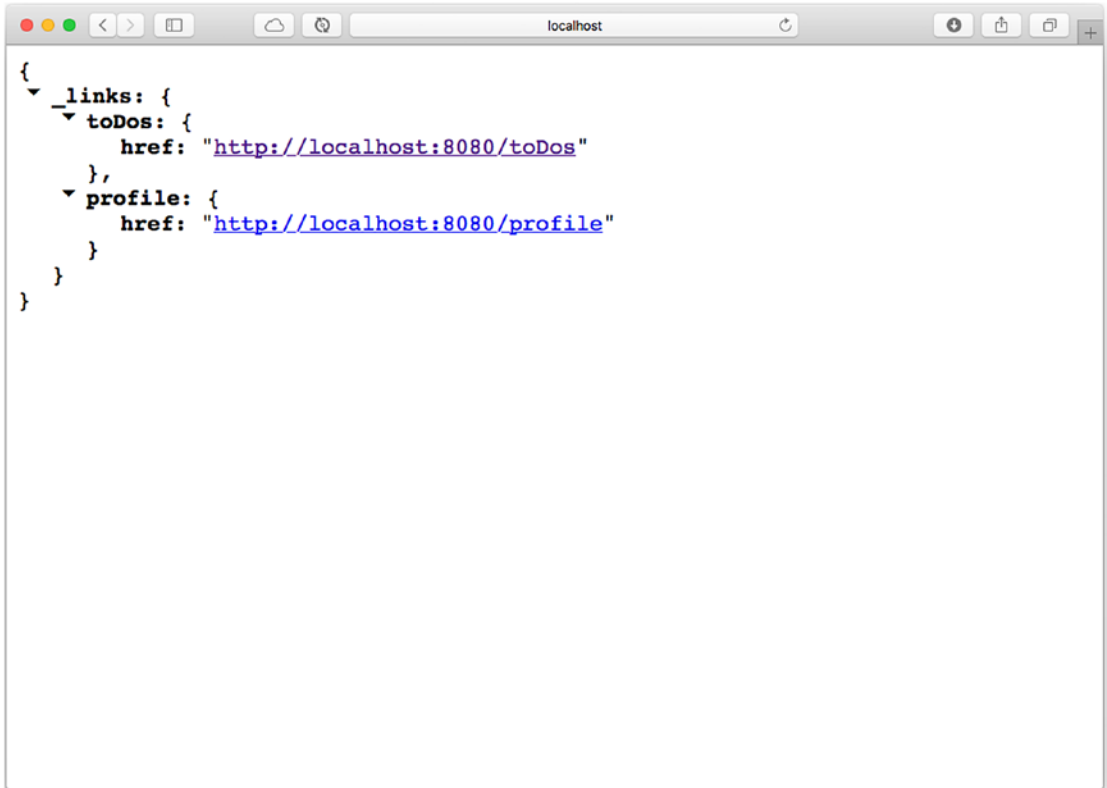


Figure 8-14. After GitHub authorization process

Congratulations!! Now you know how easy it is to integrate OAuth2 with different providers using Spring Boot and Spring Security.

Note You can find the solution to this section in the book source code on the Apress website or on GitHub at <https://github.com/Apress/pro-spring-boot-2>, or on my personal repository at <https://github.com/felipeg48/pro-spring-boot-2nd>.

Summary

In this chapter, you learned different ways to do security with Spring Boot. You learned how easy it is to secure an application by adding the `spring-boot-security-starter` dependency.

You also learned that it is easy to customize and override the defaults that Spring Boot offers you with Spring Security. You can use the `spring.security.*` properties or you can customize it with the `WebSecurityConfigurerAdapter` class.

You learned how to use JDBC and connect two applications, one of them acting as a security authority for authentication and authorization.

Lastly, you learned how easy it is to use OAuth2 with third-party authentication and authorization providers like Facebook, Google, GitHub, and more.

In the next chapter, we start working with messaging brokers.

CHAPTER 9

Messaging with Spring Boot

This chapter is all about messaging. It explains, with examples, how to use ActiveMQ for implementing the JMS (Java Message Service), RabbitMQ for implementing AMQP (Advanced Message Queuing Protocol), Redis for Pub/Sub, and WebSockets for implementing STOMP (Simple or Streaming Text-Oriented Message Protocol) with Spring Boot.

What Is Messaging?

Messaging is a way of communicating among one or more entities, and it is everywhere.

Computer messaging, in one form or another, has been around since the invention of the computer. It is defined as a method of communication between hardware and/or software components or applications. There is always a sender and one or more receivers. Messaging can be synchronous and asynchronous, pub/sub and peer-to-peer, RPC, enterprise-based, a message broker, ESB (enterprise service bus), MOM (message-oriented middleware), and so forth.

Messaging enables distributed communication that must be loosely coupled, meaning that it doesn't matter how or what message the sender is publishing, the receiver consumes the message without telling the sender.

Of course, there is a lot we could say about messaging—from the old techniques and technologies to new protocols and messaging patterns, but the intention of this chapter is to work with examples that illustrate how Spring Boot does messaging.

With this in mind, let's start creating examples using some of the technologies and message brokers out there.

JMS with Spring Boot

Let's start by using JMS. This is an old technology that is still being used by companies with legacy applications. JMS was created by Sun Microsystems to enable a way to send messages synchronously and asynchronously; it defines interfaces that need to be implemented by message brokers, such as WebLogic, IBM MQ, ActiveMQ, HornetQ, and so forth.

JMS is a Java-only technology, and so there have been attempts to create message bridges to combine JMS with other programming languages; still, it's difficult or very expensive to mix different technologies. I know that you are thinking that this is not true, because you can use Spring integration, Google Protobuffers, Apache Thrift, and other technologies to integrate JMS, but it's still a lot of work because you need to know and maintain code from all of these technologies.

ToDo App with JMS

Let's start by creating the ToDo App using JMS with Spring Boot. The idea is to send ToDo's to a JMS broker and receive and save them.

The Spring Boot team has several JMS starter poms available; in this case, you use ActiveMQ, which is an open source asynchronous messaging broker from the Apache Foundation (<http://activemq.apache.org>). One of the main advantages is that you can use either the in-memory broker or a remote broker. (You can download it and install it if you prefer; the code in this section uses the in-memory broker, but I tell you how to configure a remote broker).

You can open your favorite browser and point to the known Spring Initializr (<https://start.spring.io>); add the following values to the following fields.

- Group: `com.apress.todo`
- Artifact: `todo-jms`
- Name: `todo-jms`
- Package Name: `com.apress.todo`
- Dependencies: JMS (ActiveMQ), Web, Lombok, JPA, REST Repositories, H2, MySQL

You can select either Maven or Gradle as the project type. Then you can press the Generate Project button to download a ZIP file. Uncompress it and import the project in your favorite IDE (see Figure 9-1).

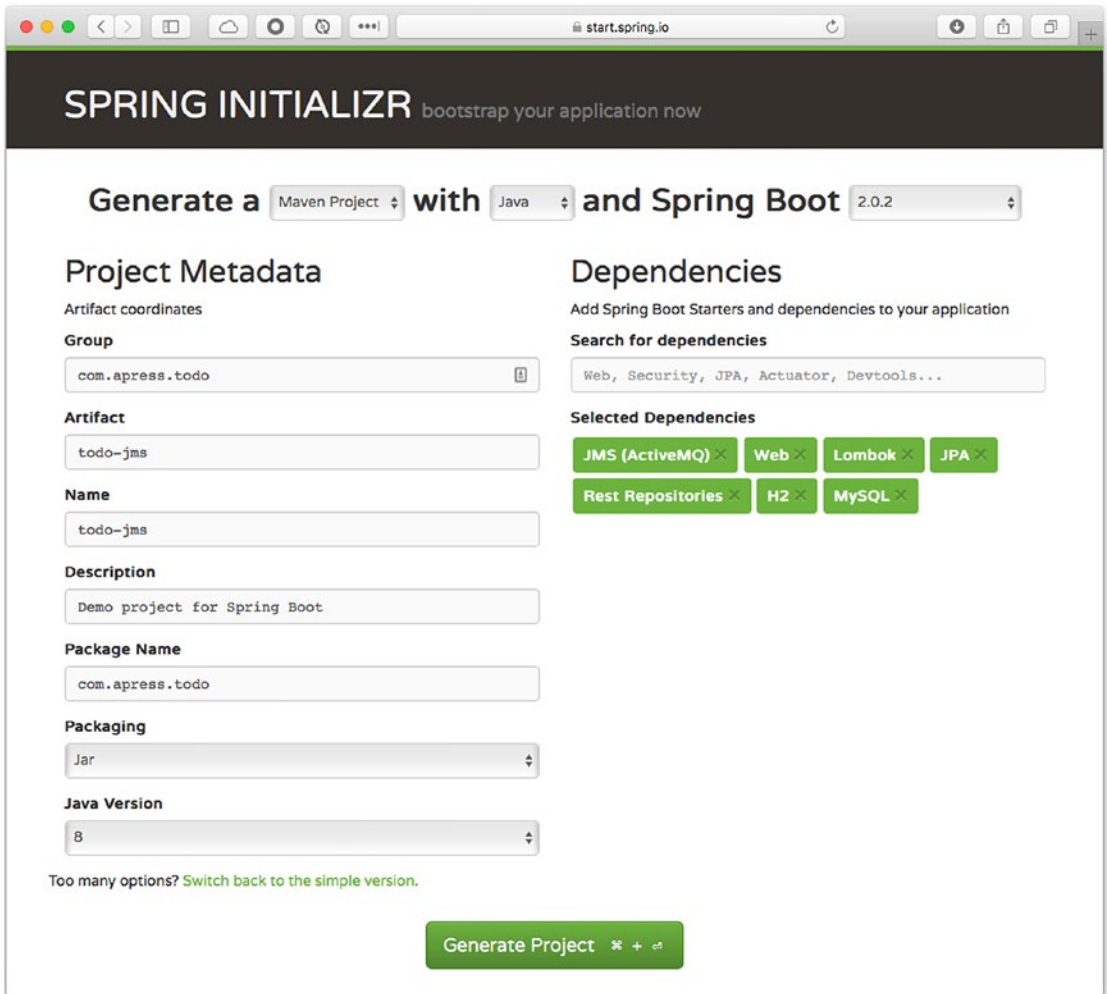


Figure 9-1. Spring Initializr

As you can see from the dependencies, you reuse the JPA and REST Repositories code from previous chapters. Instead of using Text message (a common approach for testing messaging) you use a `ToDo` instance, and it is converted as JSON format. To do this you required to add manually the next dependency to your `pom.xml` or `build.gradle`.

If you are using Maven, add the following dependency to your `pom.xml` file.

```
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
</dependency>
```

If you are using Gradle, add the following dependency to your `build.gradle` file.

```
compile("com.fasterxml.jackson.core:jackson-databind")
```

This dependency provides all the Jackson jars needed to use JSON to serialize the `ToDo` entity.

In the following sections I show you the important files, and how JMS is used in the `ToDo` app. The example uses the simple *Point-to-Point* pattern, where there is a *Producer*, a *Queue* and a *Consumer*. I'll show later on how to configure it for using a *Publisher-Subscriber* pattern with a *Producer*, a *Topic* and multiple *Consumers* later on.

ToDo Producer

Let's start by introducing the *Producer* that sends a `ToDo` to the ActiveMQ broker. This producer can be on its own project; it can be separated from the app; but for demonstration purposes you have the producer in the same code base, in the `ToDo` app.

Create the `ToDoProducer` class. This class sends a `ToDo` into a JMS Queue (see Listing 9-1).

Listing 9-1. `com.apress.todo.jms.ToDoProducer.java`

```
package com.apress.todo.jms;

import com.apress.todo.domain.ToDo;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.stereotype.Component;
```

@Component

```
public class ToDoProducer {
```

```

private static final Logger log = LoggerFactory.getLogger(ToDoProducer.
class);

private JmsTemplate jmsTemplate;

public ToDoProducer(JmsTemplate jmsTemplate){
    this.jmsTemplate = jmsTemplate;
}

public void sendTo(String destination, ToDo todo) {
    this.jmsTemplate.convertAndSend(destination, todo);
    log.info("Producer> Message Sent");
}
}

```

Listing 9-1 shows the producer class. This class is marked using the `@Component`, so it is registered as a Spring bean in the Spring application context. The `JmsTemplate` class is used, and it is very similar to other `*Template` classes that wrap all the boilerplate of the technology in use. The `JmsTemplate` instance is injected through the class constructor, and it is used to send a message using the `convertAndSend` method. You are sending a `ToDo` object (JSON String). This template has the mechanism to serialize it and send it to the ActiveMQ queue.

ToDo Consumer

Next, let's create the consumer class, which is listening for any incoming message from the ActiveMQ queue (see Listing 9-2).

Listing 9-2. `com.apress.todo.jms.ToDoConsumer.java`

```

package com.apress.todo.jms;

import com.apress.todo.domain.ToDo;
import com.apress.todo.repository.ToDoRepository;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.jms.annotation.JmsListener;
import org.springframework.stereotype.Component;

import javax.validation.Valid;

```

```

@Component
public class ToDoConsumer {

    private Logger log = LoggerFactory.getLogger(ToDoConsumer.class);

    private ToDoRepository repository;

    public ToDoConsumer(ToDoRepository repository){
        this.repository = repository;
    }

    @JmsListener(destination = "${todo.jms.destination}", containerFactory =
"jmsFactory")
    public void processToDo(@Valid ToDo todo){
        log.info("Consumer> " + todo);
        log.info("ToDo created> " + this.repository.save(todo));
    }
}

```

Listing 9-2 shows the consumer. In this class, you are using `ToDoRepository`, where it is listening for any message from the ActiveMQ queue. Make sure that you are using the `@JmsListener` annotation that makes the method process any incoming message from the queue; in this case, a valid `ToDo` (the `@Valid` annotation can be used to validate any field of the domain model). The `@JmsListener` annotation has two attributes. The `destination` attribute emphasizes the name of the queue/topic to connect to (the `destination` attribute evaluate the `todo.jms.destination` property, which you create /use in the next section). The `containerFactory` attribute is created as part of the configuration.

Configuring the ToDo App

Now, it is time to configure the `ToDo` App to send and receive `ToDo`'s. Listing 9-1 and Listing 9-2 show the `Producer` and `Consumer` classes, respectively. In both classes you are using a `ToDo` instance, meaning that it is necessary to do serialization. Most of the Java frameworks that work with serialization require that your classes implement from `java.io.Serializable`. It is an easy way to convert those classes into bytes, but this approach has been debated for years because implementing `Serializable` decreases the flexibility to modify a class's implementation once it's released for usage.

The Spring Framework offers an alternative way to do serialization without implementing `Serializable`—through a `MessageConverter` interface. This interface offers the `toMessage` and `fromMessage` methods, in which you can plug in whatever technology fits for object conversion.

Let's create a configuration that uses a `ToDo` instance for the producer and consumer (see Listing 9-3).

Listing 9-3. `com.apress.todo.config.ToDoConfig.java`

```
package com.apress.todo.config;
import com.apress.todo.error.ToDoErrorHandler;
import com.apress.todo.validator.ToDoValidator;
import org.springframework.boot.autoconfigure.jms.
DefaultJmsListenerContainerFactoryConfigurer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.jms.annotation.JmsListenerConfigurer;
import org.springframework.jms.config.DefaultJmsListenerContainerFactory;
import org.springframework.jms.config.JmsListenerContainerFactory;
import org.springframework.jms.config.JmsListenerEndpointRegistrar;
import org.springframework.jms.support.converter.
MappingJackson2MessageConverter;
import org.springframework.jms.support.converter.MessageConverter;
import org.springframework.jms.support.converter.MessageType;
import org.springframework.messaging.handler.annotation.support.
DefaultMessageHandlerMethodFactory;

import javax.jms.ConnectionFactory;
```

@Configuration

```
public class ToDoConfig {

    @Bean
    public MessageConverter jacksonJmsMessageConverter() {
        MappingJackson2MessageConverter converter = new
        MappingJackson2MessageConverter();
        converter.setTargetType(MessageType.TEXT);
    }
}
```

```

        converter.setTypeIdPropertyName("_class_");
        return converter;
    }

    @Bean
    public JmsListenerContainerFactory<?> jmsFactory(ConnectionFactory
    connectionFactory,
                                                    DefaultJmsListener
    ContainerFactory
    Configurer configurer) {
        DefaultJmsListenerContainerFactory factory = new
    DefaultJmsListenerContainerFactory();
        factory.setErrorHandler(new ToDoErrorHandler());
        configurer.configure(factory, connectionFactory);
        return factory;
    }

    @Configuration
    static class MethodListenerConfig implements JmsListenerConfigurer{

        @Override
        public void configureJmsListeners (JmsListenerEndpointRegistrar
    jmsListenerEndpointRegistrar){
            jmsListenerEndpointRegistrar.setMessageHandlerMethodFactory
            (myHandlerMethodFactory());
        }

        @Bean
        public DefaultMessageHandlerMethodFactory myHandlerMethodFactory ()
    {
            DefaultMessageHandlerMethodFactory factory = new
            DefaultMessageHandlerMethodFactory();
            factory.setValidator(new ToDoValidator());
            return factory;
        }
    }
}

```


Listing 9-3 shows the `ToDoConfig` class that is use for the app. Let's analyze it.

- `@Configuration`. This is a known annotation that marks the class for configuring the `SpringApplication` context.
- `MessageConverter`. The method `jacksonJmsMessageConverter` returns the `MessageConverter` interface. This interface promotes the implementation of `toMessage` and `fromMessage` method that helps plug in any serialization/conversion that you want to use. In this case, you are using a JSON converter by using the `MappingJackson2MessageConverter` class implementation. This class is one of the default implementations that you can find in the Spring Framework. It uses the Jackson libraries, which use mappers to convert to/from JSON to/from an object. Because you are working with `ToDo` instances, it is necessary to specify a target type (`setTargetType`), this means that the JSON object is taken as text and a type-id property name (`setTypeIdPropertyName`) that identifies a property found from/to the producer and consumer. The type-id property name must always match both the producer and consumer. It can be any value you need (preferably something that you recognize because it is used to set the name (including the package) of the class to be converted to/from JSON); in other words, the `com.apress.todo.domain.TODO` class must be shared between the producer and consumer so that the mapper knows where to get the class to/from.
- `JmsListenerContainerFactory`. The `jmsFactory` method returns `JmsListenerContainerFactory`. This bean requires `ConnectionFactory` and `DefaultJmsListenerContainerFactoryConfigurer` (both is injected by the Spring), and it creates `DefaultJmsListenerContainerFactory`, which sets up an error handler. This bean is used in the `@JmsListener` annotation by setting the `containerFactory` attribute.
- `JmsListenerConfigurer`. In this class, you are creating a static configuration. The `MethodListenerConfig` class implements the `JmsListenerConfigurer` interface. This interface requires you to register a bean that has the configuration of the validator (`ToDoValidator` class); in this case, the `DefaultMessageHandlerMethodFactory` bean.

If you don't want to validate yet, you can remove the `MethodListenerConfig` class and the `setErrorHandler` call from the `jmsFactory` bean declaration; but if you want to experiment with validation, then you need to create the `ToDoValidator` class (see Listing 9-4).

Listing 9-4. `com.apress.todo.validator.ToDoValidator.java`

```
package com.apress.todo.validator;

import com.apress.todo.domain.ToDo;
import org.springframework.validation.Errors;
import org.springframework.validation.Validator;

public class ToDoValidator implements Validator {
    @Override
    public boolean supports(Class<?> clazz) {
        return clazz.isAssignableFrom(ToDo.class);
    }

    @Override
    public void validate(Object target, Errors errors) {
        ToDo todo = (ToDo)target;

        if (todo == null) {
            errors.reject(null, "ToDo cannot be null");
        }else {
            if (todo.getDescription() == null || todo.getDescription().
                isEmpty())
                errors.rejectValue("description",null,"description cannot
                be null or empty");
        }
    }
}
```

Listing 9-4 shows the validator class that is called for every message, and validates that the description field is not empty or null. This class is implementing the `Validator` interface and implements the `supports` and `validate` methods.

This is the `ToDoErrorHandler` code.

```
package com.apress.todo.error;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.util.ErrorHandler;

public class ToDoErrorHandler implements ErrorHandler {
    private static Logger log = LoggerFactory.getLogger(ToDoErrorHandler.
        class);

    @Override
    public void handleError(Throwable t) {
        log.warn("ToDo error...");
        log.error(t.getCause().getMessage());
    }
}
```

As you can see, this class is implementing the `ErrorHandler` interface.

Now, let's create the `ToDoProperties` class that holds the `todo.jms.destination` property, which indicates which queue/topic to connect to (see Listing 9-5).

Listing 9-5. `com.apress.todo.config.ToDoProperties.java`

```
package com.apress.todo.config;

import lombok.Data;
import org.springframework.boot.context.properties.ConfigurationProperties;

@Data
@ConfigurationProperties(prefix = "todo.jms")
public class ToDoProperties {
    private String destination;
}
```

Listing 9-5 shows the `ToDoProperties` class. Remember that in Listing 9-2 (the `ToDoConsumer` class), the `processToDo` method was marked with the `@JmsListener` annotation, which exposed the `destination` attribute. This attribute gets its value from evaluating the *SpEL* (Spring Expression Language) `${todo.jms.destination}` expression that you are defining within this class.

You can set this property in the `application.properties` file.

src/main/resources/application.properties

```
# JPA
spring.jpa.generate-ddl=true
spring.jpa.hibernate.ddl-auto=create-drop

# ToDo JMS
todo.jms.destination=todoDestination
```

Running the ToDo App

Next, let's create a config class that sends a message to the Queue using the producer (see Listing 9-6).

Listing 9-6. `com.apress.todo.config.ToDoSender.java`

```
package com.apress.todo.config;

import com.apress.todo.domain.ToDo;
import com.apress.todo.jms.ToDoProducer;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.CommandLineRunner;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class ToDoSender {

    @Bean
    public CommandLineRunner sendTodos(@Value("${todo.jms.destination}")
    String destination, ToDoProducer producer){
```

```

        return args -> {
            producer.sendTo(destination,new ToDo("workout tomorrow morning!"));
        };
    }
}

```

Listing 9-6 shows the config class that sends the message using a `ToDoProducer` instance and the destination (from the `todo.jms.destination` property).

To run the app, you can either use your IDE (if you imported it) or you can use the Maven wrapper.

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

Or the Gradle wrapper.

```
./gradlew bootRun
```

You should get the following text from the logs.

```

Producer> Message Sent
Consumer> ToDo(id=null, description=workout tomorrow morning!,
created=null, modified=null, completed=false)
ToDo created> ToDo(id=8a808087645bd67001645bd6785b0000, description=workout
tomorrow morning!, created=2018-07-02T10:32:19.546, modified=2018-07-
02T10:32:19.547, completed=false)

```

You can take a look at <http://localhost:8080/todos> and see the `ToDo` created.

Using JMS Pub/Sub

If you want to use the Pub/Sub pattern, where you want to have multiple consumers receiving a message (by using a topic for subscription), I'll explain what you need to do in your app.

Because we are using Spring Boot, this makes it easier to configure a Pub/Sub pattern. If you are using default listener (a `@JmsListener(destination)` default listener container), then you can use the `spring.jms.pub-sub-domain=true` property in the `application.properties` file.

But if you are using a custom listener container, then you can set it programmatically.

@Bean

```
public DefaultMessageListenerContainer.jmsListenerContainerFactory() {
    DefaultMessageListenerContainer dmlc = new
    DefaultMessageListenerContainer();
    dmlc.setPubSubDomain(true);
    // Other configuration here ...
    return dmlc;
}
```

Remote ActiveMQ

The ToDo App is using the in-memory broker (`spring.activemq.in-memory=true`). This is probably good for demos or testing, but in reality, you use a remote ActiveMQ server. If you required a remote server, add the following keys to your `application.properties` file (modify it accordingly).

src/main/resources/application.properties

```
spring.activemq.broker-url=tcp://my-awesome-server.com:61616
spring.activemq.user=admin
spring.activemq.password=admin
```

There are many more properties that you can use for the ActiveMQ broker. Go to <https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html> and look for the `spring.activemq.*` keys.

RabbitMQ with Spring Boot

Since the first attempts with JMS by companies like Sun, Oracle, and IBM, and Microsoft with MSMQ, the protocols used were proprietary. JMS defines an Interface API, but trying to mix technologies or programming languages is a hassle. Thanks to a JPMorgan team, the AMQP (Advance Message Queuing Protocol) was created. It's an open standard application layer for MOM. In other words, AMQP is a wire-level protocol, meaning that you can use any technology or programming language with this protocol.

Messaging brokers compete with each other to prove that they are robust, reliable, and scalable, but the most important issue is how fast they are. I've been working with a lot of brokers, and so far one of the easiest to use and to scale, and the fastest, is RabbitMQ, which implements the AMQP protocol.

It would take an entire book to describe each part of RabbitMQ and all the concepts around it, but I'll try to explain some of them based on this section's example.

Installing RabbitMQ

Before I talk about RabbitMQ, let's install it. If you are using Mac OS X/Linux, you can use the brew command.

```
$ brew upgrade  
$ brew install rabbitmq
```

If you are using a UNIX or a Windows system, you can go to the RabbitMQ web site and use the installers (www.rabbitmq.com/download.html). RabbitMQ is written in Erlang, so its major dependency is to install the Erlang runtime in your system. Nowadays, all the RabbitMQ installers come with all the Erlang dependencies. Make sure the executables are in your PATH variable (for Windows and Linux, depending of what OS you are using). If you are using brew, you don't need to worry about setting the PATH variable.

RabbitMQ/AMQP: Exchanges, Bindings, and Queues

The AMQP defines three concepts that are a little different from the JMS world, but very easy to understand. AMQP defines *exchanges*, which are entities where the messages are sent. Every *exchange* takes a message and routes it to zero or more *queues*. This routing involves an algorithm that is based on the exchange type and rules, called *bindings*.

The AMPQ protocol defines five exchange types: *Direct*, *Fanout*, *Topic*, and *Headers*. Figure 9-2 shows these different exchange types.

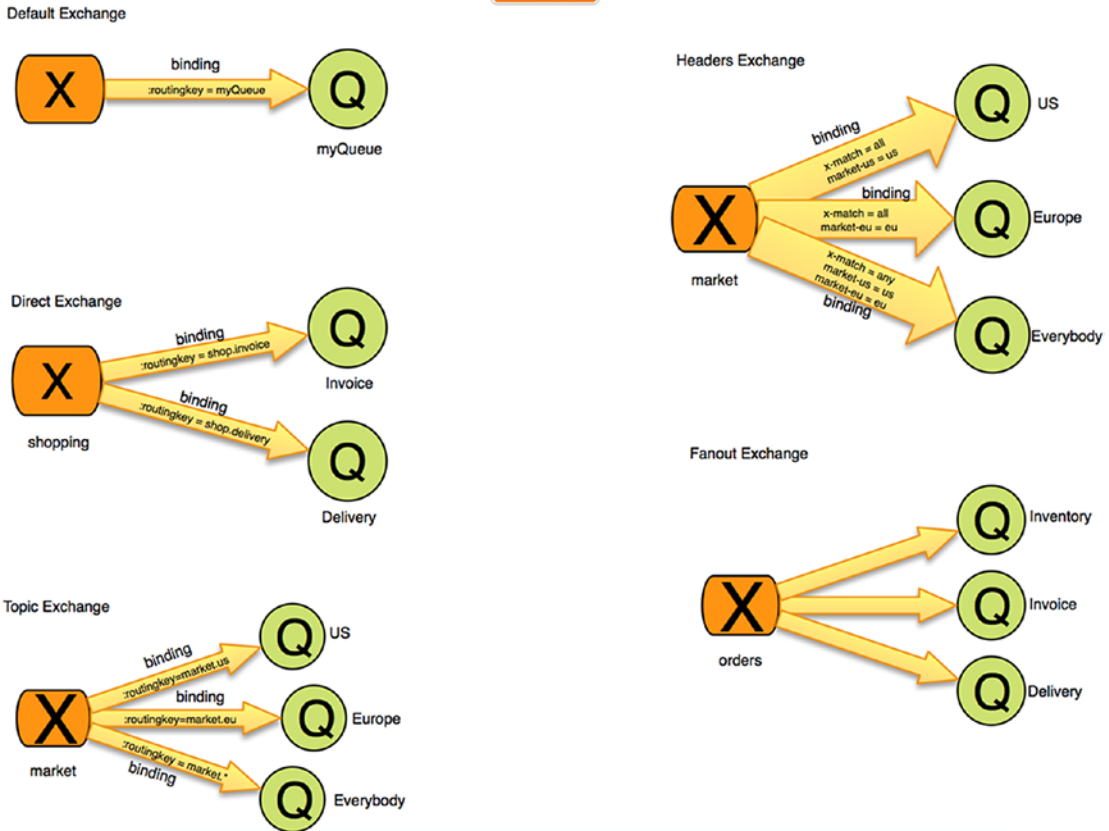


Figure 9-2. AMQP exchanges/bindings/queues

Figure 9-2 shows the possible exchange types. So, the main idea is to send a message to an exchange, including a routing key, then the exchange based on its type deliver the message to the queue (or it won't if the routing key doesn't match).

The *default exchange* is bound automatically to every queue created. The *direct exchange* is bound to a queue by a routing key; you can see this exchange type as one-to-one binding. The *topic exchange* is similar to the direct exchange; the only difference is that in its binding, you can add a wildcard into its routing key. The *headers exchange* is similar to the topic exchange; the only difference is that the binding is based on the message headers (this is a very powerful exchange, and you can do *all* and *any* expressions for its headers). The *fanout exchange* copy the message to all the bound queues; you can see this exchange as a message broadcast.

You can get more information about these topics at www.rabbitmq.com/tutorials/amqp-concepts.html.

The example in this section uses the default exchange type, which means that the routing key is equal to the name of the queue. Every time you create a queue, RabbitMQ creates a binding from the default exchange (the actual name is an empty string) to the queue using the queue's name.

ToDo App with RabbitMQ

Let's retake the ToDo app and add an AMQP messaging. The same as with the previous app, you work with ToDo instances. You send and receive a JSON message and convert it to object.

Let's start by opening your favorite browser and point to the known Spring Initializr. Add the following values to the fields.

- Group: `com.apress.todo`
- Artifact: `todo-rabbitmq`
- Name: `todo-rabbitmq`
- Package Name: `com.apress.todo`
- Dependencies: RabbitMQ, Web, Lombok, JPA, REST Repositories, H2, MySQL

You can select either Maven or Gradle as the project type. Then you can press the Generate Project button, which downloads a ZIP file. Uncompress it and import the project in your favorite IDE (see Figure 9-3).

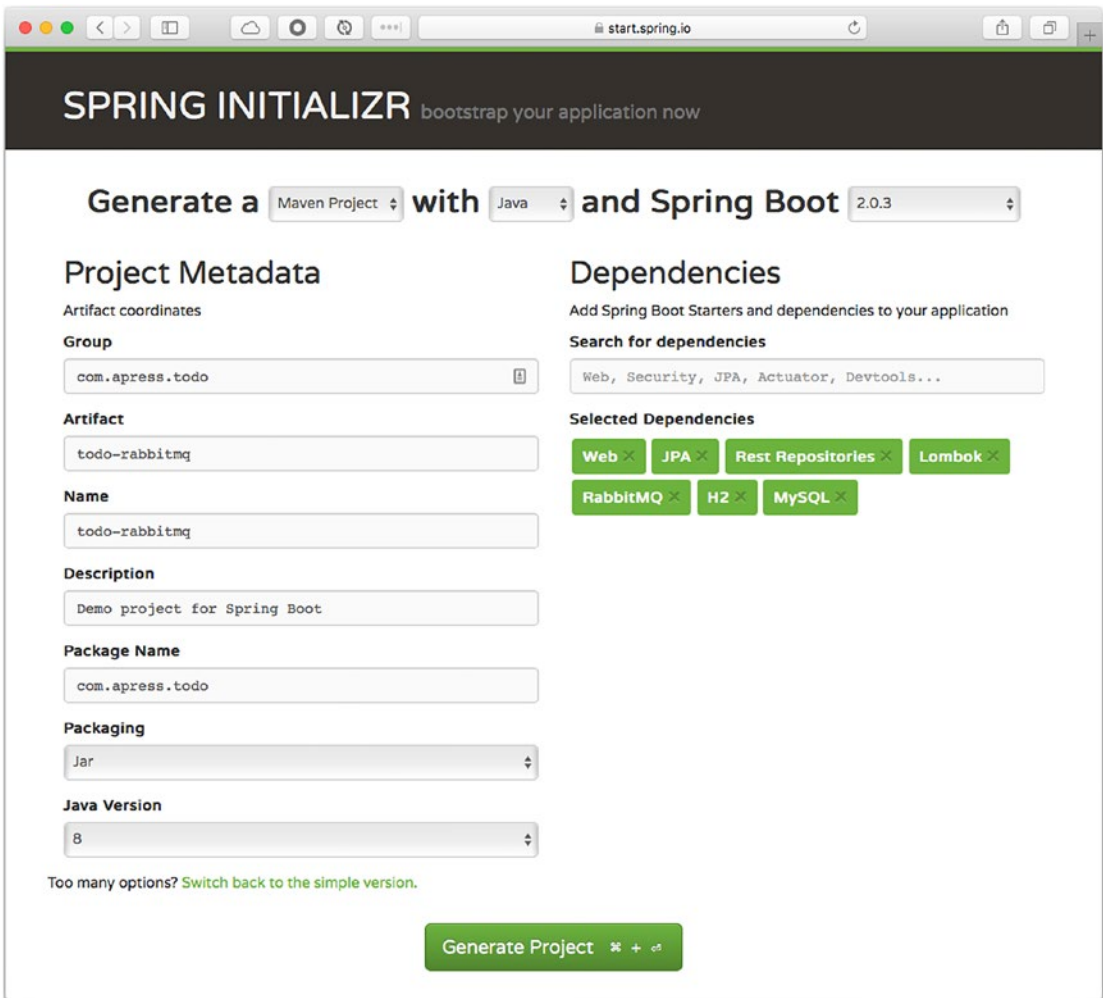


Figure 9-3. Spring Initializr <https://start.spring.io>

You can copy/paste the code of the JPA/REST project from previous chapters.

ToDo Producer

Lets' start by creating a Producer class that sends messages to Exchange (Default Exchange—direct) (see Listing 9-7).

Listing 9-7. com.apress.todo.rmq.ToDoProducer.java

```

package com.apress.todo.rmq;

import com.apress.todo.domain.ToDo;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.stereotype.Component;

@Component
public class ToDoProducer {

    private static final Logger log = LoggerFactory.getLogger(ToDoProducer.
class);
    private RabbitTemplate template;

    public ToDoProducer(RabbitTemplate template){
        this.template = template;
    }

    public void sendTo(String queue, ToDo todo){
        this.template.convertAndSend(queue, todo);
        log.info("Producer> Message Sent");
    }
}

```

Listing 9-7 shows the `ToDoProducer.java` class. Let's examine it.

- `@Component`. This annotation marks the class to be picked up by the Spring container.
- `RabbitTemplate`. The `RabbitTemplate` is a helper class that simplifies synchronous/asynchronous access to RabbitMQ for sending and/or receiving messages. This is very similar to the `JmsTemplate` you saw earlier.

- `sendTo(routingKey,message)`. This method has the routing key and the message as parameters. In this case, the routing key is the name of the queue. This method uses the `rabbitTemplate` instance to call the `convertAndSend` method that accepts the routing key and the message. Remember that the message is sent to the exchange (the default exchange) and the exchange routes the message to the right queue. This routing key happens to be the name of the queue. Also remember that by default RabbitMQ always binds the default exchange (Direct Exchange) to a queue and the routing key is the queue's name.

ToDo Consumer

Next, it's time to create the Consumer class that is listening to the specified queue (see Listing 9-8).

Listing 9-8. `com.apress.todo.rmq.ToDoConsumer.java`

```
package com.apress.todo.rmq;

import com.apress.todo.domain.ToDo;
import com.apress.todo.repository.ToDoRepository;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.stereotype.Component;

@Component
public class ToDoConsumer {

    private Logger log = LoggerFactory.getLogger(ToDoConsumer.class);
    private ToDoRepository repository;

    public ToDoConsumer(ToDoRepository repository){
        this.repository = repository;
    }

    @RabbitListener(queues = "${todo.amqp.queue}")
    public void processToDo(ToDo todo){
```

```

    log.info("Consumer> " + todo);
    log.info("ToDo created> " + this.repository.save(todo));
}
}

```

Listing 9-8 shows the `ToDoConsumer.java` class. Let's examine it.

- `@Component`. You already know this annotation. It marks the class to be picked up by the Spring container.
- `@RabbitListener`. This annotation marks the method (because you can use this annotation in a class as well) for creating a handler for any incoming messages, meaning that it creates a listener that is connected to the RabbitMQ's queue and passes that message to the method. Behind the scenes, the listener does its best to convert the message to the appropriate type by using the right message converter (an implementation of the `org.springframework.amqp.support.converter.MessageConverter` interface. This interface belongs to the `spring-amqp` project); in this case, it converts from JSON to a `ToDo` instance.

As you can see from the `ToDoProducer` and `ToDoConsumer`, the code is very simple. If you created this by only using the RabbitMQ Java client (www.rabbitmq.com/java-client.html), at least you need more lines of code to create a connection, a channel, and a message and send the message, or if you are writing a consumer, then you need to open a connection, create a channel, create a basic consumer, and get into a loop for processing every incoming message. This is a lot for simple producers or consumers. That's why the Spring AMQP team created this simple way to do a heavy task in a few lines of code.

Configuring the ToDo App

Next let's configure the app. Remember that you are sending `ToDo` instances, so practically, it is kind of the same configuration that we did with JMS. We need to set our converter and the listener container (see Listing 9-9).

Listing 9-9. com.apress.todo.config.ToDoConfig.java

```

package com.apress.todo.config;

import org.springframework.amqp.core.Queue;
import org.springframework.amqp.rabbit.config.
SimpleRabbitListenerContainerFactory;
import org.springframework.amqp.rabbit.connection.ConnectionFactory;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.amqp.support.converter.
Jackson2JsonMessageConverter;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class ToDoConfig {

    @Bean
    public SimpleRabbitListenerContainerFactory rabbitListenerContainer
Factory(ConnectionFactory connectionFactory) {
        SimpleRabbitListenerContainerFactory factory = new
SimpleRabbitListenerContainerFactory();
        factory.setConnectionFactory(connectionFactory);
        factory.setMessageConverter(new Jackson2JsonMessageConverter());
        return factory;
    }

    @Bean
    public RabbitTemplate rabbitTemplate(ConnectionFactory connectionFactory){
        RabbitTemplate template = new RabbitTemplate(connectionFactory);
        template.setMessageConverter(new Jackson2JsonMessageConverter());
        return template;
    }
}

```

```

@Bean
public Queue queueCreation(@Value("${todo.amqp.queue}") String queue){
    return new Queue(queue,true,false,false);
}
}

```

Listing 9-9 shows you the configuration. It has several bean definitions; let's examine them.

- `SimpleRabbitListenerContainerFactory`. This factory is required when using the `@RabbitListener` annotation for custom setup because you are working with `ToDo` instances; it is necessary set the message converter.
- `Jackson2JsonMessageConverter`. This converter is used for producing (with the `RabbitTemplate`) and for consuming (`@RabbitListener`); it uses the Jackson libraries for doing its mapping and conversion.
- `RabbitTemplate`. This a helper class that can send and receive messages. In this case, it is necessary to customize it to produce JSON objects using the Jackson converter.
- `Queue`. You can manually create a queue, but in this case, you are creating it programmatically. You pass the name of the queue, if is going to be durable or exclusive, and auto-delete.

Remember that in the AMQP protocol, you need an exchange that is bound to a queue, so this particular example creates at runtime a queue named `spring-boot`, and by default, all the queues are bound to a default exchange. That's why you didn't provide any information about an exchange. So, when the producer sends the message, it is sent first to the default exchange and then routed to the queue (`spring-boot`).

Running the ToDo App

Let's create the sender class that sends the ToDo message (see Listing 9-10).

Listing 9-10. `com.apress.todo.config.ToDoSender.java`

```
package com.apress.todo.config;

import com.apress.todo.domain.ToDo;
import com.apress.todo.rmq.ToDoProducer;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.CommandLineRunner;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class ToDoSender {

    @Bean
    public CommandLineRunner sendToDos(@Value("${todo.amqp.queue}") String
    destination, ToDoProducer producer){
        return args -> {
            producer.sendTo(destination,new ToDo("workout tomorrow morning!"));
        };
    }
}
```

Add the following keys (that declare the queue to send-to/consume-from) to your `application.properties` file.

```
todo.amqp.queue=spring-boot
```

Before you run your example, make sure your RabbitMQ server is up and running. You can start it by opening a terminal and executing the following command.

```
$ rabbitmq-server
```

Make sure that you have access to the RabbitMQ web console by going to `http://localhost:15672/` with `guest/guest` credentials. If you have problems accessing the web console, make sure that you have the Management plugin enabled by running the following command.


```
$ rabbitmq-plugins list
```

If the entire list has the boxes unchecked, then the Management plugin is not enabled yet (normally happens with fresh installations). To enable this plugin, you can execute the following command.

```
$ rabbitmq-plugins enable rabbitmq_management --online
```

Now, you can try again. You should then see a web console similar to Figure 9-4.

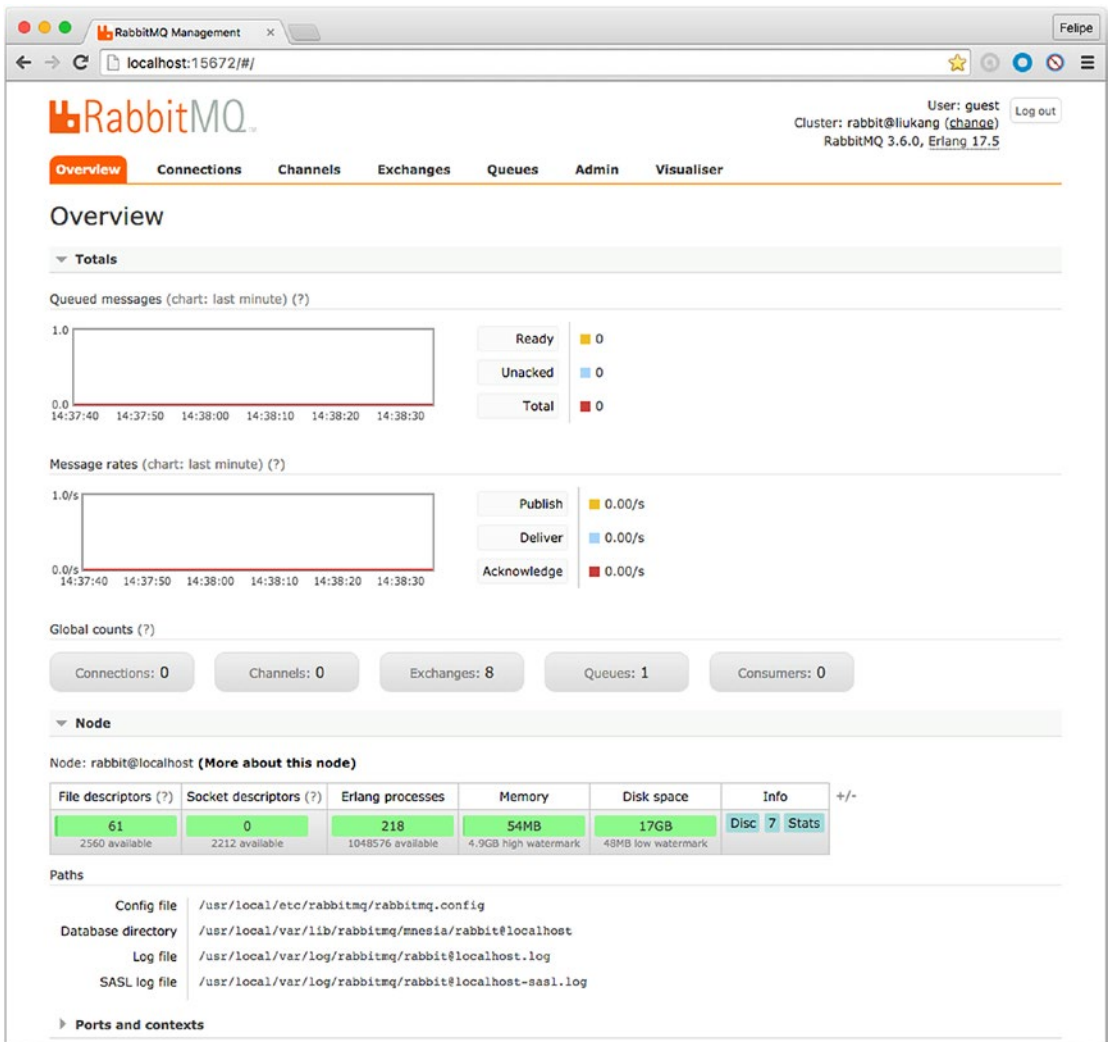


Figure 9-4. RabbitMQ web console management

Figure 9-4 shows the RabbitMQ web console. Now you can run the project as usual, using your IDE. If you are using Maven, execute

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

If you are using Gradle, execute

```
$./gradlew bootRun
```

After you execute this command, you should have something similar to the following output.

```
Producer> Message Sent  
Consumer> Todo(id=null, description=workout tomorrow morning!,  
created=null, modified=null, completed=false)  
Todo created> Todo(id=8a808087645bd67001645bd6785b0000, description=workout  
tomorrow morning!, created=2018-07-02T10:32:19.546, modified=2018-07-  
02T10:32:19.547, completed=false)
```

If you take a look at the RabbitMQ web console in the Queues tab, you should have defined the spring-boot queue (see Figure 9-5).

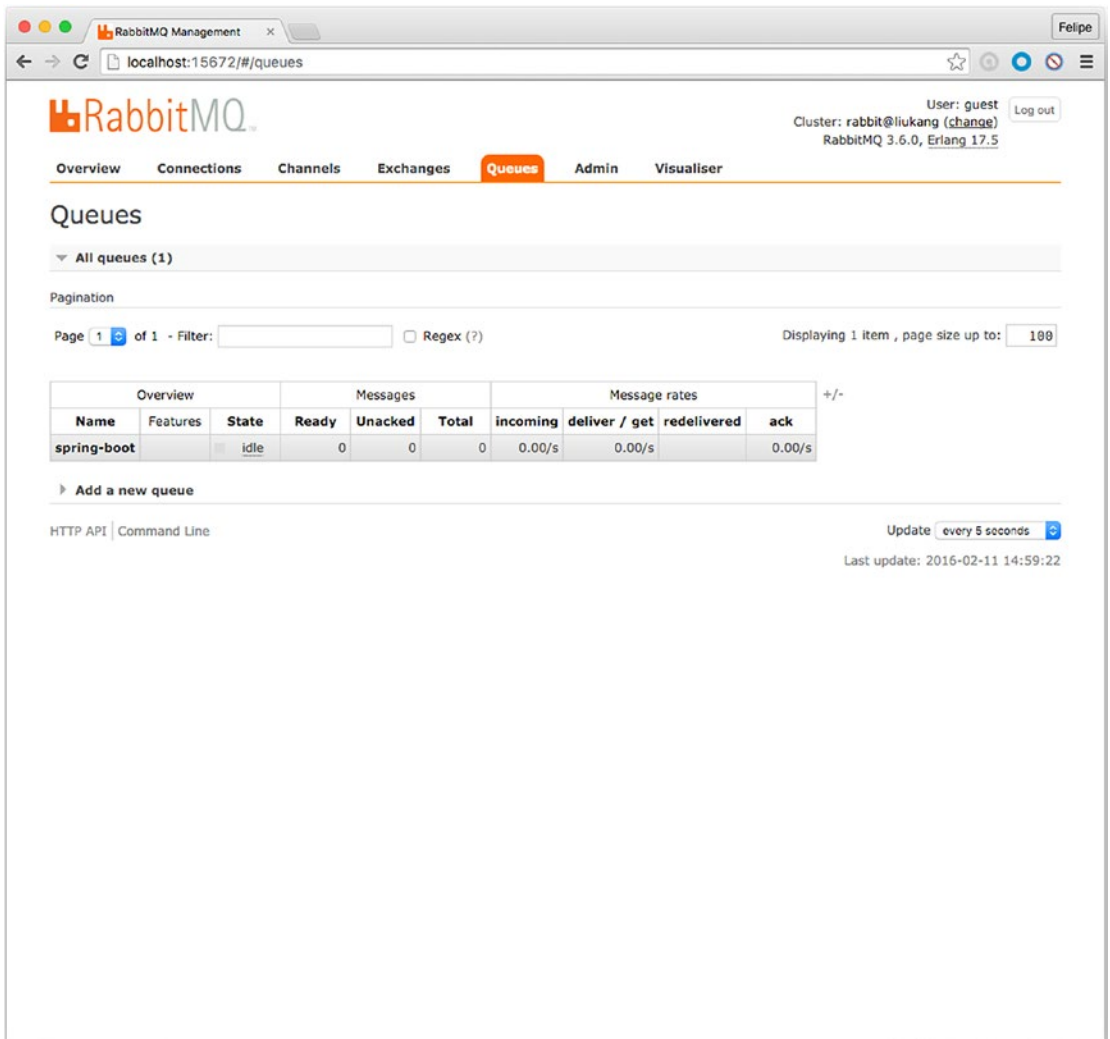


Figure 9-5. RabbitMQ web console Queues tab

Figure 9-5 shows the Queues tab from the RabbitMQ web console. The message you sent was delivered right away. If you want to play a little more and see a part of the throughput, you can modify the `ToDoSender` class as shown in Listing 9-11, but don't forget to stop your app.

Listing 9-11. Version 2 of `com.apress.todo.config.ToDoSender.java`

```

package com.apress.todo.config;

import com.apress.todo.domain.ToDo;
import com.apress.todo.rmqs.ToDoProducer;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Configuration;
import org.springframework.scheduling.annotation.EnableScheduling;
import org.springframework.scheduling.annotation.Scheduled;

import java.text.SimpleDateFormat;
import java.util.Date;

@EnableScheduling
@Configuration
public class ToDoSender {

    @Autowired
    private ToDoProducer producer;
    @Value("${todo.amqp.queue}")
    private String destination;
    private SimpleDateFormat dateFormat = new SimpleDateFormat("HH:mm:ss");

    @Scheduled(fixedRate = 500L)
    private void sendTodos(){
        producer.sendTo(destination,new ToDo("Thinking on Spring Boot at "
        + dateFormat.format(new Date())));
    }
}

```

Listing 9-11 shows a modified version of the `ToDoSender` class. Let's examine this new version.

- `@EnableScheduling`. This annotation tells (via auto-configuration) the Spring container that the `org.springframework.scheduling.annotation.ScheduledAnnotationBeanPostProcessor` class needs to be created. It registers all the methods annotated with `@Scheduled` to be invoked by an `org.springframework.scheduling.TaskScheduler` interface implementation according to the `fixedRate`, `fixedDelay`, or cron expression in the `@Scheduled` annotation.
- `@Scheduled(fixedDelay = 500L)`. This annotation tells the `TaskScheduler` interface implementation to execute the `sendTodos` method with a fixed delay of 500 milliseconds. This means that every half second you send a message to the queue.

The other part of the app you already know. So if you execute the project again, you should see endless messaging. While this is running, take a look at the RabbitMQ console and see the output. You can put a `for` loop to send more messages in a half second.

Remote RabbitMQ

If you want to access a remote RabbitMQ, you add the following properties to the `application.properties` file.

```
spring.rabbitmq.host=mydomain.com
spring.rabbitmq.username=rabbituser
spring.rabbitmq.password=thisissecured
spring.rabbitmq.port=5672
spring.rabbitmq.virtual-host=/production
```

You can always read about all the properties for RabbitMQ in the Spring Boot reference at <https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html>.

Now you know how easy it is to use RabbitMQ with Spring Boot. If you want to learn more about RabbitMQ and the Spring AMQP technology, you can get more information at the main projects web site at <http://projects.spring.io/spring-amqp/>.

You can stop RabbitMQ by pressing Ctrl+C, where you start the broker. There are more options on how to use RabbitMQ, like creating a cluster or having high availability. You can learn more information about it at www.rabbitmq.com.

Redis Messaging with Spring Boot

Now it's Redis' turn. Redis (REmote DIctionary Server) is a NoSQL key-value store database. It's written in C, and even though it has a small footprint in its core, it's very reliable, scalable, powerful, and super fast. Its primary function is to store data structures, such as lists, hashes, strings, sets, and sorted sets. A main feature provides a publish/subscribe messaging system, which is why you are going to use Redis as the message broker.

Installing Redis

Installing Redis is very simple. If you are using Mac OS X/Linux, you can use brew and execute the following.

```
$ brew update && brew install redis
```

If you are using a different flavor of UNIX or Windows, you can go to the Redis web site and download the Redis installers at <http://redis.io/download>. Or if you want to compile it according to your system, you can do that as well by downloading the source code.

ToDo App with Redis

Using Redis for Pub/Sub messaging is very simple and very similar to other technologies. You send and receive ToDo's using the Pub/Sub messaging pattern with Redis.

Let's start by opening your favorite browser and point to Spring Initializr . Add the following values to the fields.

- Group: `com.apress.todo`
- Artifact: `todo-redis`

- Name: todo-redis
- Package Name: com.apress.todo
- Dependencies: Redis, Web, Lombok, JPA, REST Repositories, H2, MySQL

You can select either Maven or Gradle as the project type. Then you can press the Generate Project button, which downloads a ZIP file. Uncompress it and import the project in your favorite IDE (see Figure 9-6).

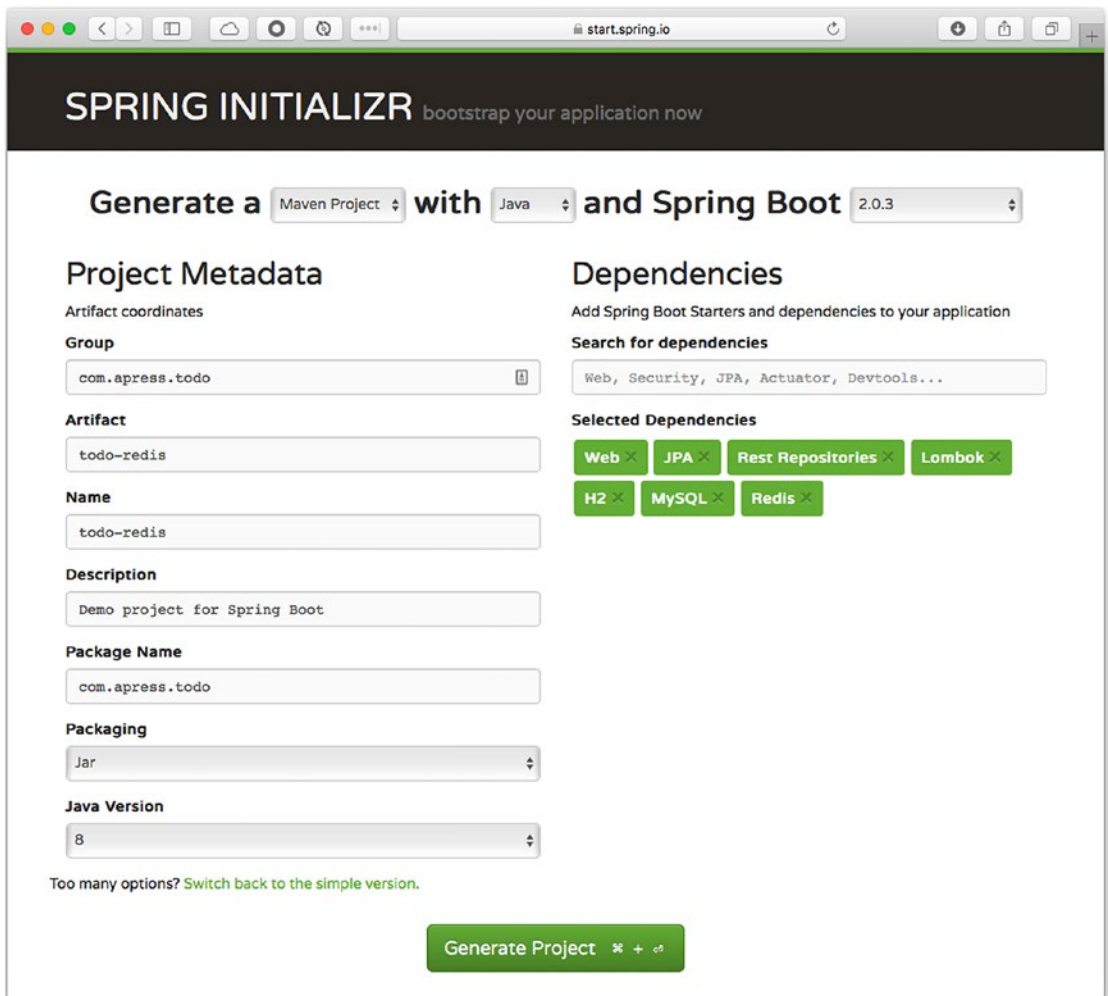


Figure 9-6. *Spring Initializr*

You use the ToDo domain and repo from previous chapters.

ToDo Producer

Let's create the Producer class that sends a `ToDo` instance to a specific topic (see Listing 9-12).

Listing 9-12. `com.apress.todo.redis.ToDoProducer.java`

```
package com.apress.todo.redis;

import com.apress.todo.domain.ToDo;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.stereotype.Component;

@Component
public class ToDoProducer {

    private static final Logger log = LoggerFactory.getLogger(ToDoProducer.class);
    private RedisTemplate redisTemplate;

    public ToDoProducer(RedisTemplate redisTemplate){
        this.redisTemplate = redisTemplate;
    }

    public void sendTo(String topic, ToDo toDo){
        log.info("Producer> ToDo sent");
        this.redisTemplate.convertAndSend(topic, toDo);
    }
}
```

Listing 9-12 shows the Producer class. It is very similar to previous technologies. It uses a `*Template` pattern class; in this case, the `RedisTemplate` that sends `ToDo` instances to a specific topic.

ToDo Consumer

Next, create the consumer that subscribes to the topic (see Listing 9-13).

Listing 9-13. `com.apress.todo.redis.ToDoConsumer.java`

```
package com.apress.todo.redis;

import com.apress.todo.domain.ToDo;
import com.apress.todo.repository.ToDoRepository;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Component;

@Component
public class ToDoConsumer {

    private static final Logger log = LoggerFactory.getLogger(ToDoConsumer.class);
    private ToDoRepository repository;

    public ToDoConsumer(ToDoRepository repository){
        this.repository = repository;
    }

    public void handleMessage(ToDo todo) {
        log.info("Consumer> " + todo);
        log.info("ToDo created> " + this.repository.save(todo));
    }
}
```

Listing 9-13 shows the consumer that is subscribed to the topic for any incoming `ToDo` messages. It is important to know that it is *mandatory* to have a `handleMessage` method name to use the listener (this is a constraint when creating a `MessageListenerAdapter`).

Configuring the ToDo App

Next, let's create the configuration for the ToDo app (see Listing 9-14).

Listing 9-14. `com.apress.todo.config.ToDoConfig.java`

```
package com.apress.todo.config;

import com.apress.todo.domain.ToDo;
import com.apress.todo.redis.ToDoConsumer;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.redis.connection.RedisConnectionFactory;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.data.redis.listener.PatternTopic;
import org.springframework.data.redis.listener.
RedisMessageListenerContainer;
import org.springframework.data.redis.listener.adapter.
MessageListenerAdapter;
import org.springframework.data.redis.serializer.
Jackson2JsonRedisSerializer;

@Configuration
public class ToDoConfig {

    @Bean
    public RedisMessageListenerContainer container(RedisConnectionFactory
connectionFactory,

                                                MessageListenerAdapter
toDoListenerAdapter,
        @Value("${todo.redis.
topic}") String topic) {

        RedisMessageListenerContainer container = new
RedisMessageListenerContainer();
        container.setConnectionFactory(connectionFactory);
    }
}
```

```

        container.addListener(toDoListenerAdapter, new
        PatternTopic(topic));
        return container;
    }

    @Bean
    MessageListenerAdapter toDoListenerAdapter(ToDoConsumer consumer) {
        MessageListenerAdapter messageListenerAdapter = new MessageListener
        Adapter(consumer);
        messageListenerAdapter.setSerializer(new Jackson2JsonRedisSerializer
        <>(ToDo.class));
        return messageListenerAdapter;
    }

    @Bean
    RedisTemplate<String, ToDo> redisTemplate(RedisConnectionFactory
    connectionFactory){
        RedisTemplate<String,ToDo> redisTemplate = new
        RedisTemplate<String,ToDo>();
        redisTemplate.setConnectionFactory(connectionFactory);
        redisTemplate.setDefaultSerializer(new Jackson2JsonRedisSerializer<
        >(ToDo.class));
        redisTemplate.afterPropertiesSet();
        return redisTemplate;
    }
}

```

Listing 9-14 shows the configuration needed for the ToDo app. This class declares the following Spring beans.

- `RedisMessageListenerContainer`. This class is in charge of connecting to the Redis topic.
- `MessageListenerAdapter`. This adapter takes a POJO (Plain Old Java Object) class to process the message. As a requirement, the method must be named `handleMessage`; this method receives the message from the topic as a `ToDo` instance, which is why it also requires a serializer.

- `Jackson2JsonRedisSerializer`. This serializer converts from/to the `ToDo` instance.
- `RedisTemplate`. This class implements the `Template` pattern and is very similar to the other messaging technologies. This class requires a serializer to work with JSON and to/from `ToDo` instances.

This customization is needed to work with the JSON format and do the right conversion to/from `ToDo` instances; but you can avoid everything and use the default configuration that requires a serializable object (like a `String`) to send and use the `StringRedisTemplate` instead.

In the `application.properties` file, add the following content.

```
# JPA
spring.jpa.generate-ddl=true
spring.jpa.hibernate.ddl-auto=create-drop

# ToDo Redis
todo.redis.topic=todos
```

Running the `ToDo` App

Before you run the `ToDo` app, make sure that you have the Redis server up and running. To start it, execute the following command in a terminal.

```
$ redis-server
89887:C 11 Feb 20:17:55.320 # Warning: no config file specified, using the
default config. In order to specify a config file use redis-server /path/
to/redis.conf
89887:M 11 Feb 20:17:55.321 * Increased maximum number of open files to
10032 (it was originally set to 256).
```


Now you can run the project as usual (by running it inside your IDE or using Maven or Gradle). If you are using Maven, execute

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

After executing this command, you should have something similar to the following output in your logs.

```
...
Producer> Message Sent
Consumer> Todo(id=null, description=workout tomorrow morning!,
created=null, modified=null, completed=false)
Todo created> Todo(id=8a808087645bd67001645bd6785b0000, description=workout
tomorrow morning!, created=2018-07-02T10:32:19.546, modified=2018-07-
02T10:32:19.547, completed=false)
...
```

If you take a look at the Redis shell, you should see something like the following.

- 1) "message"
- 2) "todos"
- 3) "{\"id\":null,\"description\":\"workout tomorrow morning!\",\"created\":null,\"modified\":null,\"completed\":false}"

And of course, you can take a look in your browser at <http://localhost:8080/todos> to see the new ToDo.

Well done! You have created a Spring Bot messaging app using Redis. You can shut down Redis by pressing Ctrl+C.

Remote Redis

If you want to access Redis remotely, you need to add the following properties to the `application.properties` file.

```
spring.redis.database=0
spring.redis.host=localhost
spring.redis.password=mysecurepassword
spring.redis.port=6379
```

You can always read about all the properties for Redis in the Spring Boot reference at <https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html>.

You saw that you need to use Redis as a messaging broker, but if you want to know more about the key-value store with Spring, you can check out the Spring Data Redis project at <http://projects.spring.io/spring-data-redis/>.

WebSockets with Spring Boot

It might seem logical that a topic about WebSockets should be in the web chapter, but I consider WebSockets more related to messaging, and that's why this section is in this chapter.

WebSockets is a new way of communication, replacing client/server web technology. It allows long-held single TCP socket connections between the client and server. It's also called *push* technology, which is where the server can send data to the web without the client doing long polling to request a new change.

This section shows you an example where you send a message through a REST endpoint (Producer) and receive the messages (Consumer) using a webpage and JavaScript libraries.

ToDo App with WebSockets

Create the ToDo app that uses JPA REST Repositories. Every time there is a new ToDo, it is posted to a webpage. The connection from the webpage to the ToDo app uses WebSockets using the STOMP protocol.

Let's start by opening your favorite browser and point to Spring Initializr. Add the following values to the fields.

- Group: `com.apress.todo`
- Artifact: `todo-websocket`
- Name: `todo-websocket`
- Package Name: `com.apress.todo`
- Dependencies: `Websocket`, `Web`, `Lombok`, `JPA`, `REST Repositories`, `H2`, `MySQL`

You can select either Maven or Gradle as the project type. Then you can press the Generate Project button, which downloads a ZIP file. Uncompress it and import the project in your favorite IDE (see Figure 9-7).

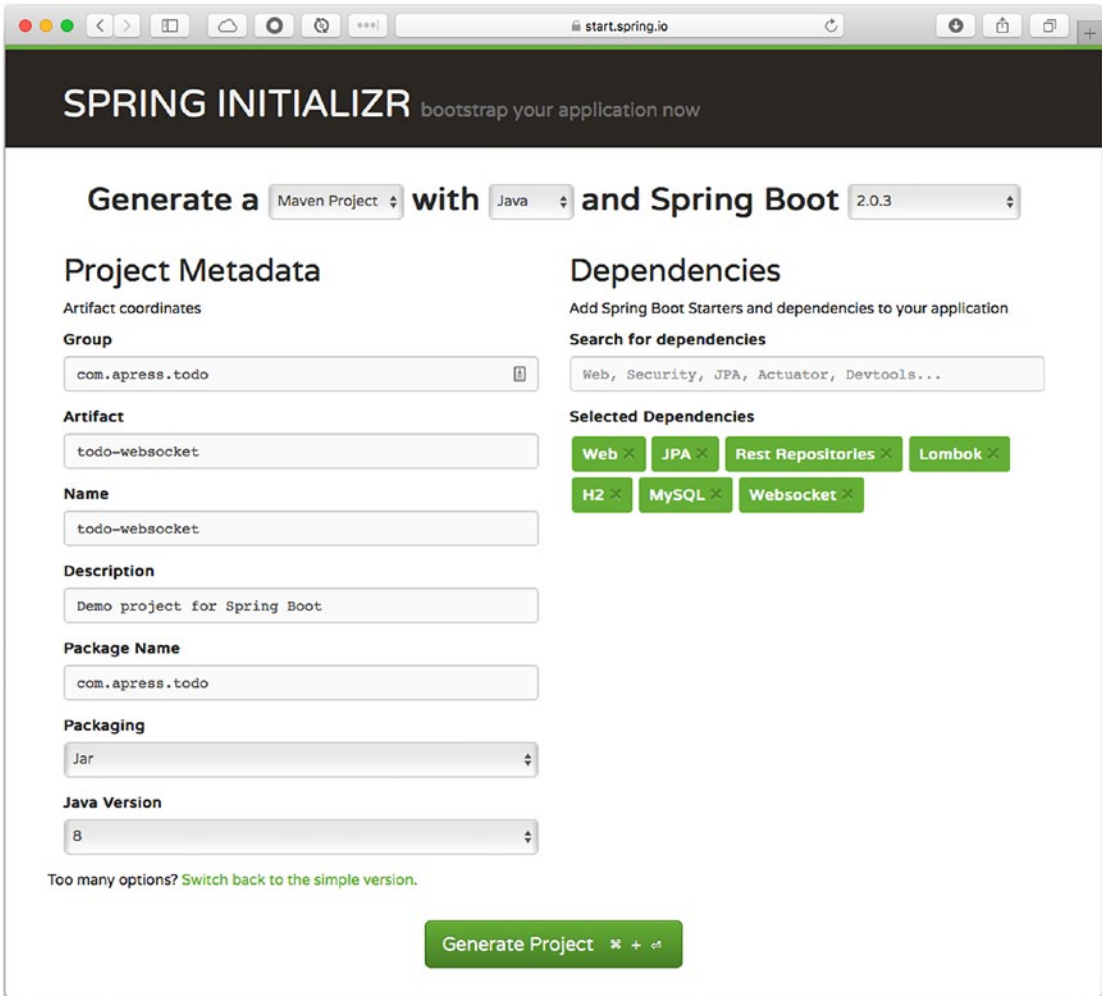


Figure 9-7. Spring Initializr

You can reuse and copy/paste the `ToDo` and `ToDoRepository` classes. Also you need to add the following dependencies; if you are using Maven, add the following to the `pom.xml` file.


```

<dependency>
  <groupId>org.webjars</groupId>
  <artifactId>sockjs-client</artifactId>
  <version>1.1.2</version>
</dependency>
<dependency>
  <groupId>org.webjars</groupId>
  <artifactId>stomp-websocket</artifactId>
  <version>2.3.3</version>
</dependency>
<!-- jQuery -->
<dependency>
  <groupId>org.webjars</groupId>
  <artifactId>jquery</artifactId>
  <version>3.1.1</version>
</dependency>
<!-- Bootstrap -->
<dependency>
  <groupId>org.webjars</groupId>
  <artifactId>bootstrap</artifactId>
  <version>3.3.5</version>
</dependency>

```

If you are using Gradle, add the following dependencies to the `build.gradle` file.

```

compile('org.webjars:sockjs-client:1.1.2')
compile('org.webjars:stomp-websocket:2.3.3')
compile('org.webjars:jquery:3.1.1')
compile('org.webjars:bootstrap:3.3.5')

```

These dependencies create the web client that you need to connect to the messaging broker. The WebJars are a very convenient way to include external resources as packages, instead of worrying about downloading one by one.

ToDo Producer

The producer sends a STOMP message to a topic when a new ToDo is posted by using the HTTP POST method. To do this, it is necessary to catch the event that the Spring Data REST emits when the domain class is persisted to the database.

The Spring Data REST framework has several events that allow control before, during, and after a persistence action. Create a `ToDoEventHandler` class that is listening for the after-create event (see Listing 9-15).

Listing 9-15. `com.apress.todo.event.ToDoEventHandler.java`

```
package com.apress.todo.event;

import com.apress.todo.config.ToDoProperties;
import com.apress.todo.domain.ToDo;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.data.rest.core.annotation.HandleAfterCreate;
import org.springframework.data.rest.core.annotation.
RepositoryEventHandler;
import org.springframework.messaging.simp.SimpMessagingTemplate;
import org.springframework.stereotype.Component;

@Component
@RepositoryEventHandler(ToDo.class)
public class ToDoEventHandler {

    private Logger log = LoggerFactory.getLogger(ToDoEventHandler.class);
    private SimpMessagingTemplate simpMessagingTemplate;
    private ToDoProperties todoProperties;

    public ToDoEventHandler(SimpMessagingTemplate simpMessagingTemplate,
        ToDoProperties todoProperties){
        this.simpMessagingTemplate = simpMessagingTemplate;
        this.todoProperties = todoProperties;
    }
}
```

```

@HandleAfterCreate
public void handleToDoSave(ToDo todo){
    this.simpMessagingTemplate.convertAndSend(this.todoProperties.
    getBroker() + "/new",todo);
    log.info(">> Sending Message to WS: ws://todo/new - " + todo);
}
}

```

Listing 9-15 shows you the event handler that is receiving the after-create event. Let's analyze it.

- `@RepositoryEventHandler`. This annotation tells the `BeanPostProcessor` that this class needs to be inspected for handler methods.
- `SimpMessagingTemplate`. This class is another implementation of the `Template` pattern and is used to send messages using the STOMP protocol. It behaves the same way as the other `*Template` classes from previous sections.
- `ToDoProperties`. This class is a custom properties handler. It describes the broker (`todo.ws.broker`), the endpoint (`todo.ws.endpoint`), and the application endpoint for WebSockets.
- `@HandleAfterCreate`. This annotation marks the method to get any event that happens after the domain class was persisted to the database. As you can see, it uses the `ToDo` instance that was saved into the database. In this method, you are using the `SimpMessagingTemplate` to send a `ToDo` instance to the `/todo/new` endpoint. Any subscriber to that endpoint gets the `ToDo` in JSON format (STOMP).

Next, let's create the `ToDoProperties` class that hold the endpoints information (see Listing 9-16).

Listing 9-16. com.apress.todo.config.ToDoProperties.java

```

package com.apress.todo.config;

import lombok.Data;
import org.springframework.boot.context.properties.ConfigurationProperties;

@Data
@ConfigurationProperties(prefix = "todo.ws")
public class ToDoProperties {

    private String app = "/todo-api-ws";
    private String broker = "/todo";
    private String endpoint = "/stomp";

}

```

The `ToDoProperties` class is a helper to hold information about the broker (`/stomp`) and where the web client connect to (topic - `/todo/new`).

Configuring the ToDo App

This time the `ToDo` app creates a messaging broker that accepts `WebSocket` communication and uses the `STOMP` protocol for message interchange.

Create the config class (see [Listing 9-17](#)).

Listing 9-17. com.apress.todo.config.ToDoConfig.java

```

package com.apress.todo.config;

import org.springframework.boot.context.properties.
EnableConfigurationProperties;
import org.springframework.context.annotation.Configuration;
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;

```

@Configuration**@EnableWebSocketMessageBroker****@EnableConfigurationProperties**(ToDoProperties.class)

```
public class ToDoConfig implements WebSocketMessageBrokerConfigurer {

    private ToDoProperties props;

    public ToDoConfig(ToDoProperties props){
        this.props = props;
    }

    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry) {
        registry.addEndpoint(props.getEndpoint()).setAllowedOrigins("*").
            withSockJS();
    }

    @Override
    public void configureMessageBroker(MessageBrokerRegistry config) {
        config.enableSimpleBroker(props.getBroker());
        config.setApplicationDestinationPrefixes(props.getApp());
    }
}
```

Listing 9-17 shows the `ToDoConfig` class. Let's examine it.

- **@Configuration**. You know that this marks the class as configuration for the Spring container.
- **@EnableWebSocketMessageBroker**. This annotation uses auto-configuration to create all the necessary artifacts to enable broker-backed messaging over WebSockets using a very high-level messaging subprotocol. If you need to customize the endpoints, you need to override the methods from the `WebSocketMessageBrokerConfigurer` interface.
- `WebSocketMessageBrokerConfigurer`. It overrides methods to customize the protocols and endpoints.

- `registerStompEndpoints(StompEndpointRegistry registry)`. This method registers the STOMP protocol; in this case, it registers the /stomp endpoint and uses the JavaScript library SockJS (<https://github.com/sockjs>).
- `configureMessageBroker(MessageBrokerRegistry config)`. This method configures the message broker options. In this case, it enables the broker in the /todo endpoint. This means that the clients that want to use the WebSockets broker need to use the /todo to connect.

Next, let's add information to the `application.properties` file.

src/main/resources/application.properties

```
# JPA
spring.jpa.generate-ddl=true
spring.jpa.hibernate.ddl-auto=create-drop

# Rest Repositories
spring.data.rest.base-path=/api

# WebSocket
todo.ws.endpoint=/stomp
todo.ws.broker=/todo
todo.ws.app=/todo-api-ws
```

The `application.properties` file is declaring a new REST base-path endpoint (/api) because the client is an HTML page and it is the default `index.html`; this means that the REST Repositories live in the /api/* endpoint and not in the root of the app.

ToDo Web Client

The web client is the one making a connection to the messaging broker, where subscribe (using the STOMP protocol) receives any new ToDo that was posted. This client can be any type that handles WebSockets and knows the STOMP protocol.

Let's create a simple `index.html` page that connects to the broker (see Listing 9-18).

Listing 9-18. src/main/resources/static/index.html

```

<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>ToDo WebSockets</title>
  <link rel="stylesheet" href="/webjars/bootstrap/3.3.5/css/bootstrap.
min.css">
  <link rel="stylesheet" href="/webjars/bootstrap/3.3.5/css/bootstrap-
theme.min.css">
</head>
<body>
<div class="container theme-showcase" role="main">
  <div class="jumbotron">
    <h1>What ToDo?</h1>
    <p>An easy way to find out what your are going to do NEXT!</p>
  </div>

  <div class="page-header">
    <h1>Everybody ToDo's</h1>
  </div>
  <div class="row">
    <div class="col-sm-12">
      <div class="panel panel-primary">
        <div class="panel-heading">
          <h3 class="panel-title">ToDo:</h3>
        </div>
        <div class="panel-body">
          <div id="output">
          </div>
        </div>
      </div>
    </div>
  </div>
</div>
</div>

```

```

<script src="/webjars/jquery/3.1.1/jquery.min.js"></script>
<script src="/webjars/sockjs-client/1.1.2/sockjs.min.js"></script>
<script src="/webjars/stomp-websocket/2.3.3/stomp.min.js"></script>

<script>

    $(function(){
        var stompClient = null;
        var socket = new SockJS('http://localhost:8080/stomp');
        stompClient = Stomp.over(socket);

        stompClient.connect({}, function (frame) {
            console.log('Connected: ' + frame);

            stompClient.subscribe('/todo/new', function (data) {
                console.log('>>>> ' + data);
                var json = JSON.parse(data.body);
                var result = "<span><strong>[" + json.created + "]"</strong>&nbsp;" + json.description + "</span><br/>";
                $("#output").append(result);
            });
        });
    });
</script>
</body>
</html>

```

Listing 9-18 shows the index.html, the client that uses the SockJS class to connect to the /stomp endpoint. It subscribes to the /todo/new topic and waits until get a new ToDo is added to the list. The reference to the JavaScript libraries and the CSS is the WebJars class resource.

Running the ToDo App

Now you are ready to start your ToDo app. You can run the application as usual, either using your IDE or in a command line. If you are using Maven, execute

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


If you are using Gradle, execute

```
$ ./gradlew bootRun
```

Open a browser and go to `http://localhost:8080`. You should see an empty `ToDo's` box. Next, open a terminal and execute the following commands.

```
$ curl -XPOST -d '{"description":"Learn to play Guitar"}' -H "Content-Type: application/json" http://localhost:8080/api/todos
```

```
$ curl -XPOST -d '{"description":"read Spring Boot Messaging book from Apress"}' -H "Content-Type: application/json" http://localhost:8080/api/todos
```

You can add more if you like. In your browser, you are seeing the `ToDo's` (see [Figure 9-8](#)).

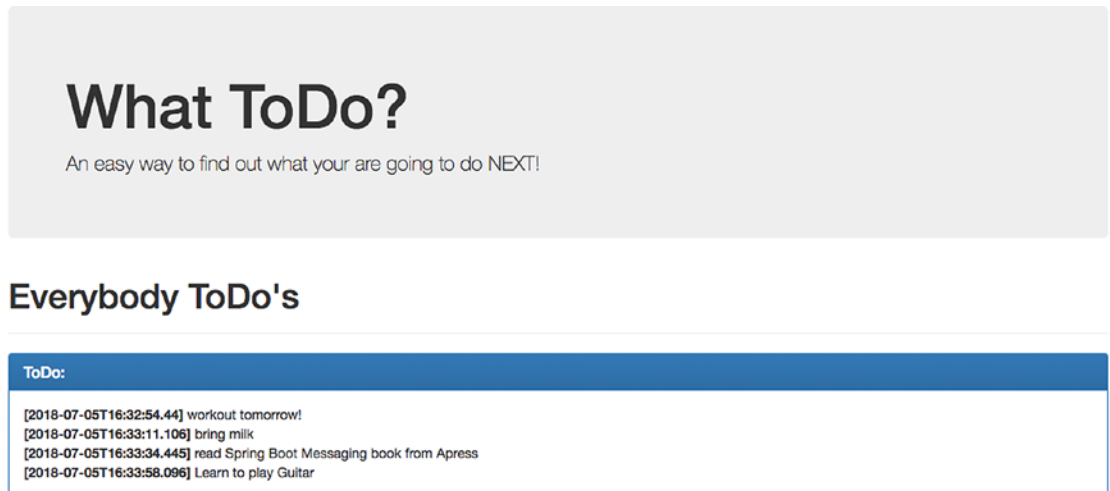


Figure 9-8. *SockJS and Stomp messages: ToDo's List*

Figure 9-8 shows the result of posting messages through WebSockets. Now imagine the possibilities for new applications that require notification in real time (such as creating real-time chatrooms, updating stocks on the fly for your customers, or updating your website without preview or restart). With Spring Boot and WebSockets, you are covered.

Note All the code is available from the Apress site. You can also get the latest at <https://github.com/felipeg48/pro-spring-boot-2nd>.

Summary

This chapter discussed all the technologies that are used for messaging, including JMS and Artemis. It also discussed how to connect to a remote server by providing the server name and port in the `application.properties` file.

You learned about AMQP and RabbitMQ and how you can send and receive messages using Spring Boot. You also learned about Redis and how to use its Pub/Sub messaging. Finally, you learned about WebSockets and how easy it is to implement it with Spring Boot.

If you are into messaging, I wrote *Spring Boot Messaging* (Apress, 2017) (www.apress.com/us/book/9781484212257), which talks about it in detail and exposes more messaging patterns, from simple application events to cloud solutions using Spring Cloud Stream and its transport abstraction.

The next chapter discusses the Spring Boot Actuator and how to monitor your Spring Boot application.

CHAPTER 10

Spring Boot Actuator

This chapter discusses the Spring Boot Actuator module and explains how you can use all its features to monitor your Spring Boot applications.

A common task during and after development that every developer does is to start checking out the logs. Developers check to see if the business logic works as it supposed to, or checks out the processing time of services, and so on. Even though they should have their unit, integration, and regression tests in place, they are not exempt from external failures, including the network (connections, speed, etc.), disk (space, permissions, etc.), and more.

When you deploy to production, it's even more critical. You must pay attention to your applications and sometimes to the whole system. When you start depending on non-functional requirements, such as monitoring systems that check the health of the different applications, or maybe that sets alerts when your application gets to a certain threshold, or worse, when your application crashes, you need to act ASAP.

Developers depend on many third-party technologies to do their job, and I'm not saying that this is bad, but this means that all the heavy lifting is by the DevOps teams. They must monitor every single application and the entire system as a whole.

Spring Boot Actuator

Spring Boot includes an Actuator module, which introduces production-ready non-functional requirements to your application. The Spring Boot Actuator module provides monitoring, metrics, and auditing—right out of the box.

What makes the Actuator module more attractive is that you can expose data through different technologies, such as HTTP (endpoints) and JMX. Spring Boot Actuator metrics monitoring can be done using the Micrometer Framework (<http://micrometer.io/>), that allows you to write once your metrics code and use it in any vendor-neutral engine, such as Prometheus, Atlas, CloudWatch, Datadog, and many, many more.

ToDo App with Actuator

Let's start using the Spring Boot Actuator module in the ToDo application to see how Actuator works. You can start from scratch or you can follow along in the next sections to learn what you need to do. If you are starting from scratch, then you can go to Spring Initializr (<https://start.spring.io>) and add the following values to the fields.

- Group: `com.apress.todo`
- Artifact: `todo-actuator`
- Name: `todo-actuator`
- Package Name: `com.apress.todo`
- Dependencies: `Web`, `Lombok`, `JPA`, `REST Repositories`, `Actuator`, `H2`, `MySQL`

You can select either Maven or Gradle as the project type. Then you can press the **Generate Project** button, which downloads a ZIP file. Uncompress it and import the project in your favorite IDE (see Figure 10-1).

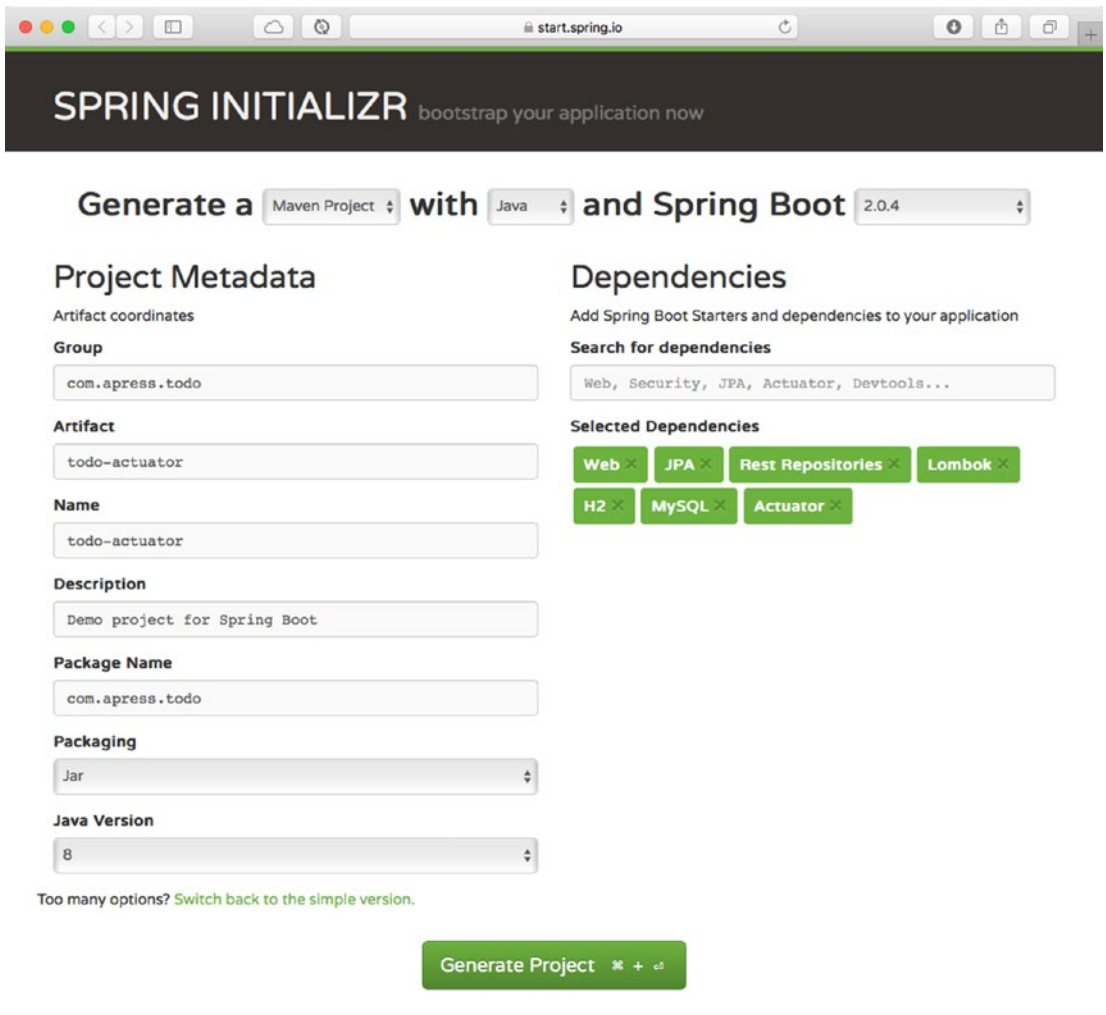


Figure 10-1. *Spring Initializr*

There is nothing from other projects right now; the only new dependency is the Actuator module. You can copy/reuse the `ToDo` domain class and the `ToDoRepository` interface (see Listings 10-1 and 10-2).

Listing 10-1. `com.apress.todo.domain.ToDo.java`

```
package com.apress.todo.domain;

import lombok.Data;
import org.hibernate.annotations.GenericGenerator;
import javax.persistence.*;
```

```
import javax.validation.constraints.NotBlank;
import javax.validation.constraints.NotNull;
import java.time.LocalDateTime;

@Entity
@Data
public class ToDo {

    @Id
    @GeneratedValue(generator = "system-uuid")
    @GenericGenerator(name = "system-uuid", strategy = "uuid")
    private String id;
    @NotNull
    @NotBlank
    private String description;

    @Column(insertable = true, updatable = false)
    private LocalDateTime created;
    private LocalDateTime modified;
    private boolean completed;

    public ToDo(){}
    public ToDo(String description){
        this.description = description;
    }

    @PrePersist
    void onCreate() {
        this.setCreated(LocalDateTime.now());
        this.setModified(LocalDateTime.now());
    }

    @PreUpdate
    void onUpdate() {
        this.setModified(LocalDateTime.now());
    }
}
```

Listing 10-2. `com.apress.todo.repository.ToDoRepository.java`

```
package com.apress.todo.repository;

import com.apress.todo.domain.ToDo;
import org.springframework.data.repository.CrudRepository;

public interface ToDoRepository extends CrudRepository<ToDo,String> { }
```

Before running the `ToDo` app, take a look that you have the `spring-boot-starter-actuator` dependency in your `pom.xml` (if you are using Maven) or `build.gradle` (if you are using Gradle).

You can run the `ToDo` app, and the important thing to notice is the logs output. You should have something similar.

```
INFO 41925 --- [main] s... : Mapped "{[/actuator/health],methods=[GET],
produces=[application/vnd.spring-boot.actuator.v2+json || application/
json]}" ...
INFO 41925 --- [main] s... : Mapped "{[/actuator/info],methods=[GET],
produces=[application/vnd.spring-boot.actuator.v2+json || application/
json]}" ...
INFO 41925 --- [main] s... : Mapped "{[/actuator],methods=[GET],produces=
[application/vnd.spring-boot.actuator.v2+json || application/json]}" ...
```

By default, the Actuator module exposes three endpoints that you can visit.

- `/actuator/health`. This endpoint provides basic application health information. If access is from the browser or with a command line, you get the following response:

```
{
  "status": "UP"
}
```

- `/actuator/info`. This endpoint displays arbitrary application information. If you access this endpoint, you get an empty response; but if you add the following to your `application.properties` file:

```
spring.application.name=todo-actuator
info.application-name=${spring.application.name}
```

```
info.developer.name=Awesome Developer
info.developer.email=awesome@example.com
```

You get the following:

```
{
  "application-name": "todo-actuator",
  "developer": {
    "name": "Awesome Developer",
    "email": "awesome@example.com"
  }
}
```

- /actuator. This endpoint is the prefix of all the actuator endpoints. If you go to this endpoint through a browser or command line, you get this:

```
{
  "_links": {
    "self": {
      "href": "http://localhost:8080/actuator",
      "templated": false
    },
    "health": {
      "href": "http://localhost:8080/actuator/health",
      "templated": false
    },
    "info": {
      "href": "http://localhost:8080/actuator/info",
      "templated": false
    }
  }
}
```

By default, all the endpoints (there are more) are enabled, except for the /actuator/shutdown endpoint; but why are only two endpoints exposed (health and information)? Actually, all of them are exposed through JMX, and this is because some of them contain sensitive information; so, it's important to know what information to expose through the web.

If you want to expose them over the web, there are two properties: `management.endpoints.web.exposure.include` and `management.endpoints.web.exposure.exclude`. You can list them individually separated by a comma or include all of them by using the `*`.

The same applies for exposing the endpoints through JMX with the properties `management.endpoints.jmx.exposure.include` and `management.endpoints.jmx.exposure.exclude`. Remember that by default all the endpoints are exposed through JMX.

As I mentioned before, you not only have a way to expose the endpoints but also to enable them. You can use the following semantic: `management.endpoint.<ENDPOINT-NAME>.enabled`. So, if you want to enable the `/actuator/shutdown` (it is disabled by default) you need to do this in `application.properties`.

```
management.endpoint.shutdown.enabled=true
```

You can add the following property to your `application.properties` file to expose all the web actuator endpoints.

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

If you take a look at the output, you get more actuator endpoints, like `/actuator/beans`, `/actuator/conditions`, and so forth. Let's review some of them in more detail.

/actuator

The `/actuator` endpoint is the prefix of all the endpoints, but if you access it, it provides a hypermedia-based discovery page for all the other endpoints. So, if you go to `http://localhost:8080/actuator`, you should see something similar to Figure 10-2.

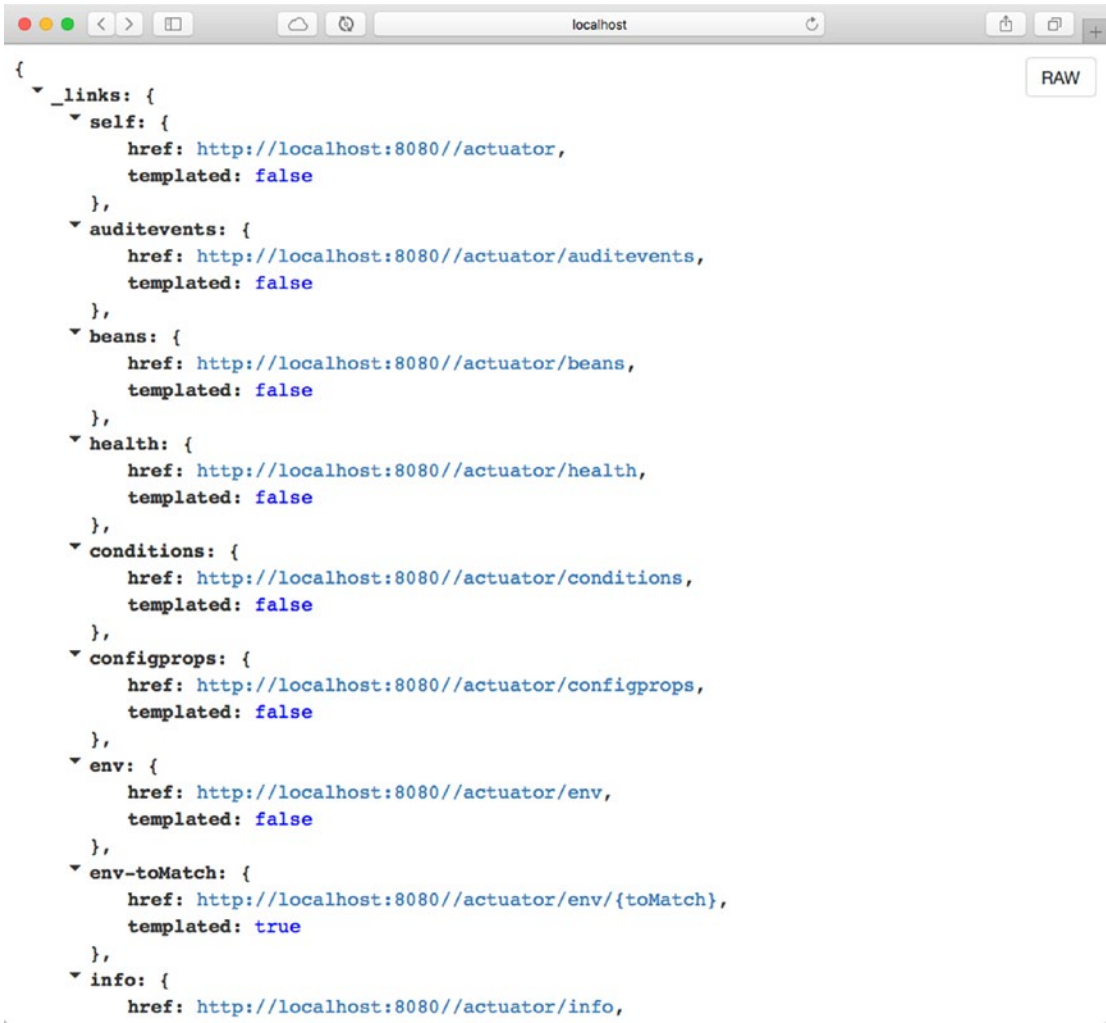


Figure 10-2. `http://localhost:8080/actuator`

`/actuator/conditions`

This endpoint displays the auto-configuration report. It gives you two groups: `positiveMatches` and `negativeMatches`. Remember that the main feature of Spring Boot is that it auto-configures your application by seeing the classpath and dependencies. This has everything to do with the starter poms and extra dependencies that you add to your `pom.xml` file. If you go to `http://localhost:8080/actuator/conditions`, you should see something similar to Figure 10-3.

```

{
  contexts: {
    todo-actuator: {
      positiveMatches: {
        "AuditAutoConfiguration#auditListener": [
          {
            condition: "OnBeanCondition",
            message: "@ConditionalOnMissingBean (types:
            org.springframework.boot.actuate.audit.listener.AbstractAuditListener;
            SearchStrategy: all) did not find any beans"
          }
        ],
        "AuditAutoConfiguration.AuditEventRepositoryConfiguration": [
          {
            condition: "OnBeanCondition",
            message: "@ConditionalOnMissingBean (types:
            org.springframework.boot.actuate.audit.AuditEventRepository;
            SearchStrategy: all) did not find any beans"
          }
        ],
        "AuditEventsEndpointAutoConfiguration#auditEventsEndpoint": [
          {
            condition: "OnBeanCondition",
            message: "@ConditionalOnBean (types:
            org.springframework.boot.actuate.audit.AuditEventRepository;
            SearchStrategy: all) found bean 'auditEventRepository';
            @ConditionalOnMissingBean (types:
            org.springframework.boot.actuate.audit.AuditEventsEndpoint;
            SearchStrategy: all) did not find any beans"
          },
          {
            condition: "OnEnabledEndpointCondition",
            message: "@ConditionalOnEnabledEndpoint no property
            management.endpoint.auditevents.enabled found so using endpoint default"
          }
        ]
      }
    }
  }
}

```

Figure 10-3. <http://localhost:8080/actuator/conditions>

/actuator/beans

This endpoint displays all the Spring beans that are used in your application. Remember that even though you add a few lines of code to create a simple web application, behind the scenes, Spring starts to create all the necessary beans to run your app. If you go to <http://localhost:8080/actuator/beans>, you should see something similar to Figure 10-4.

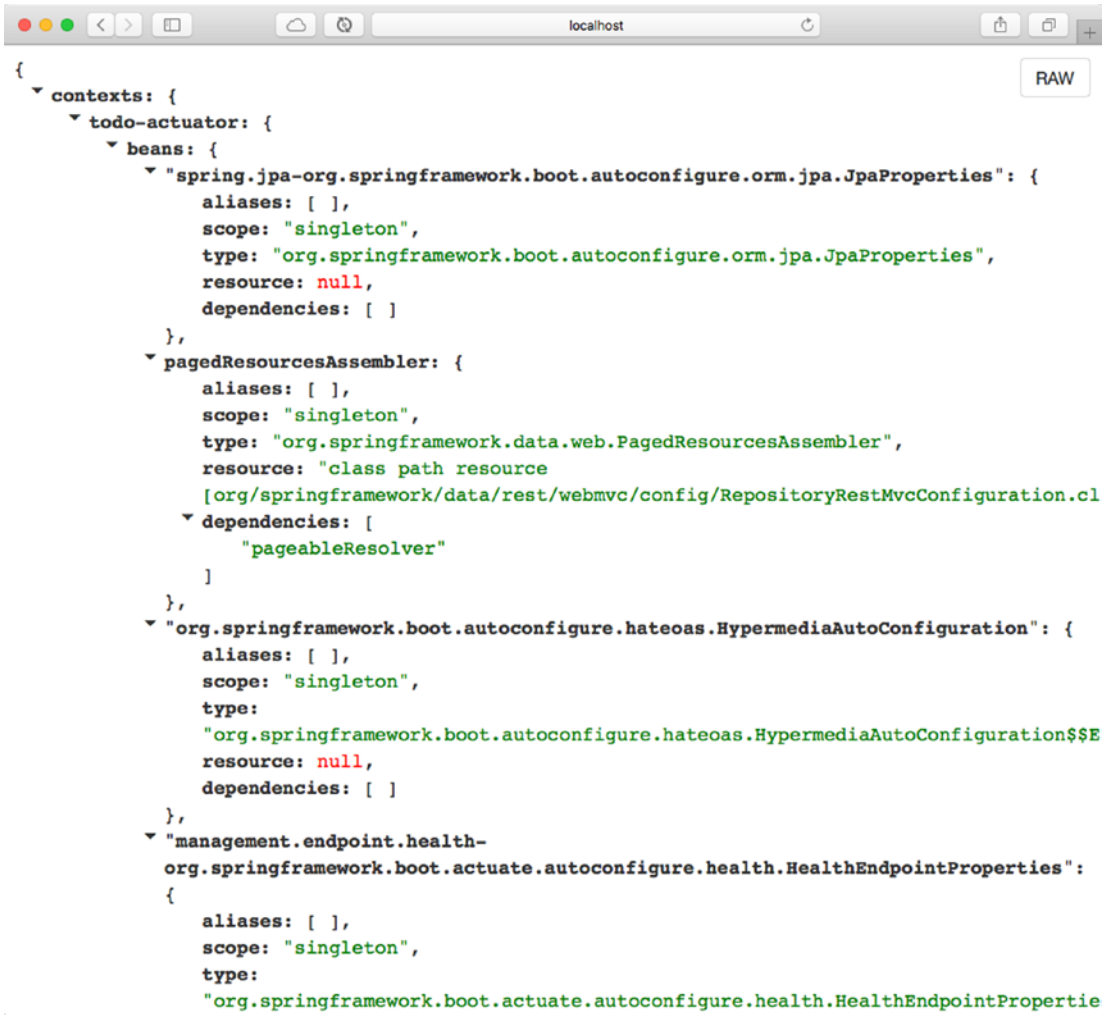
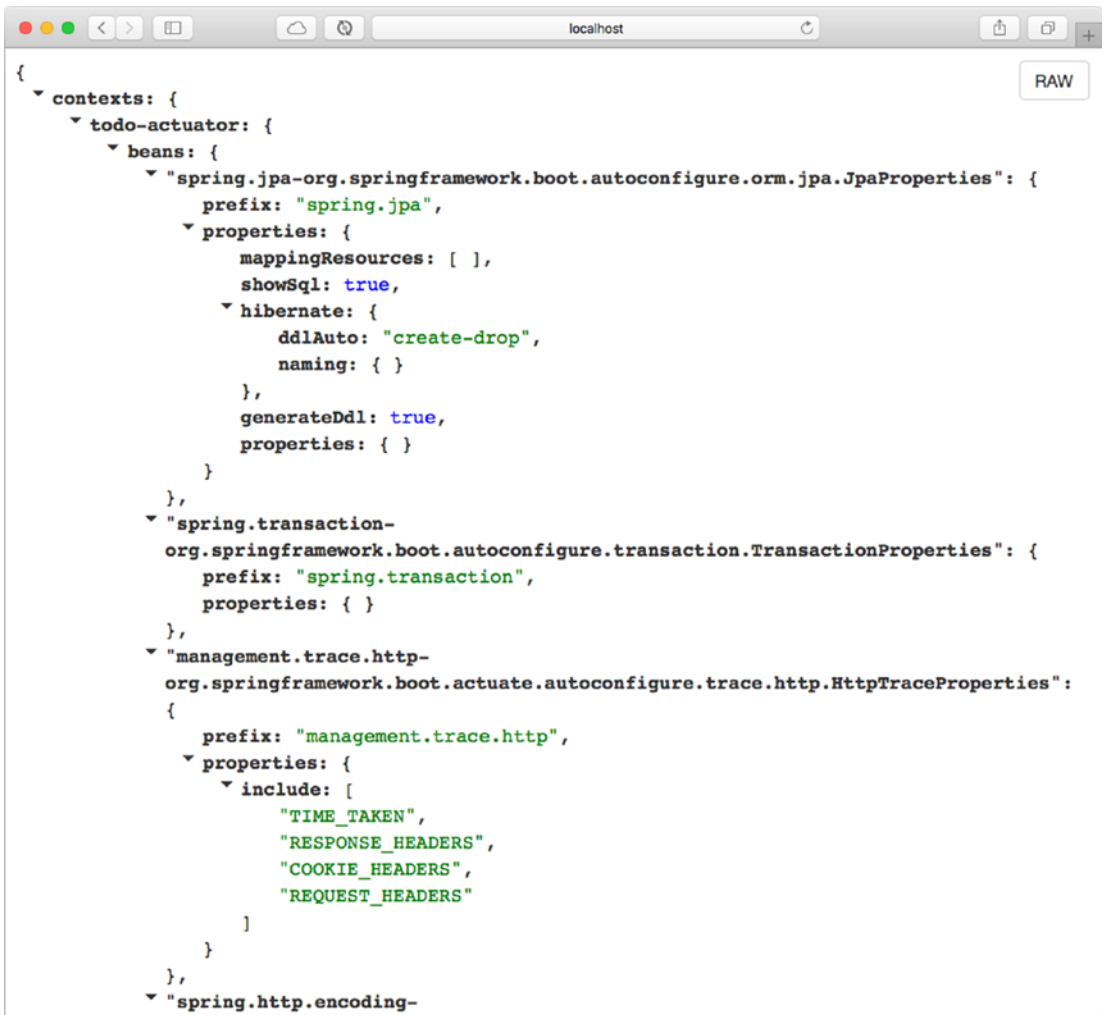


Figure 10-4. <http://localhost:8080/actuator/beans>

/actuator/configprops

This endpoint lists all the configuration properties that are defined by the @ConfigurationProperties beans, which is something that I showed you in earlier chapters. Remember, you can add your own configuration properties prefixes and they can be defined and accessed in the application.properties or YAML files. Figure 10-5 shows an example of this endpoint.



```

{
  contexts: {
    todo-actuator: {
      beans: {
        "spring.jpa-org.springframework.boot.autoconfigure.orm.jpa.JpaProperties": {
          prefix: "spring.jpa",
          properties: {
            mappingResources: [ ],
            showSql: true,
            hibernate: {
              ddlAuto: "create-drop",
              naming: { }
            },
            generateDdl: true,
            properties: { }
          }
        },
        "spring.transaction-
org.springframework.boot.autoconfigure.transaction.TransactionProperties": {
          prefix: "spring.transaction",
          properties: { }
        },
        "management.trace.http-
org.springframework.boot.actuate.autoconfigure.trace.http.HttpTraceProperties":
        {
          prefix: "management.trace.http",
          properties: {
            include: [
              "TIME_TAKEN",
              "RESPONSE_HEADERS",
              "COOKIE_HEADERS",
              "REQUEST_HEADERS"
            ]
          }
        }
      },
      "spring.http.encoding-

```

Figure 10-5. <http://localhost:8080/actuator/configprops>

/actuator/threaddump

This endpoint performs a thread dump of your application. It shows all the threads running and their stack trace of the JVM that is running your app. Go to <http://localhost:8080/actuator/threaddump> endpoint (see Figure 10-6).

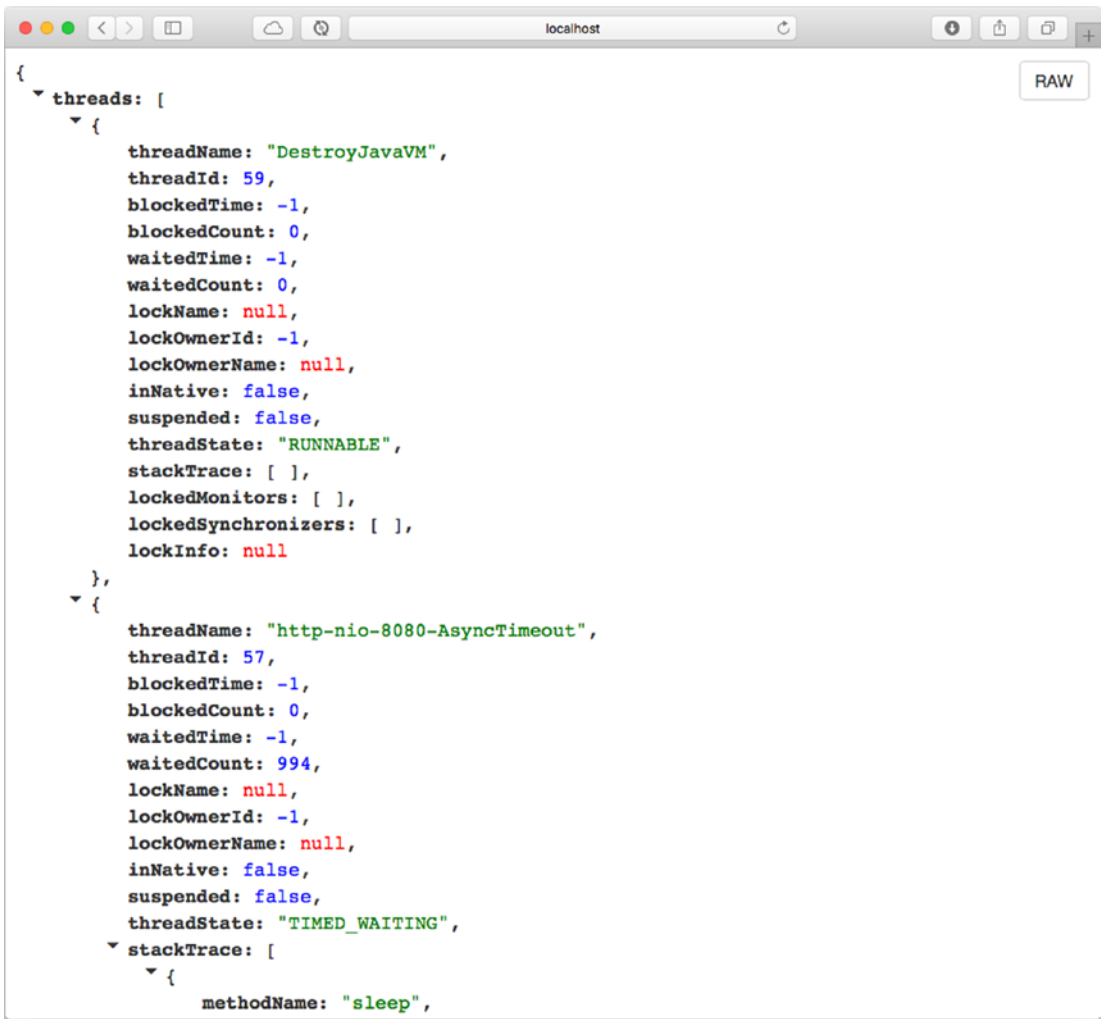


Figure 10-6. <http://localhost:8080/actuator/threaddump>

/actuator/env

This endpoint exposes all the properties from Spring’s ConfigurableEnvironment interface. This shows any active profiles and system environment variables and all application properties, including the Spring Boot properties. Go to <http://localhost:8080/actuator/env> (see Figure 10-7).



```

{
  activeProfiles: [ ],
  propertySources: [
    {
      name: "server.ports",
      properties: {
        "local.server.port": {
          value: 8080
        }
      }
    },
    {
      name: "servletContextInitParams",
      properties: { }
    },
    {
      name: "systemProperties",
      properties: {
        "com.sun.management.jmxremote.authenticate": {
          value: "false"
        },
        "java.runtime.name": {
          value: "Java(TM) SE Runtime Environment"
        },
        "spring.output.ansi.enabled": {
          value: "always"
        },
        "sun.boot.library.path": {
          value:
            "/Library/Java/JavaVirtualMachines/jdk1.8.0_141.jdk/Contents/Home/jre/lib"
        },
        "java.vm.version": {
          value: "25.141-b15"
        },
        "gopherProxySet": {
          value: "false"
        }
      }
    }
  ]
}

```

Figure 10-7. <http://localhost:8080/actuator/env>

/actuator/health

This endpoint shows the health of the application. By default, it shows you the overall system health.

```

{
  "status": "UP"
}

```

If you want to see more information about other systems, you need to use the following property in the `application.properties` file.

```
management.endpoint.health.show-details=always
```

Modify the `application.properties` and re-run the `ToDo` app. If you have a database app (we are), you see the database status, and by default, you also see the `diskSpace` from your system. If you are running your app, you can go to `http://localhost:8080/actuator/health` (see [Figure 10-8](#)).

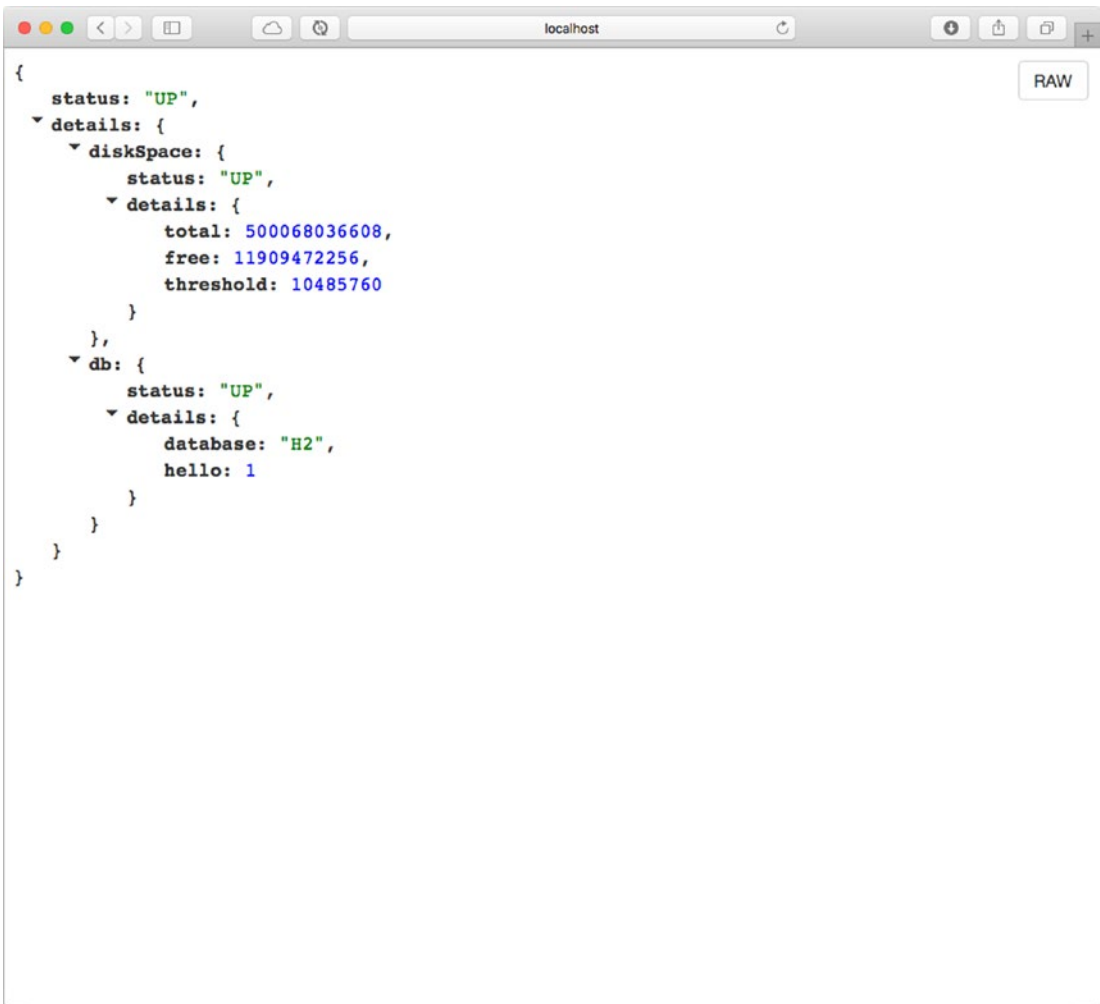


Figure 10-8. `http://localhost:8080/actuator/health` - with details

/actuator/info

This endpoint displays the public application information. This means that you need to add this information to `application.properties`. It's recommended that you add it if you have multiple Spring Boot applications.

/actuator/loggers

This endpoint displays all the loggers available in your app. Figure 10-9 shows the level for a specific package.



```

{
  levels: [
    "OFF",
    "ERROR",
    "WARN",
    "INFO",
    "DEBUG",
    "TRACE"
  ],
  loggers: {
    ROOT: {
      configuredLevel: "INFO",
      effectiveLevel: "INFO"
    },
    com: {
      configuredLevel: null,
      effectiveLevel: "INFO"
    },
    "com.apress": {
      configuredLevel: null,
      effectiveLevel: "INFO"
    },
    "com.apress.todo": {
      configuredLevel: null,
      effectiveLevel: "INFO"
    },
    "com.apress.todo.TODOActuatorApplication": {
      configuredLevel: null,
      effectiveLevel: "INFO"
    },
    "com.apress.todo.interceptor": {
      configuredLevel: null,
      effectiveLevel: "INFO"
    },
    "com.apress.todo.interceptor.ToDoMetricInterceptor": {
      configuredLevel: null,

```

Figure 10-9. `http://localhost:8080/actuator/loggers`

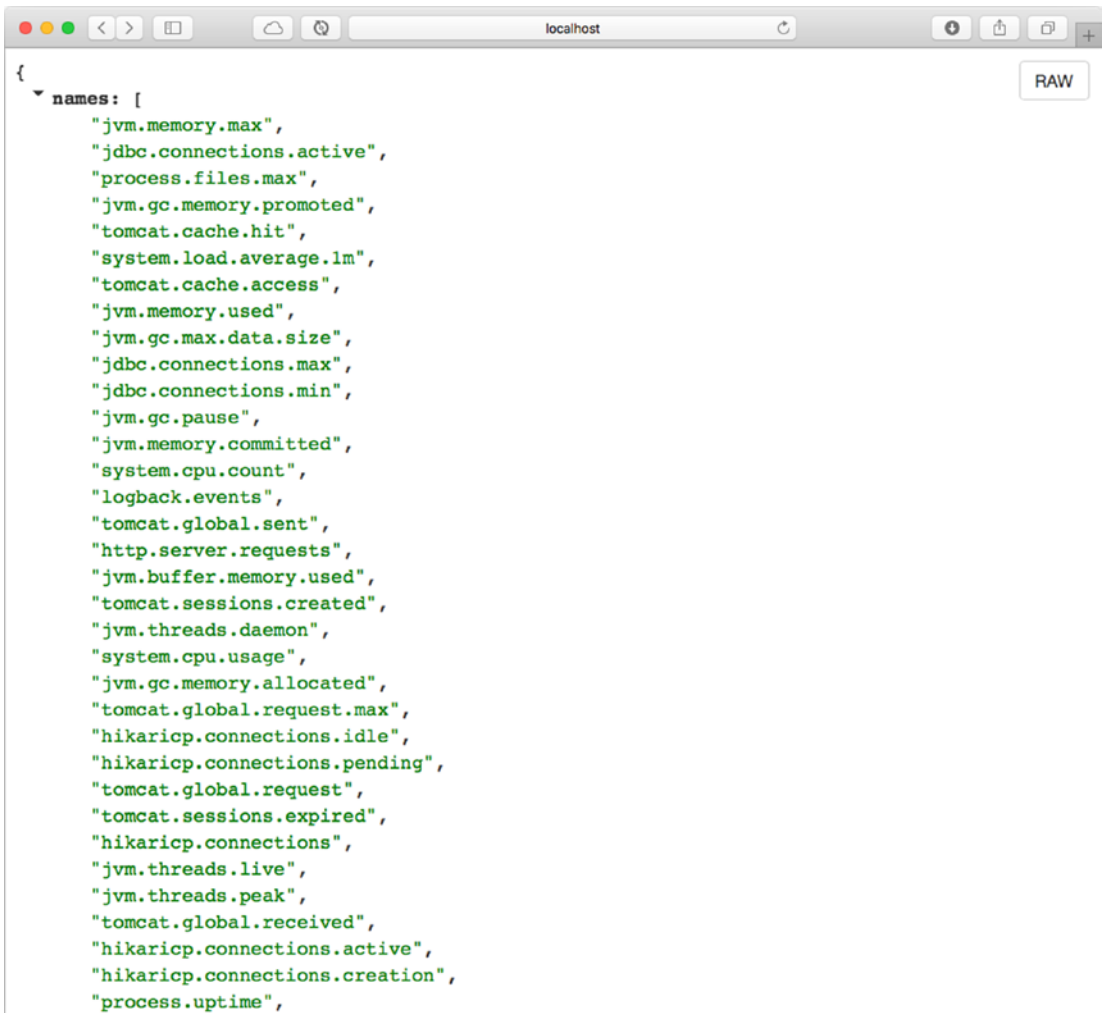
/actuator/loggers/{name}

With this endpoint, you can look for a particular package and its log level. So, if you configure, for example, `logging.level.com.apress.todo=DEBUG` and you reach the `http://localhost:8080/actuator/loggers/com.apress.todo` endpoint, you get the following.

```
{  
  "configuredLevel": DEBUG,  
  "effectiveLevel": "DEBUG"  
}
```

/actuator/metrics

This endpoint shows the metrics information of the current application, where you can determine the how much memory it's using, how much memory is free, the uptime of your application, the size of the heap being used, the number of threads used, and so on (see Figure 10-10 and Figure 10-11).



```

{
  names: [
    "jvm.memory.max",
    "jdbc.connections.active",
    "process.files.max",
    "jvm.gc.memory.promoted",
    "tomcat.cache.hit",
    "system.load.average.1m",
    "tomcat.cache.access",
    "jvm.memory.used",
    "jvm.gc.max.data.size",
    "jdbc.connections.max",
    "jdbc.connections.min",
    "jvm.gc.pause",
    "jvm.memory.committed",
    "system.cpu.count",
    "logback.events",
    "tomcat.global.sent",
    "http.server.requests",
    "jvm.buffer.memory.used",
    "tomcat.sessions.created",
    "jvm.threads.daemon",
    "system.cpu.usage",
    "jvm.gc.memory.allocated",
    "tomcat.global.request.max",
    "hikaricp.connections.idle",
    "hikaricp.connections.pending",
    "tomcat.global.request",
    "tomcat.sessions.expired",
    "hikaricp.connections",
    "jvm.threads.live",
    "jvm.threads.peak",
    "tomcat.global.received",
    "hikaricp.connections.active",
    "hikaricp.connections.creation",
    "process.uptime",
  ]
}

```

Figure 10-10. *http://localhost:8080/actuator/metrics*

You can access every metric by adding the name at the end of the endpoint; so if you want to know more about `jvm.memory.max`, you need to reach `http://localhost:8080/actuator/metrics/jvm.memory.max` (see Figure 10-11).

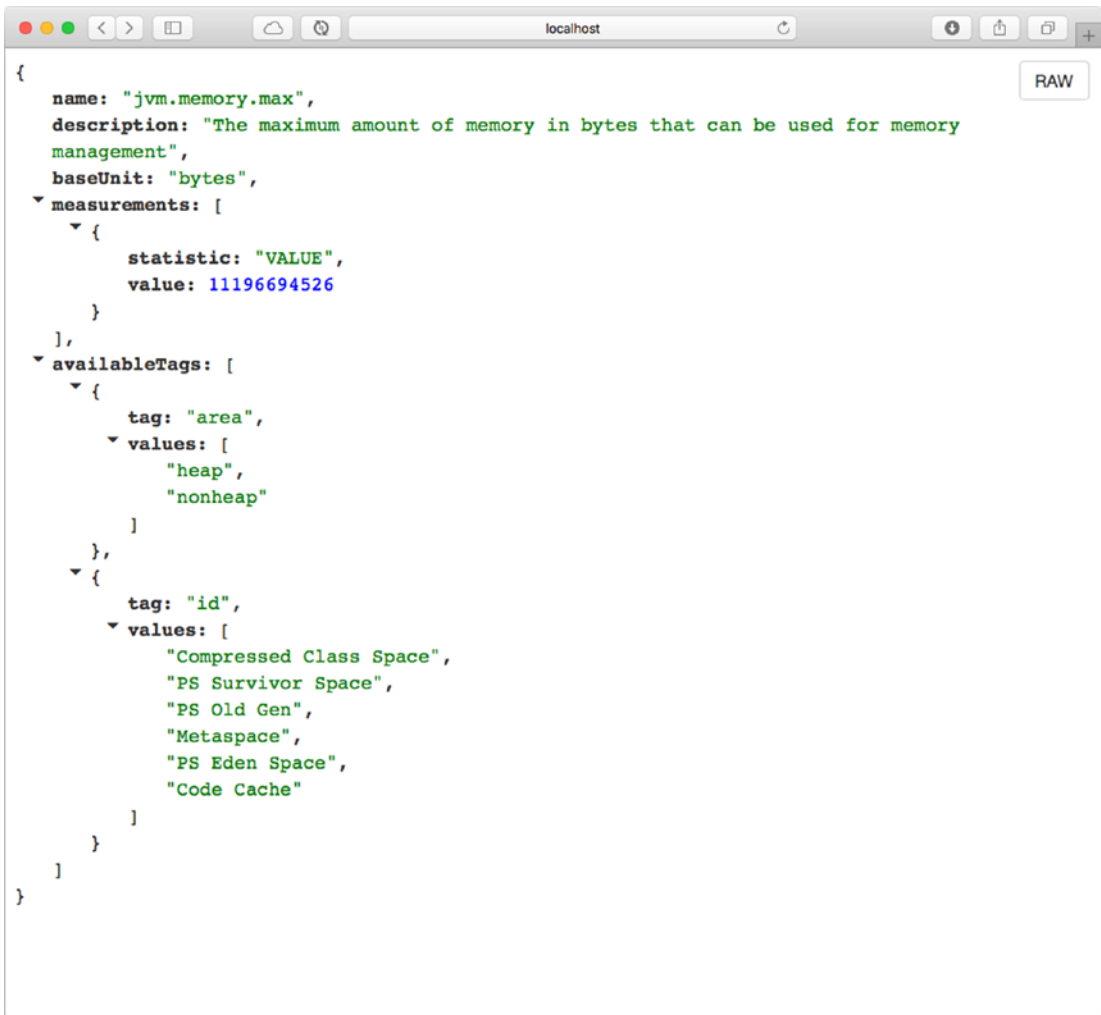


Figure 10-11. *http://localhost:8080/actuator/metrics/jvm.memory.max*

If you take a look at Figure 10-11, in the availableTags section, you can get more information by appending tag=KEY:VALUE. You can use `http://localhost:8080/actuator/metrics/jvm.memory.max?tag=area:heap` and get information about the heap.

/actuator/mappings

This endpoint shows all the lists of all the @RequestMapping paths declared in your application. This is very useful if you want to know more about which mappings are declared. If your application is running, you can go to the `http://localhost:8080/actuator/mappings` endpoint (see Figure 10-12).

```

{
  contexts: {
    todo-actuator: {
      mappings: {
        dispatcherServlets: {
          dispatcherServlet: [
            {
              handler: "ResourceHttpRequestHandler [locations=[class path resource [META-INF/resources/], class path resource [resources/], class path resource [static/], class path resource [public/], ServletContext resource [/], class path resource []], resolvers=[org.springframework.web.servlet.resource.PathResourceResolver@2b2f7516]",
              predicate: "/*/favicon.ico",
              details: null
            },
            {
              handler: "public java.lang.Object org.springframework.boot.actuate.endpoint.web.servlet.AbstractWebMvcEndpoint<java.lang.String>",
              predicate: "{[/actuator/auditevents],methods=[GET],produces=[application/vnd.spring-boot.actuator.v2+json || application/json]}",
              details: {
                handlerMethod: {
                  className: "org.springframework.boot.actuate.endpoint.web.servlet.AbstractWebMvcEndpoint",
                  name: "handle",
                  descriptor: "(Ljavax/servlet/http/HttpServletRequest;Ljava/util/Map;)Ljava/lang/Object;"
                },
                requestMappingConditions: {
                  consumes: [ ],
                  headers: [ ],
                  methods: [ "GET" ],
                  params: [ ],
                }
              }
            }
          ]
        }
      }
    }
  }
}

```

Figure 10-12. `http://localhost:8080/actuator/mappings`

/actuator/shutdown

This endpoint is not enabled by default. It allows the application to be gracefully shut down. This endpoint is sensitive, which means that it can be used with security, and it should be. If your application is running, you can stop it now. If you want to enable the `/actuator/shutdown` endpoint, you need to add the following to the application's properties.

```
management.endpoint.shutdown.enabled=true
```

It's wise to have this endpoint secured. You'd need to add the `spring-boot-starter-security` dependency to your `pom.xml` (if you are using Maven).

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

If you are using Gradle, you can add the following dependency to your `build.gradle`.

```
compile ('org.springframework.boot: spring-boot-starter-security')
```

Remember that by adding the security dependency, you enable security by default. The username is `user` and the password is printed in the logs. Also, you can establish better security by using in-memory, database, or LDAP users; see the Spring Boot security chapter for more information.

For now, let's add `management.endpoint.shutdown.enabled=true` and the `spring-boot-starter-security` dependency and rerun the application. After running the application, take a look at the logs and save the password that is printed so that it can be used with the `/actuator/shutdown` endpoint.

```
...
Using default security password: 2875411a-e609-4890-9aa0-22f90b4e0a11
...
```

Now if you open a terminal, you can execute the following command.

```
$ curl -i -X POST http://localhost:8080/shutdown -u user:2875411a-e609-4890-9aa0-22f90b4e0a11
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
X-Content-Type-Options: nosniff
X-XSS-Protection: 1; mode=block
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: 0
```

```
X-Frame-Options: DENY
Strict-Transport-Security: max-age=31536000 ; includeSubDomains
X-Application-Context: application
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked
Date: Wed, 17 Feb 2018 04:22:58 GMT

{"message":"Shutting down, bye..."}
```

As you can see from this output, you are using a POST method to access the `/actuator/shutdown` endpoint, and you are passing the user and the password that was printed before. The result is the Shutting down, bye... message. And of course, your application is terminated. Again, it's important to know that this particular endpoint must be secured at all times.

/actuator/httptrace

This endpoint shows the trace information, which are normally the last few HTTP requests. This endpoint can be useful to see all the request information and the information returned to debug your application at the HTTP level. You can run your application and go to `http://localhost:8080/actuator/httptrace`. You should see something similar to Figure 10-13.

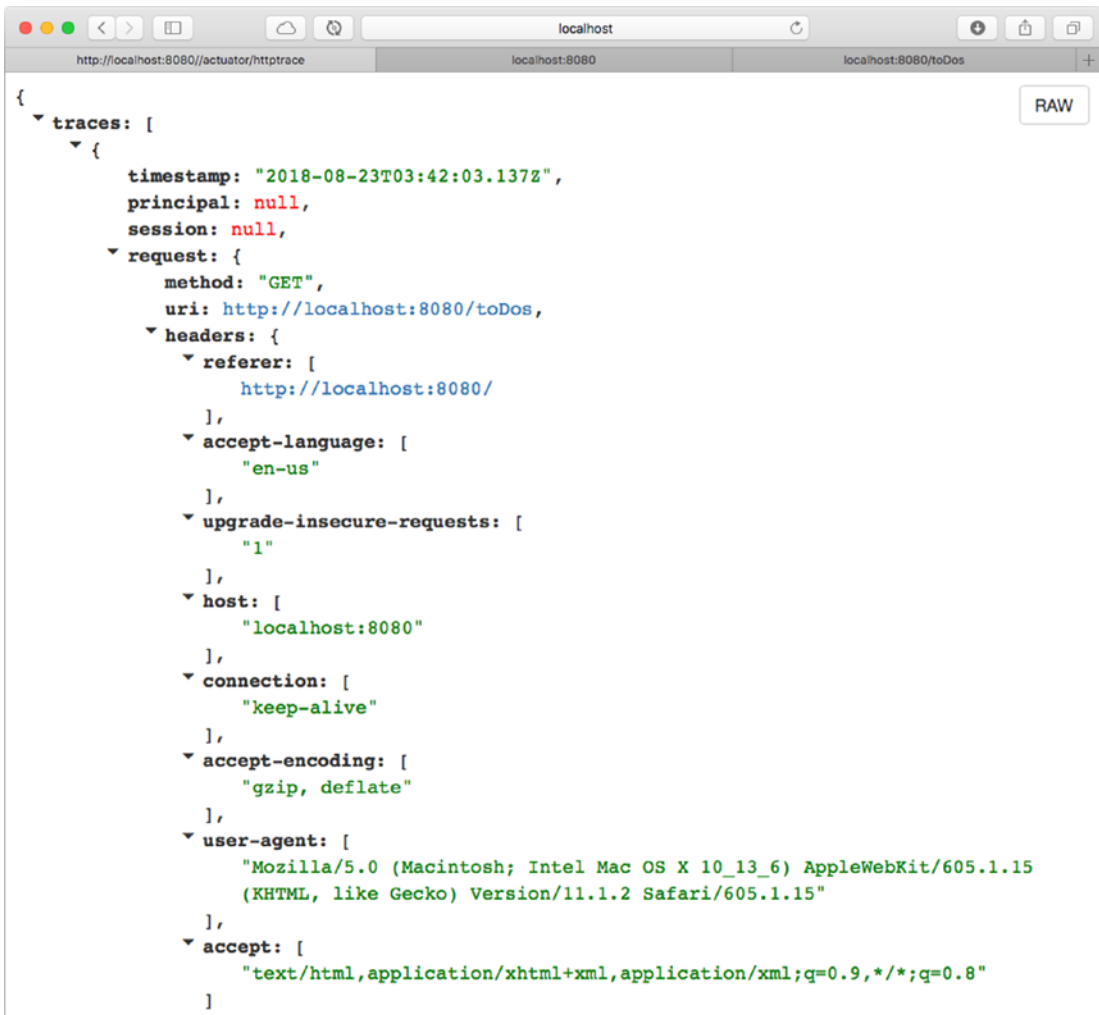


Figure 10-13. `http://localhost:8080/actuator/httptrace`

Changing the Endpoint ID

You can configure the endpoint ID, which changes the name. Imagine that you don't like the `/actuator/beans` endpoint, which is referring to Spring beans, so what about if you change this endpoint to `/actuator/spring`.

You make this change in the `application.properties` file in the form of `management.endpoints.web.path-mapping.<endpoint-name>=<new-name>`; for example, `management.endpoints.web.path-mapping.beans=spring`.

If you re-run your application (stop and restart to apply the changes), you can access the `/actuator/beans` endpoint using the `/actuator/spring` endpoint instead.

Actuator CORS Support

With the Spring Boot Actuator module, you can configure CORS (cross-origin resource sharing), which allows you to specify which cross domains are authorized to use the actuator's endpoints. Normally, this allows interapplications connect to your endpoints, and due to security reasons, only the authorized domains are able to execute these endpoints.

You configure this in the `application.properties` file.

```
management.endpoints.web.cors.allowed-origins=http://mydomain.com
management.endpoints.web.cors.allowed-methods=GET, POST
```

If your application is running, stop it and re-run it.

Normally in the `management.endpoints.web.cors.allowed-origins`, you should put a domain name like `http://mydomain.com` or maybe `http://localhost:9090` (not the `*`), which allows access to your endpoints to avoid any hack to your site. This would be very similar to using the `@CrossOrigin(origins = "http://localhost:9000")` annotation in any controller.

Changing the Management Endpoints Path

By default, the Spring Boot Actuator has its management in `/actuator` as the root, which means that all the actuator's endpoints can be accessed from `/actuator`; for example, `/actuator/beans`, `/actuator/health`, and so on. Before you continue, stop your application. You can change its management context path by adding the following property to the `application.properties` file.

```
management.endpoints.web.base-path=/monitor
```

If you re-run your application, you see that `EndpointHandlerMapping` is mapping all the endpoints by adding the `/monitor/<endpoint-name>` context path. You can now access the `/httptrace` endpoint through `http://localhost:8080/monitor/httptrace`.

You can also change the server address, add SSL, use a particular IP, or change the port for the endpoints, with the `management.server.*` properties.

```
management.server.servlet.context-path=/admin
management.server.port=8081
management.server.address=127.0.0.1
```

This configuration has its endpoint with the context-path `/admin/actuator/<endpoint-name>`. The port is 8081 (this means that you have two ports listening: 8080 for your application and 8081 for your management endpoints). The endpoints or management is bound to the 127.0.0.1 address.

If you want to disable the endpoints (for security reasons), you have two options, you can use `management.endpoints.enabled-by-default=false` or you can use the `management.server.port=-1` properties.

Securing Endpoints

You can secure your actuator endpoints by including `spring-boot-starter-security` and configuring `WebSecurityConfigurerAdapter`; this is through the `HttpSecurity` and `RequestMatcher` configurations.

```
@Configuration
public class ToDoActuatorSecurity extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .requestMatcher(EndpointRequest.toAnyEndpoint())
            .authorizeRequests()
            .anyRequest().hasRole("ENDPOINT_ADMIN")
            .and()
            .httpBasic();
    }
}
```

It is important to know that the `ENDPOINT_ADMIN` role is required to access the endpoints for security.

Configuring Endpoints

By default, you see that actuator endpoints cache responds to read operations that don't accept any parameters; so if you need to change this behavior, you can use the `management.endpoint.<endpoint-name>.cache.time-to-live` property. And as

another example, if you need to change the `/actuator/beans` cache, you can add the following to the `application.properties` file.

```
management.endpoint.beans.cache.time-to-live=10s
```

Implementing Custom Actuator Endpoints

You can extend or create a custom actuator endpoint. You need to mark your class with `@Endpoint` and also mark your methods with `@ReadOperation`, `@WriteOperation` or `@DeleteOperation`; by default, your endpoint is exposed over JMX and through the web over HTTP.

You can be more specific and decide if you only want to expose your endpoint to JMX, and then mark your class as `@JmxEndpoint`. If you only need it for the web, then you mark your class with `@WebEndpoint`.

When creating the methods, you can accept parameters, which are converted to the right type needed by the `ApplicationConversionService` instance. These types consume the `application/vnd.spring-boot.actuator.v2+json` and `application/json` content type.

You can return any type (even `void` or `Void`) in any method signature. Normally, the returned content type varies depending on the type. If it is an `org.springframework.core.io.Resource` type, it returns an `application/octet-stream` content type; for all other types, it returns an `application/vnd.spring-boot.actuator.v2+json`, `application/json` content type.

When using your custom actuator endpoint over the web, the operations have their own HTTP method defined: `@ReadOperation` (`Http.GET`), `@WriteOperation` (`Http.POST`) and `@DeleteOperation` (`Http.DELETE`).

ToDo App with Custom Actuator Endpoints

Let's create a custom endpoint (`/todo-stats`) that shows the count of `ToDo`'s in the database and how many are completed. Also, we can create a write operation that can complete a `ToDo` and even an operation for removing a `ToDo`.

Let's create `ToDoStatsEndpoint` to hold all the logic of the custom endpoint (see Listing 10-3).

Listing 10-3. com.apress.todo.actuator.ToDoStatsEndpoint.java

```

package com.apress.todo.actuator;

import com.apress.todo.domain.ToDo;
import com.apress.todo.repository.ToDoRepository;
import lombok.AllArgsConstructor;
import lombok.Data;
import org.springframework.boot.actuate.endpoint.annotation.*;
import org.springframework.stereotype.Component;

@Component
@Endpoint(id="todo-stats")
public class ToDoStatsEndpoint {

    private ToDoRepository todoRepository;

    ToDoStatsEndpoint(ToDoRepository todoRepository){
        this.todoRepository = todoRepository;
    }

    @ReadOperation
    public Stats stats() {
        return new Stats(this.todoRepository.count(),this.todoRepository.
            countByCompleted(true));
    }

    @ReadOperation
    public ToDo getToDo(@Selector String id) {
        return this.todoRepository.findById(id).orElse(null);
    }

    @WriteOperation
    public Operation completeToDo(@Selector String id) {
        ToDo todo = this.todoRepository.findById(id).orElse(null);
        if(null != todo){
            todo.setCompleted(true);
            this.todoRepository.save(todo);
            return new Operation("COMPLETED",true);
        }
    }
}

```

```

        return new Operation("COMPLETED",false);
    }

    @DeleteOperation
    public Operation removeToDo(@Selector String id) {
        try {
            this.todoRepository.deleteById(id);
            return new Operation("DELETED",true);
        }catch(Exception ex){
            return new Operation("DELETED",false);
        }
    }

    @AllArgsConstructor
    @Data
    public class Stats {
        private long count;
        private long completed;
    }

    @AllArgsConstructor
    @Data
    public class Operation{
        private String name;
        private boolean successful;
    }
}

```

Listing 10-3 shows the custom endpoint that does operations such as showing stats (the total number of ToDo's and the number that are completed). It gets a ToDo object, removes it, and sets it as completed. Let's review it.

- **@Endpoint.** Identifies a type as being an actuator endpoint that provides information about the running application. Endpoints can be exposed over a variety of technologies, including JMX and HTTP. This is the `ToDoStatsEndpoint` class that is your endpoint for the actuator.

- `@ReadOperation`. Identifies a method on an endpoint as being a read operation (sees that this class returns the `ToDo` by ID).
- `@Selector`. A selector can be used on a parameter of an endpoint method to indicate that the parameter selects a subset of the endpoint's data. This is a way to modify a value, in this case, for updating the `ToDo` as completed.
- `@WriteOperation`. Identifies a method on an endpoint as being a write operation. This is very similar to a POST event.
- `@DeleteOperation`. Identifies a method on an endpoint as being a delete operation.

Listing 10-3 is very similar to a REST API, but in this case, all of these methods are exposed through a JMX protocol. Also, the `stats` method is using `todoRepository` to call the `countByCompleted` method. Let's add it to the `ToDoRepository` interface (see Listing 10-4).

Listing 10-4. `com.apress.todo.repository.ToDoRepository.java - v2`

```
package com.apress.todo.repository;

import com.apress.todo.domain.ToDo;
import org.springframework.data.repository.CrudRepository;

public interface ToDoRepository extends CrudRepository<ToDo,String> {
    public long countByCompleted(boolean completed);
}
```

Listing 10-4 shows version 2 of the `ToDoRepository` interface. This interface now has a new method declaration, `countByCompleted`; remember that this is a *named query* method and the Spring Data module takes care of creating the proper SQL statement to count the number of `ToDo`'s that are already completed.

As you can see, this is very straightforward for creating a custom endpoint. Now, if you run the application, and go to `http://localhost:8080/actuator`, you should see the `todo-stats` endpoint listed (see Figure 10-14).



Figure 10-14. *http://localhost:8080/actuator - todo-stats custom endpoint*

If you click the first todo-stats link, you see what's shown in Figure 10-15.

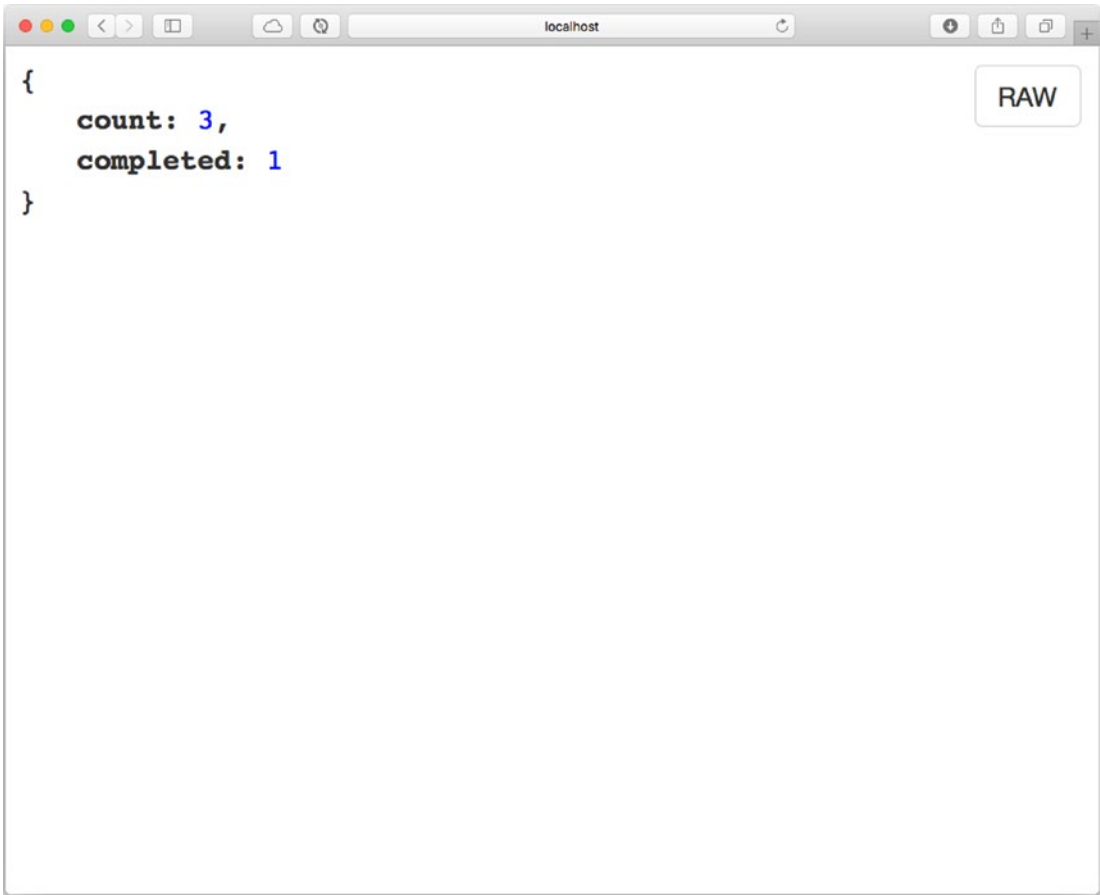
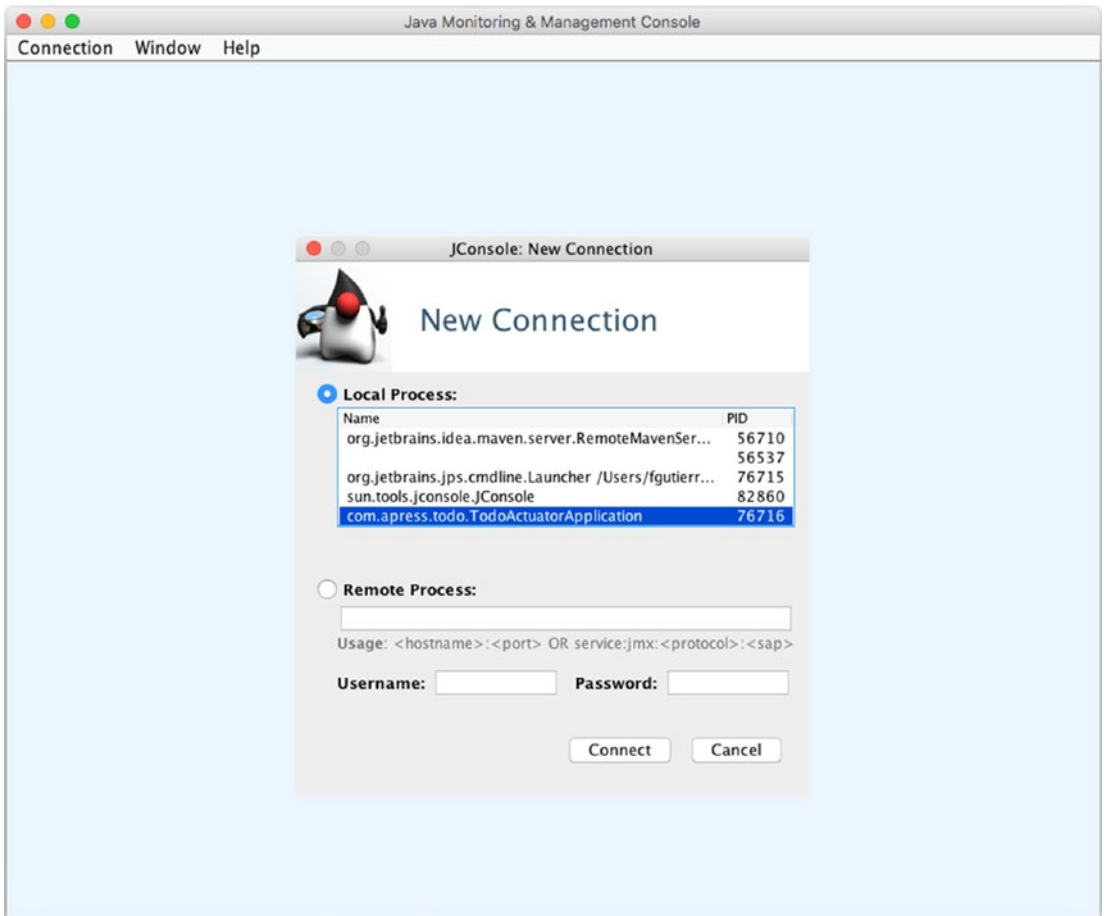


Figure 10-15. *http://localhost:8080/actuator/todo-stats*

Very easy right? But what about the other operations. Let's try them out. For this we are going to use JMX with JConsole (it comes with the JDK installation). You can open a terminal window and execute the `jconsole` command.

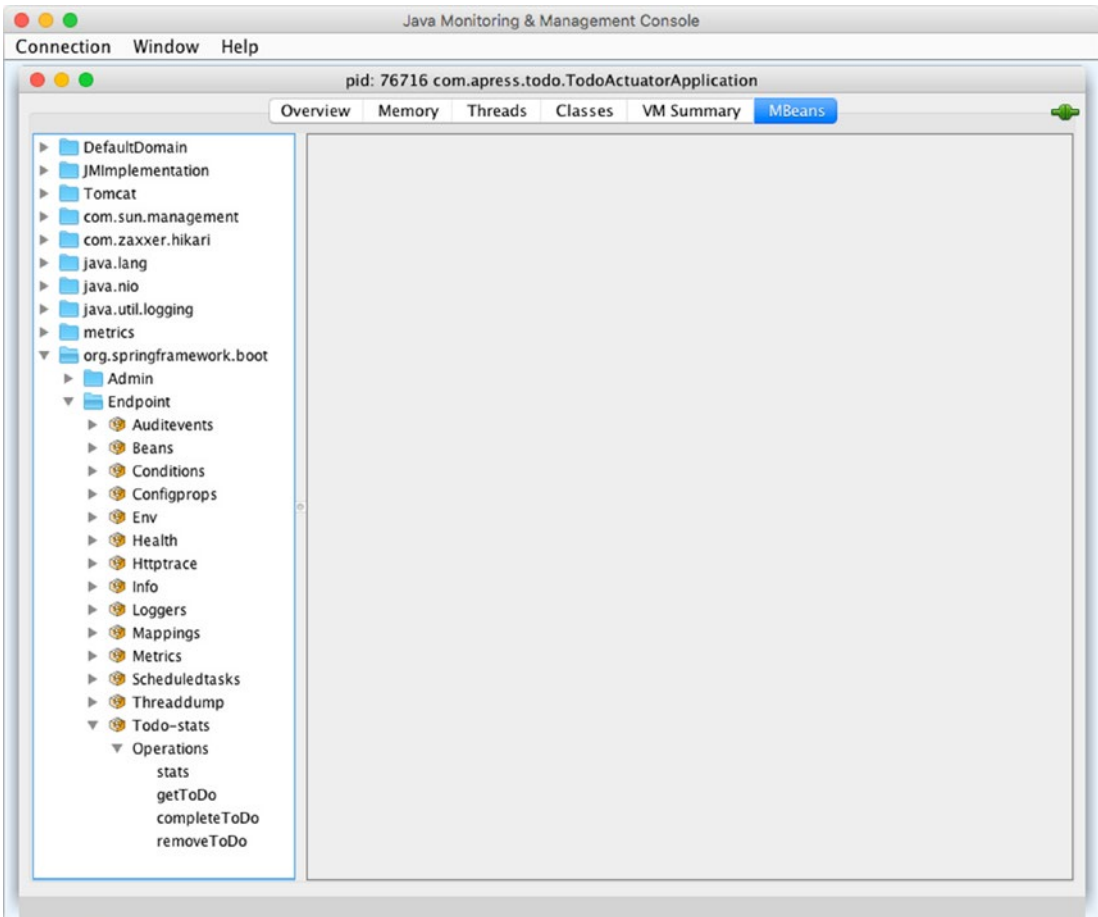
1. Select from the `com.apress.todo.ToDoActuatorApplication` list and click Connect.



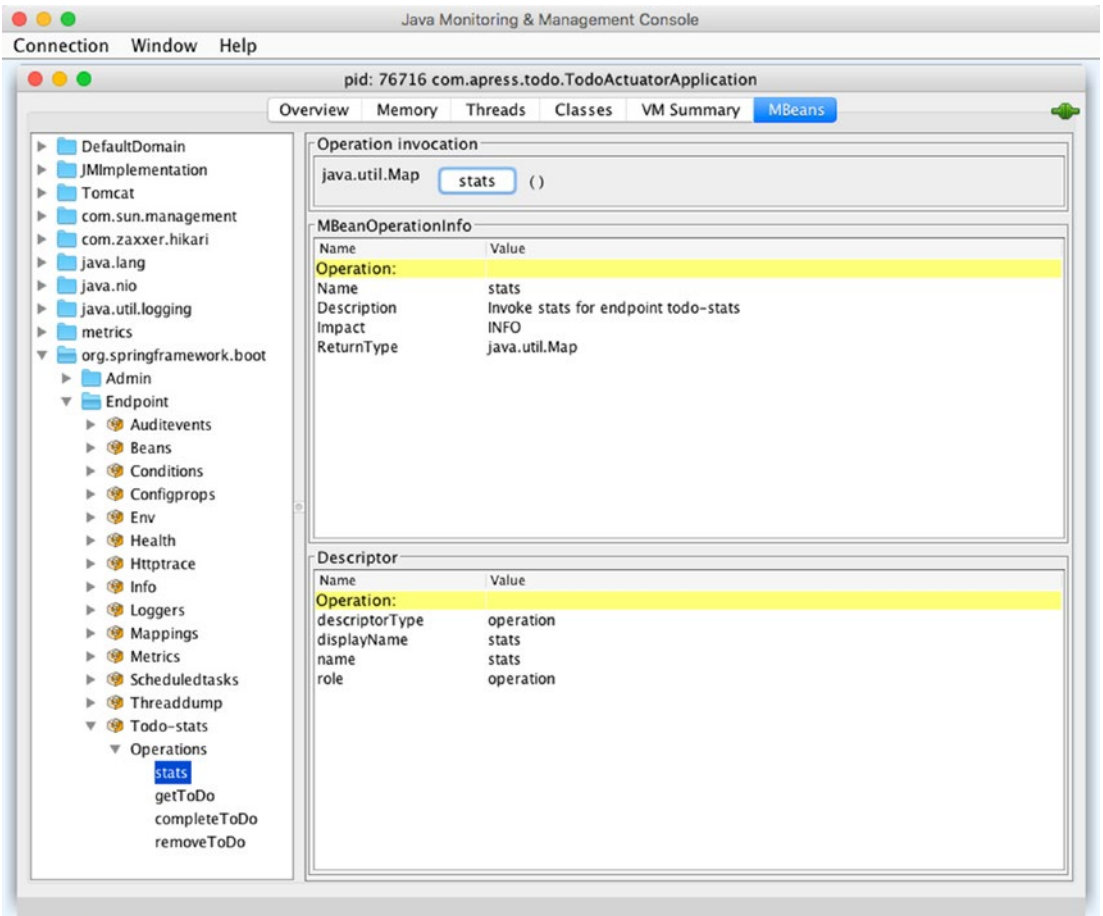
- Right now there is no secured connection, but it's OK to click the Insecure Connection button.



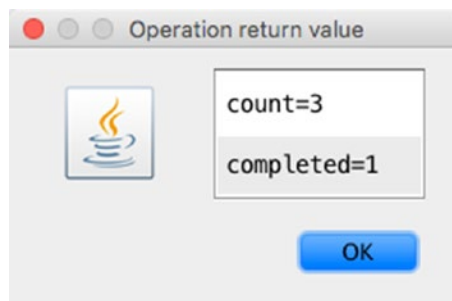
- From the main screen, select the MBeans tab. Expand the `org.springframework.boot` package and the Endpoint folder. You see the `Todo-stats`. You can expand it and see all the operations.



- Click the stats item to see the MBeans operation stats.

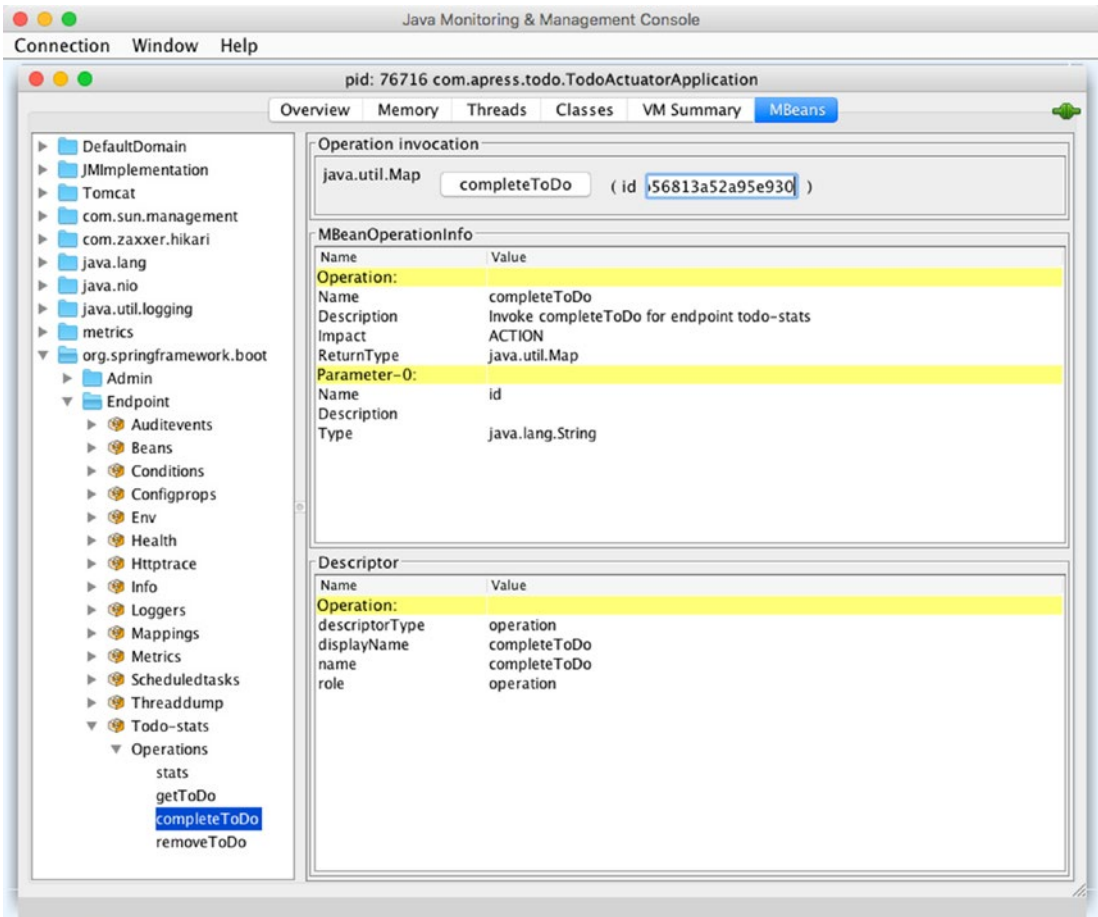


- You can click the stats button (that actually is the call to the stats method), and you will get.



As you can see, this is the same as going through the web. You can experiment with the `completeToDo` operation.

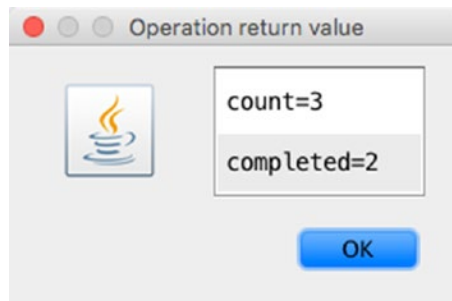
6. Click the `completeToDo` operation. On the right, fill out the ID field with `ebcf1850563c4de3b56813a52a95e930`, which is the Buy Movie Tickets `ToDo` that is not completed.



7. Click `completeToDo` to get the confirmation (an `Operation` object).



8. If you redo the stats operation, you should now see that two are completed.



As you can see, it is very easy to use JMX with the JConsole tool. You now know how to create a custom endpoint for the data that you need.

Spring Boot Actuator Health

Nowadays, we are looking for visibility in our systems, meaning that we need to monitor them closely and react to any event. I remember a long time ago, the way to monitor a server was with a simple ping; but now that's not enough. We not only monitor servers but systems and their insights. We are still required to see if our system is up, and if not, we need to get more information about the problem.

Spring Boot Actuator health endpoint to the rescue! The `/actuator/health` endpoint provides the status, or a health check, of your running application. It provides a particular property, `management.endpoint.health.show-details`, which you can use to show more information about the entire system. The following are the possible values.

- never. The details are never shown; this is the default.
- when-authorized. Details are shown only to authorized users; you can configure the roles by setting the `management.endpoint.health.roles` property.
- always. All the details are shown to all users.

Spring Boot Actuator offers the `HealthIndicator` interface that collects all information about the system; it returns a `Health` instance that contains all of this information. Actuator Health has several out-of-the-box health indicators that are auto-configured using a health aggregator to determine the final status of the system. It is very similar to a logging level.

You can up and running. Don't worry. I am going to show this with an example.

The following are some of the health indicators that are auto-configured.

- `CassandraHealthIndicator`. Checks that the Cassandra database is up and running.
- `DiskSpaceHealthIndicator`. Checks for low disk space.
- `RabbitHealthIndicator`. Checks that the Rabbit server is up and running.
- `RedisHealthIndicator`. Checks that the Redis server is up and running.
- `DataSourceHealthIndicator`. Checks for a database connection from the data source.
- `MongoHealthIndicator`. Checks that the MongoDB is up and running.
- `MailHealthIndicator`. Checks that the mail server is up.
- `SolrHealthIndicator`. Checks that the Solr server is up.
- `JmsHealthIndicator`. Checks that the JMS broker is up and running.
- `ElasticsearchHealthIndicator`. Checks that the ElasticSearch cluster is up.
- `Neo4jHealthIndicator`. Checks that the Neo4j sever is up and running.
- `InfluxDBHealthIndicator`. Checks that the InfluxDB server is up.

There are many more. All of these are auto-configured if the dependency is in your classpath; in other words, you don't need to worry about configuring or using them.

Let's test the health indicator.

1. Make sure that you have (in the `ToDo` app) the management endpoints.
`web.exposure.include=*` in the application properties file.
2. Add the `management.endpoint.health.show-details=always` property to the application properties file.
3. If you run the `ToDo` app, and access `http://localhost:8080/actuator/health`, you should get the following.



```

{
  status: "UP",
  details: {
    diskSpace: {
      status: "UP",
      details: {
        total: 500068036608,
        free: 9864806400,
        threshold: 10485760
      }
    },
    db: {
      status: "UP",
      details: {
        database: "H2",
        hello: 1
      }
    }
  }
}

```

The H2 database `DataSourceHealthIndicator` and `DiskSpaceHealthIndicator` are being auto-configured.

- 4. Add the following dependency to your pom.xml file (if you are using Maven).

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

If you are using Gradle, add the following dependency to your build.gradle file.

```
compile('org.springframework.boot:spring-boot-starter-amqp')
```

- 5. You guessed right. We are adding AMQP dependencies. Re-run the app and take a look at the /actuator/health endpoint.



Because you added the `spring-boot-starter-amqp` dependency, it is about the RabbitMQ broker, and you have the actuator, `RabbitHealthIndicator` is auto-configured to reach for a local host (or a specific broker with `spring.rabbitmq.*` properties settings). If it is alive, then it reports it. In this case, you see that some failed to connect in the logs, and in the health endpoint, you see that the system is down. If you have a RabbitMQ broker (from the previous chapter), you can run it (with the `rabbitmq-server` command) and refresh the health endpoint. You see that everything is up!



```

{
  status: "UP",
  details: {
    rabbit: {
      status: "UP",
      details: {
        version: "3.7.7"
      }
    },
    diskSpace: {
      status: "UP",
      details: {
        total: 500068036608,
        free: 11998363648,
        threshold: 10485760
      }
    },
    db: {
      status: "UP",
      details: {
        database: "H2",
        hello: 1
      }
    }
  }
}

```

That's it. This is how you can use all the out-of-the-box health indicators. Add the required dependency—and you got it!

ToDo App with Custom HealthIndicator

Now it's the ToDo's app turn to have its own custom health indicator. It's very easy to implement one. You need to implement the `HealthIndicator` interface and return the desired state with a `Health` instance.

Create `ToDoHealthCheck` to visit a `FileSystem` path and perform a check up to see if it is available, readable, and writable (see Listing 10-5).

Listing 10-5. `com.apress.todo.actuator.ToDoHealthCheck.java`

```
package com.apress.todo.actuator;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.actuate.health.Health;
import org.springframework.stereotype.Component;

import java.io.File;

@Component
public class ToDoHealthCheck implements HealthIndicator {

    private String path;

    public ToDoHealthCheck(@Value("${todo.path:/tmp}")String path){
        this.path = path;
    }

    @Override
    public Health health() {

        try {

            File file = new File(path);
            if(file.exists()){

                if(file.canWrite())
                    return Health.up().build();

                return Health.down().build();

            }else{
```


Listing 10-7. com.apress.todo.config.ToDoConfig.java

```

package com.apress.todo.config;

import org.springframework.boot.context.properties.
EnableConfigurationProperties;
import org.springframework.context.annotation.Configuration;

@EnableConfigurationProperties(ToDoProperties.class)
@Configuration
public class ToDoConfig {
}

```

There is nothing extraordinary in this class. Add the `application.properties` file in the new property.

```
todo.path=/tmp/todo
```

If you are using Windows, you can play around with something like

```
todo.path=C:\\tmp\\todo
```

Review the documentation for the right characters. So, you are all set. If you re-run your `ToDo` app, and take a look at the response of the `/actuator/health` you should get the following.



```
{
  status: "OUT_OF_SERVICE",
  details: {
    todoHealthCheck: {
      status: "OUT_OF_SERVICE"
    },
    diskSpace: {
      status: "UP",
      details: {
        total: 500068036608,
        free: 8785756160,
        threshold: 10485760
      }
    },
    db: {
      status: "UP",
      details: {
        database: "H2",
        hello: 1
      }
    }
  }
}
```

You have the `todoHealthCheck` key in the JSON response; and it match with the logic set. Next, fix the issue by making a writeable `/tmp/todo` directory and if you refresh the page you get.



```
{
  status: "UP",
  details: {
    toDoHealthCheck: {
      status: "UP"
    },
    diskSpace: {
      status: "UP",
      details: {
        total: 500068036608,
        free: 9855713280,
        threshold: 10485760
      }
    },
    db: {
      status: "UP",
      details: {
        database: "H2",
        hello: 1
      }
    }
  }
}
```

You can configure the status/severity order (e.g., a logging level) by using the following property in your application.properties file.

management.health.status.order=FATAL, DOWN, OUT_OF_SERVICE, UNKNOWN, UP

If you are using the health endpoint over HTTP, every status/severity has its own HTTP code or mapping available.

- DOWN - 503
- OUT_OF_SERVICE - 503
- UP - 200
- DOWN - 200

You can use your own code by using

```
management.health.status.http-mapping.FATAL=503
```

Also, you can create your own status, like `IN_BAD_CONDITION`, by using `Health.status("IN_BAD_CONDITION").build();`

Creating a custom health indicator with Spring Boot Actuator is very easy!

Spring Boot Actuator Metrics

Nowadays, every system is required to be monitored. It is necessary to keep visibility by watching what is happening in every application, both individually and as a whole. Spring Boot Actuator offers basic metrics and integration and auto-configuration with Micrometer (<http://micrometer.io>).

Micrometer provides a simple face over instrumentation clients for many popular monitoring systems; in other words, you can write the monitoring code once, and use any other third-party system, such as Prometheus, Netflix Atlas, CloudWatch, Datadog, Graphite, Ganglia, JMX, InfluxDB/Telegraf, New Relic, StatsD, SignalFX, and WaveFront (and more coming).

Remember that Spring Boot Actuator has `/actuator/metrics`. If you run the `ToDo` app, you get the basic metrics; you learned that in previous sections. What I haven't shown you is how to create your custom metrics using Micrometer. The idea is to write the code once and use any other third-party monitoring tool. Spring Boot Actuator and Micrometer expose those metrics to the chosen monitoring tool.

Let's jump directly to implementing Micrometer code and use Prometheus and Grafana to see how easy it is to use.

ToDo App with Micrometer: Prometheus and Grafana

Let's implement Micrometer code and use Prometheus and Grafana.

So far we have seen that the Spring Data REST module creates web MVC controllers (REST endpoints) on our behalf once it sees all the interfaces that extend the `Repository<T, ID>` interface. Imagine for a moment that we need to intercept those web requests and start creating an aggregate; a metric that tells us how many times a particular REST endpoint and HTTP method have been requested. This helps us to

identify which endpoint is a candidate for a microservice. This interceptor also gets all the requests, including the /actuator ones.

To do this, Spring MVC offers a `HandlerInterceptor` interface that we can use. It has three default methods, but we only need one of them. Let's start by creating the `ToDoMetricInterceptor` class (see Listing 10-8).

Listing 10-8. `com.apress.todo.interceptor.ToDoMetricInterceptor.java`

```
package com.apress.todo.interceptor;

import io.micrometer.core.instrument.MeterRegistry;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.web.servlet.HandlerInterceptor;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class ToDoMetricInterceptor implements HandlerInterceptor {

    private static Logger log = LoggerFactory.
        getLogger(ToDoMetricInterceptor.class);

    private MeterRegistry registry;
    private String URI, pathKey, METHOD;

    public ToDoMetricInterceptor(MeterRegistry registry) {
        this.registry = registry;
    }

    @Override
    public void afterCompletion(HttpServletRequest request,
HttpServletResponse response, Object handler, Exception ex)
throws Exception {
        URI = request.getRequestURI();
        METHOD = request.getMethod();
        if (!URI.contains("prometheus")){
            log.info(" >> PATH: {}",URI);
            log.info(" >> METHOD: {}", METHOD);
        }
    }
}
```



```

        pathKey = "api_".concat(METHOD.toLowerCase()).concat(URI.
        replaceAll("/", "_").toLowerCase());
        this.registry.counter(pathKey).increment();
    }
}

```

Listing 10-8 shows the `ToDoMetricInterceptor` class (it is implementing the `HandlerInterceptor` interface). This interface has three default methods: `preHandle`, `postHandle`, and `afterCompletion`. This class only implements the `afterCompletion` method. This method has `HttpServletRequest`, which is helpful for discovering which endpoint and HTTP method have been requested.

Your class is using the `MeterRegistry` instance, which is part of the `Micrometer Framework`. The implementation gets the path and the method from the request instance and uses the `counter` method to get incremented. The `pathKey` is very simple; if there is a GET request to the `/todos` endpoint, the `pathKey` is `api_get_todos`. If there is a POST request to the `/todos` endpoint, the `pathKey` is `api_post_todos`, and so forth. So, if there are several requests to the `/todos`, the `registry` increments (using that `pathKey`) and aggregates to the existing value.

Next, let's make sure that `ToDoMetricInterceptor` is being picked up and configured by Spring MVC. Open the `ToDoConfig` class and add a `MappedInterceptor` bean (see Listing 10-9 for version 2).

Listing 10-9. `com.apress.todo.config.ToDoConfig.java v2`

```

package com.apress.todo.config;

import com.apress.todo.interceptor.ToDoMetricInterceptor;
import io.micrometer.core.instrument.MeterRegistry;
import org.springframework.boot.context.properties.
EnableConfigurationProperties;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.handler.MappedInterceptor;

@EnableConfigurationProperties(ToDoProperties.class)
@Configuration
public class ToDoConfig {

```

```

@Bean
public MappedInterceptor metricInterceptor(MeterRegistry registry) {
    return new MappedInterceptor(new String[]{"/**"},
        new ToDoMetricInterceptor(registry));
}
}

```

Listing 10-9 shows the new `ToDoConfig` class, which has `MappedInterceptor`. It uses `ToDoMetricInterceptor` for every request by using the `"/**"` matcher.

Next, let's add two dependencies that export data to JMX and Prometheus. If you have Maven, you can add the following dependencies to the `pom.xml` file.

```

<dependency>
  <groupId>io.micrometer</groupId>
  <artifactId>micrometer-registry-jmx</artifactId>
</dependency>
<dependency>
  <groupId>io.micrometer</groupId>
  <artifactId>micrometer-registry-prometheus</artifactId>
</dependency>

```

If you are using Gradle, you can add the following dependencies to `build.gradle`.

```

compile('io.micrometer:micrometer-registry-jmx')
compile('io.micrometer:micrometer-registry-prometheus')

```

Spring Boot Actuator auto-configures and registers every Micrometer registry, and in this case, JMX and Prometheus. For Prometheus, the actuator configures the `/actuator/prometheus` endpoint.

Prerequisites: Using Docker

Before testing the `ToDo` app with the metrics, it is necessary to install Docker (try to install the latest release).

- Install Docker CE (Community Edition) from <https://docs.docker.com/install/>
- Install Docker Compose from <https://docs.docker.com/compose/install/>

Why did I choose Docker? Well, it is an easy way to install what we need. And we are going to use it again in the following chapters. Docker Compose facilitates installing Prometheus and Grafana by using Docker's internal network that allows us to use DNS names.

docker-compose.yml

This is the `docker-compose.yml` file used to start up Prometheus and Grafana.

```
version: '3.1'

networks:
  micrometer:

services:

  prometheus:
    image: prom/prometheus
    volumes:
      - ./prometheus:/etc/prometheus/
    command:
      - '--config.file=/etc/prometheus/prometheus.yml'
      - '--storage.tsdb.path=/prometheus'
      - '--web.console.libraries=/usr/share/prometheus/console_libraries'
      - '--web.console.templates=/usr/share/prometheus/conssoles'
    ports:
      - 9090:9090
    networks:
      - micrometer
    restart: always

  grafana:
    image: grafana/grafana
    user: "104"
    depends_on:
      - prometheus
    volumes:
      - ./grafana:/etc/grafana/
```

```

ports:
  - 3000:3000
networks:
  - micrometer
restart: always

```

You can create this file with any editor. Remember, it is a YAML file and there are no tab spaces for indentation. You need to create two folders: `prometheus` and `grafana`. In each folder, there is a file.

In the `prometheus` folder, there is a `prometheus.yml` file with the following content.

```

global:
  scrape_interval:    5s
  evaluation_interval: 5s

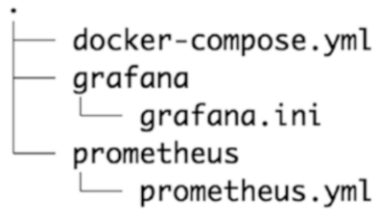
scrape_configs:
  - job_name: 'todo-app'

    metrics_path: '/actuator/prometheus'
    scrape_interval: 5s
    static_configs:
      - targets: ['host.docker.internal:8080']

```

The important thing about this file is the `metrics_path` and `targets` keys. When it finds out about `micrometer-registry-prometheus`, Spring Boot Actuator auto-configures the `/actuator/prometheus` endpoint. This value is necessary for `metrics_path`. Another very important value is the `targets` key. Prometheus scrapes the `/actuator/prometheus` endpoint every 5 seconds. It needs to know where it is located (it is using the `host.docker.internal` domain name). This is the part of Docker that looks for its host (the `localhost:8080/todo-actuator` app that is running).

The `grafana` folder contains an empty `grafana.ini` file. To make sure Grafana takes the default values, you should have the following directory structure.



2 directories, 3 files

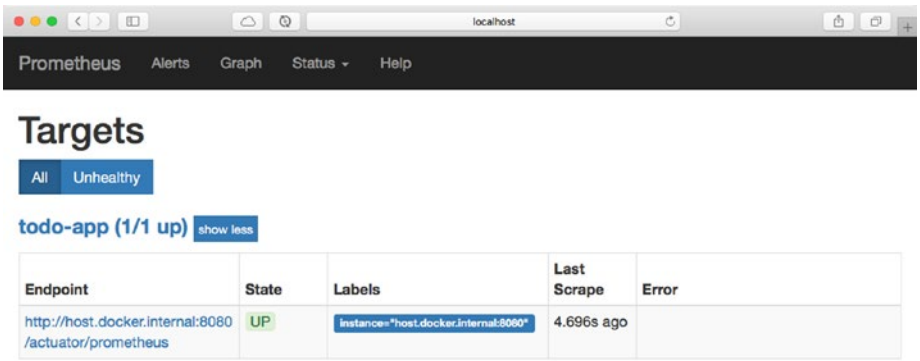
Running the ToDo App Metrics

Now, it's time to start testing and configuring Grafana to see useful metrics. Run your ToDo app. Check out the logs and make sure that the `/actuator/prometheus` endpoint is there.

Open a terminal, go where you have the `docker-compose.yml` file, and execute the following command.

```
$ docker-compose up
```

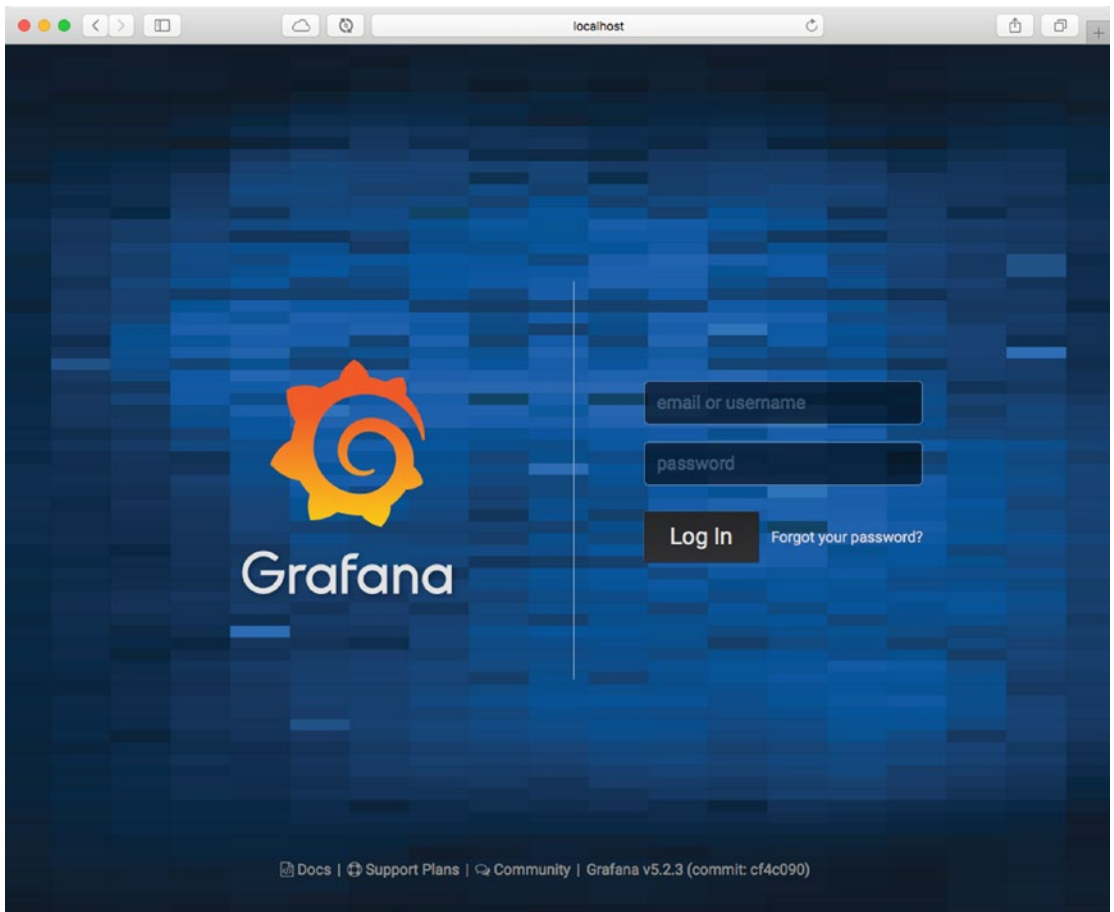
This command line starts the `docker-compose` engine, and it downloads the images and runs them. Let's make sure that the Prometheus app works by opening a browser and hitting `http://localhost:9090/targets`.



This means that the `prometheus.yml` configuration was successfully taken. In other words, Prometheus is scraping the `http://localhost:8080/actuator/prometheus` endpoint.

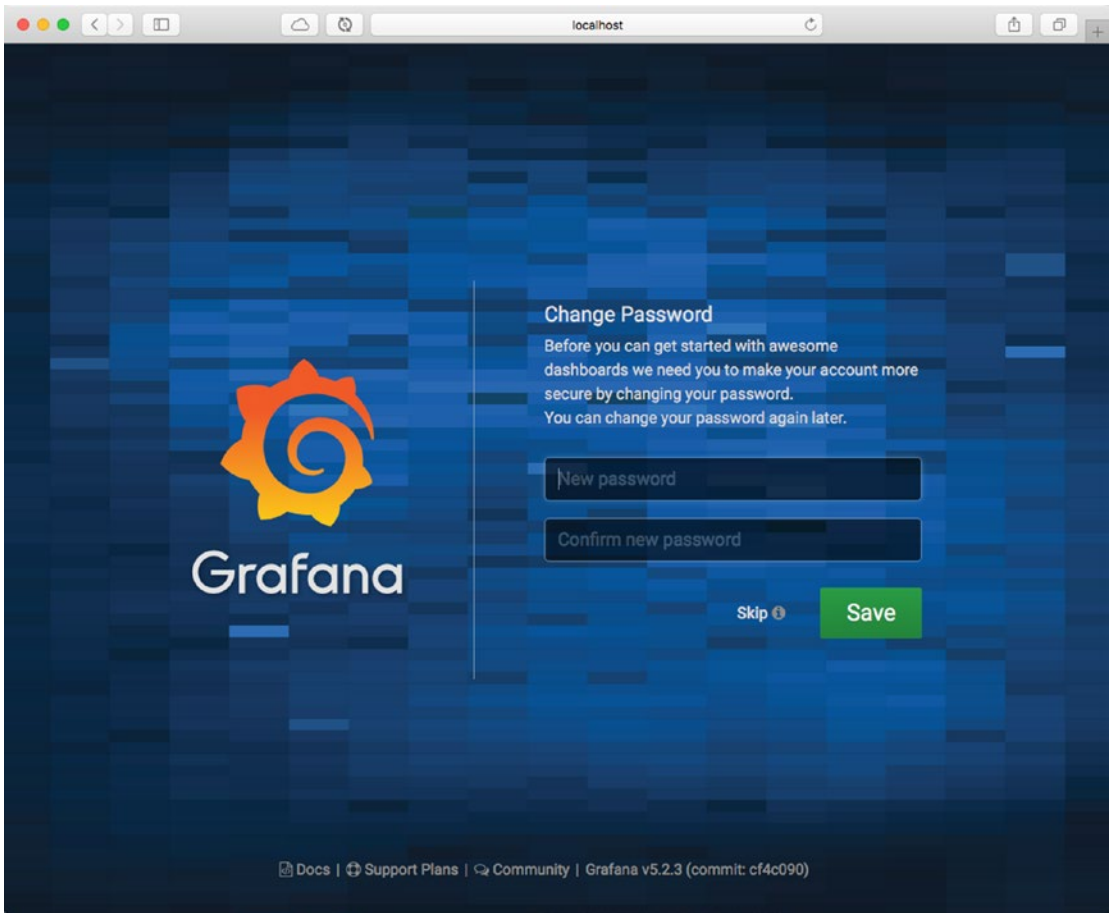
Next, let's configure Grafana.

1. Open another browser tab and hit `http://localhost:3000`.

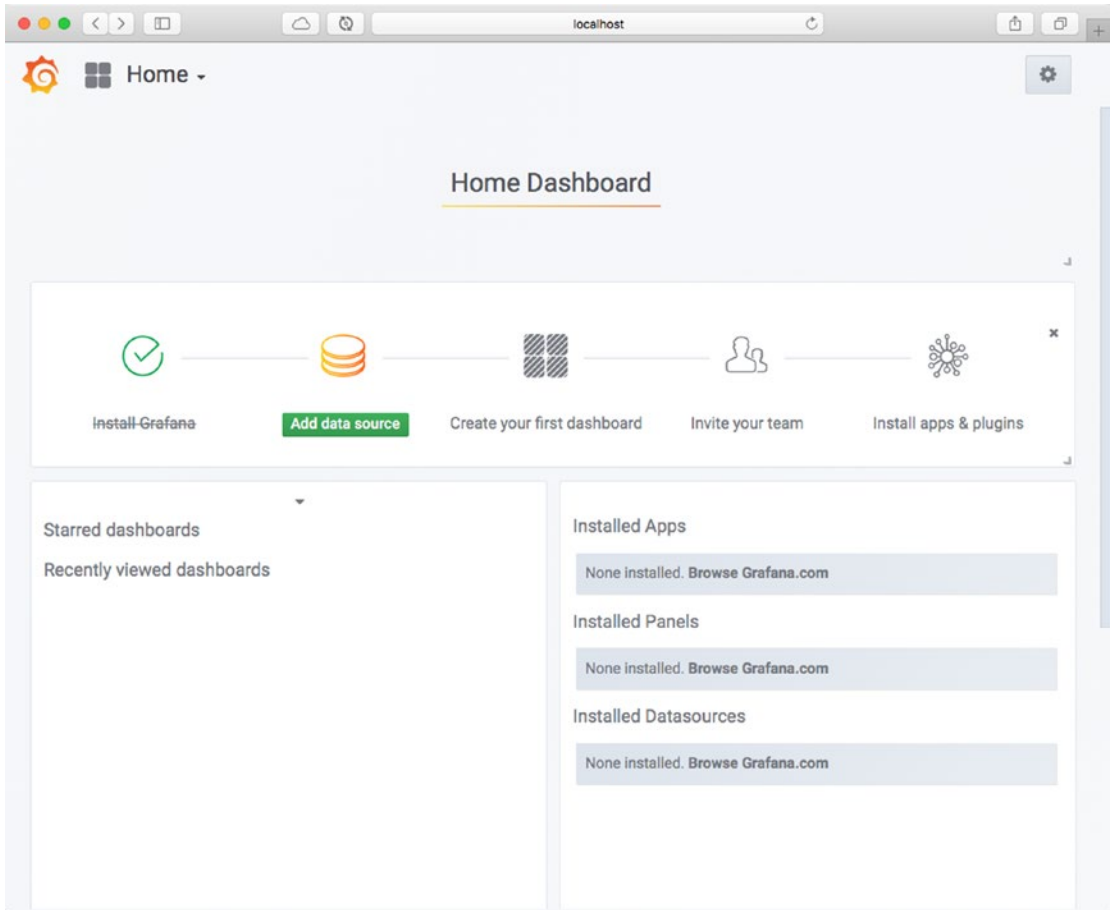


You can use admin/admin as credentials.

2. You can press the Skip button the following screen.



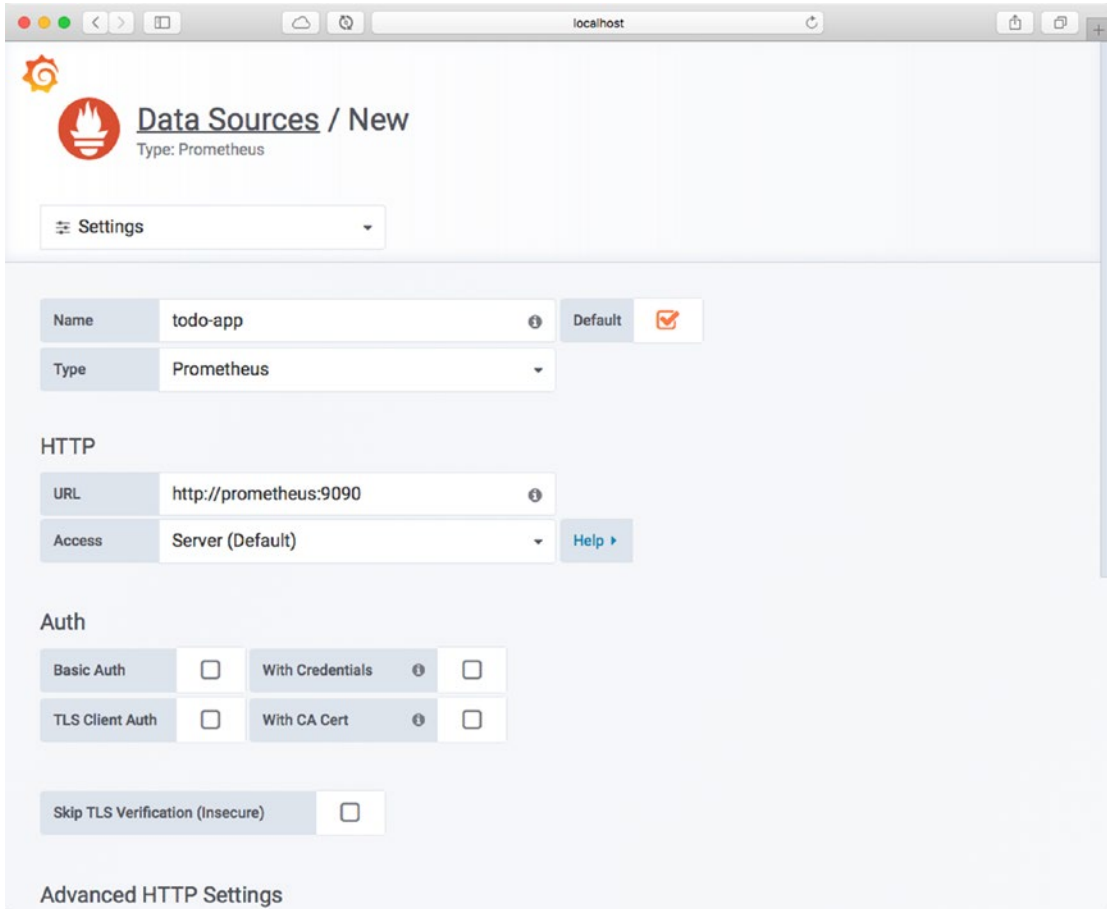
3. Click the Add Data Source icon.



4. Fill out all the required fields. The important ones are

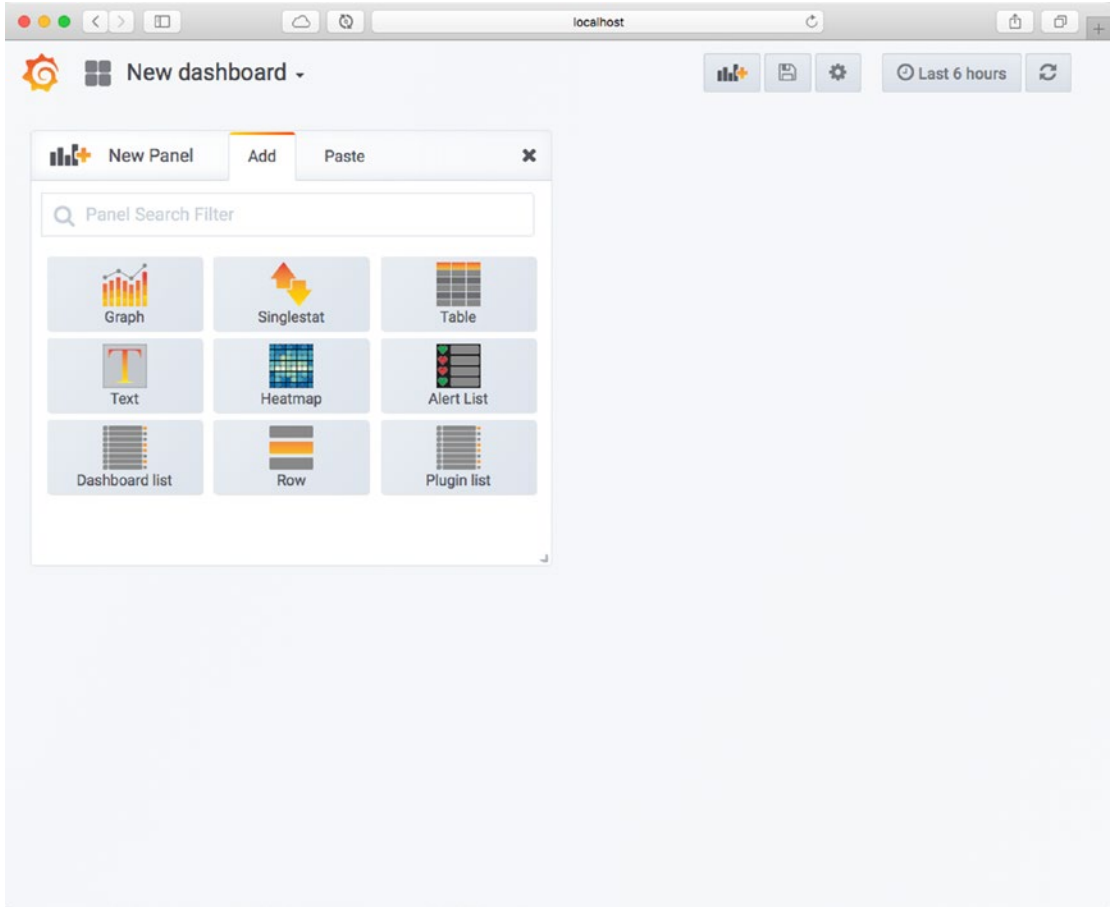
- Name: todo-app
- Type: Prometheus
- URL: `http://prometheus:9090`
- Scrape interval: 3s

- 5. Click the Save button.

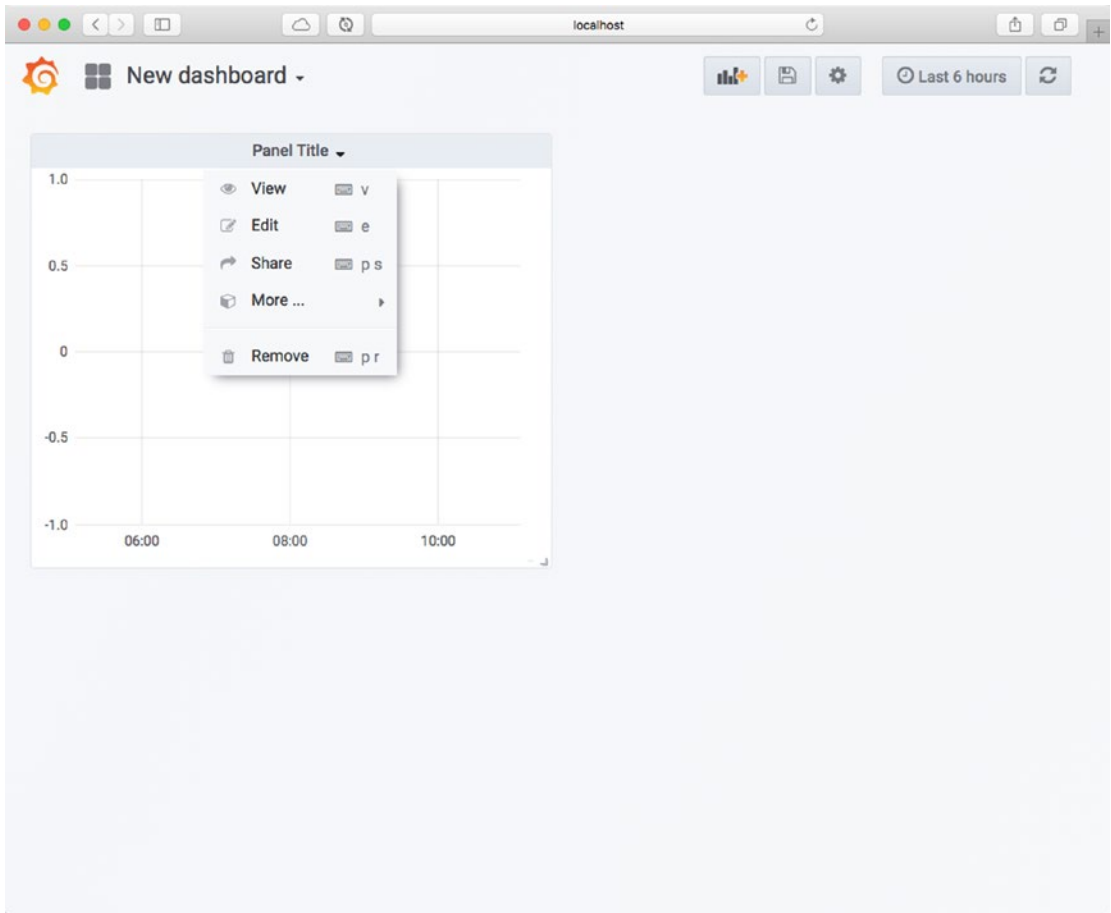


The URL value, `http://prometheus:9090`, refers to the docker-compose service, which is the internal DNS that docker-compose offers, so no need to do local host. You can leave the other values by default and click Save & Test. If everything works as expected, you see a green banner saying, *Data source is working* at the bottom of the page.

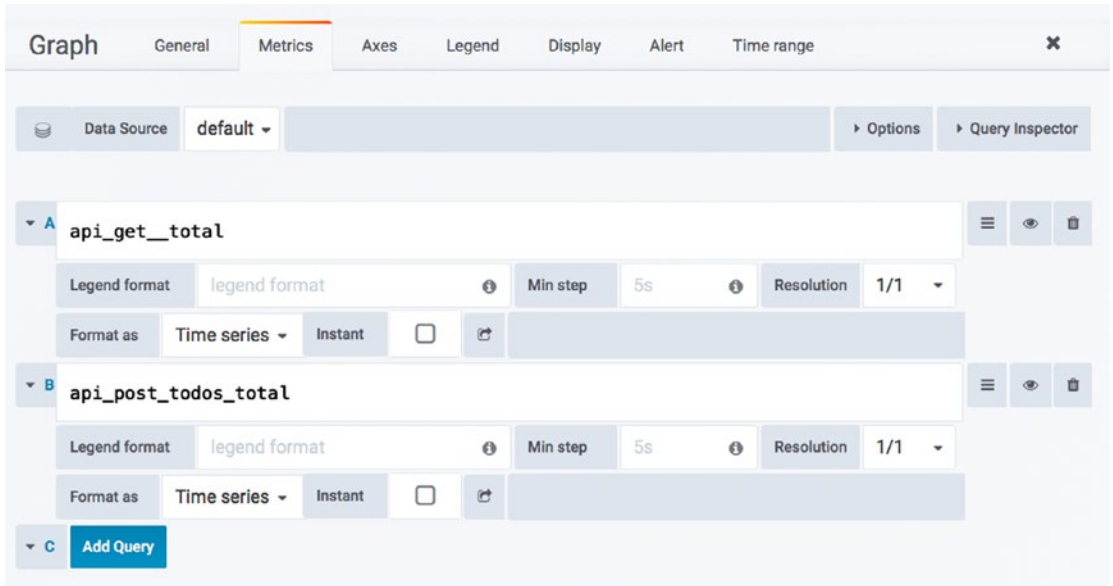
6. You can go home by going back or pointing the browser to `http://localhost:3000`. You can click the New Dashboard button on the homepage.



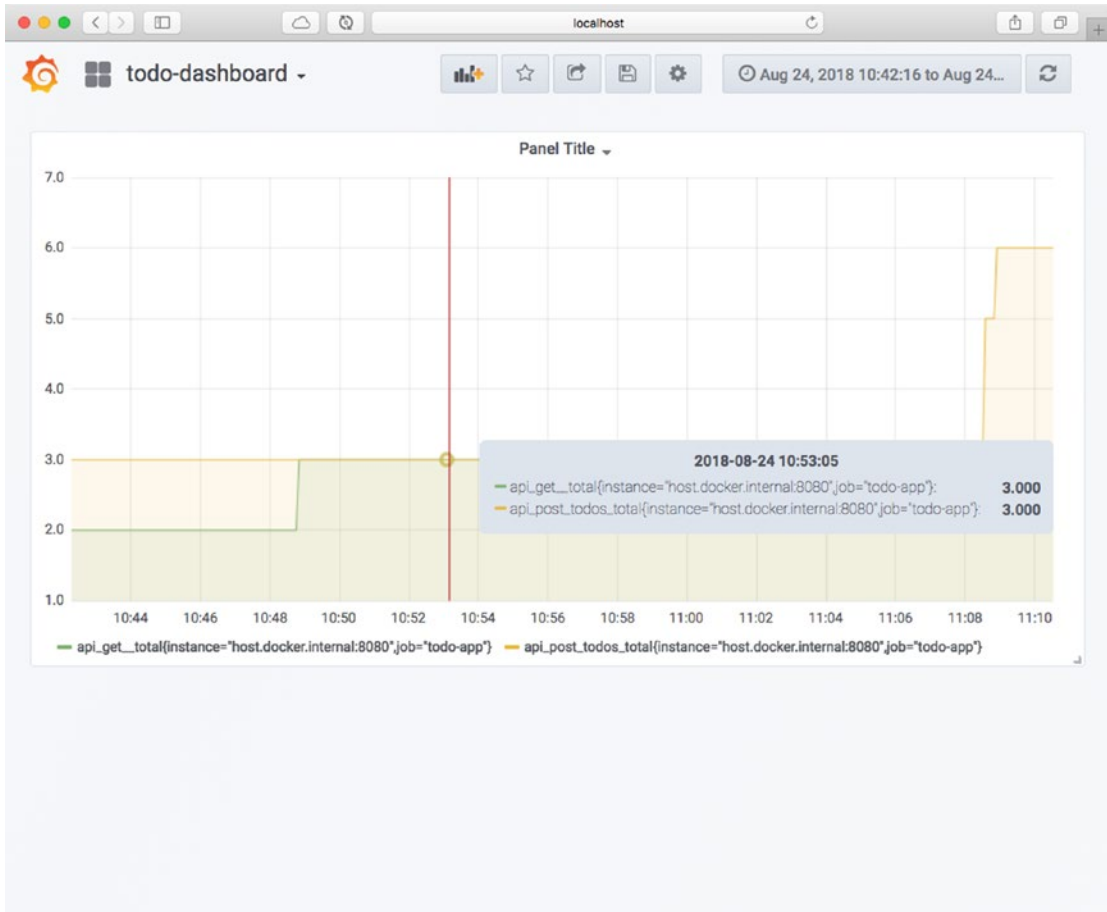
- 7. You can click the Graph icon, a panel appears. Click Panel Title and then click Edit.



- 8. Configure two queries, the `api_get_total` and `api_post_todos_total`, which were generated as metrics by the Micrometer and Spring Boot Actuator for Prometheus engine.



9. Perform requests to the `/todos` (several times) and post to the `/todos` endpoints. You see something like the next figure.



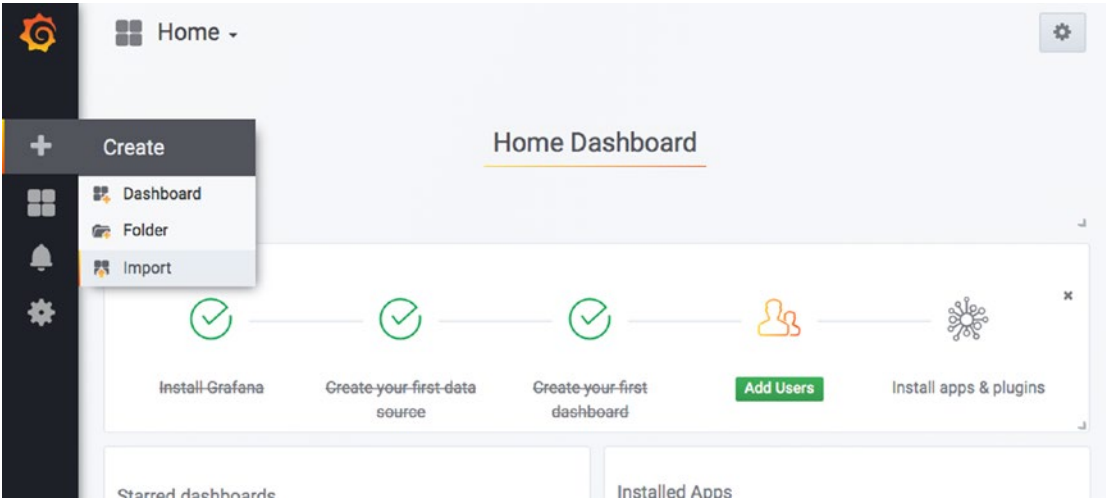
Congratulations! You have created custom metrics using Micrometer, Prometheus, and Grafana.

General Stats for Spring Boot with Grafana

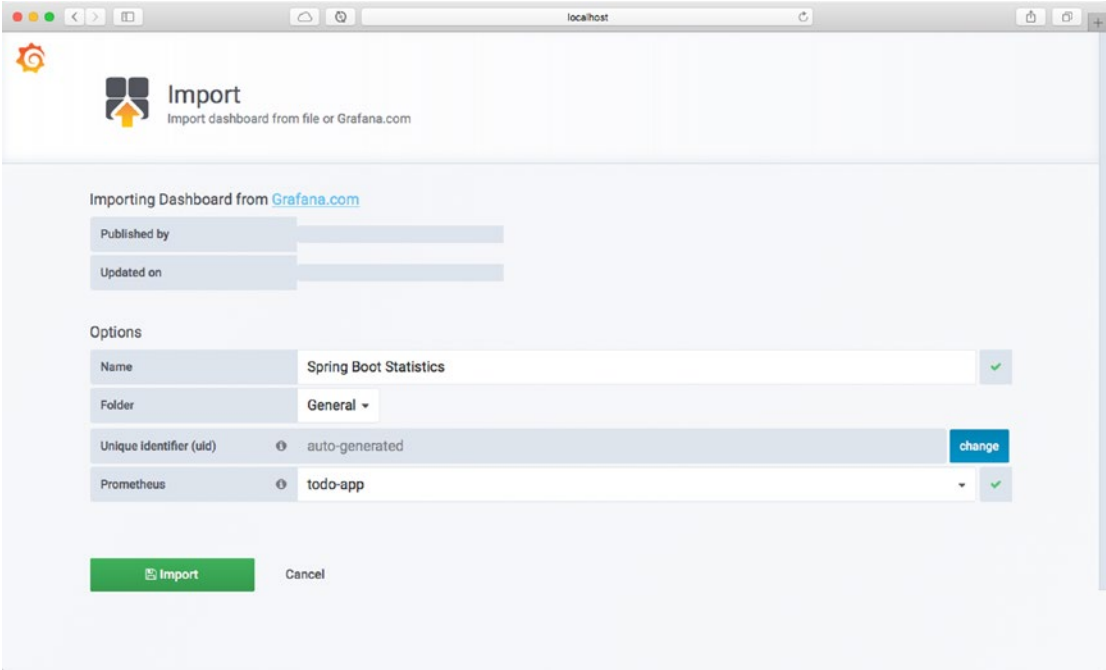
I found a very useful configuration for Grafana that allows you to take advantage of every metric that Spring Boot Actuator exposes. This configuration can be imported into Grafana.

Download this configuration from <https://grafana.com/dashboards/6756>. The file name is `spring-boot-statistics_rev2.json`. You need it next.

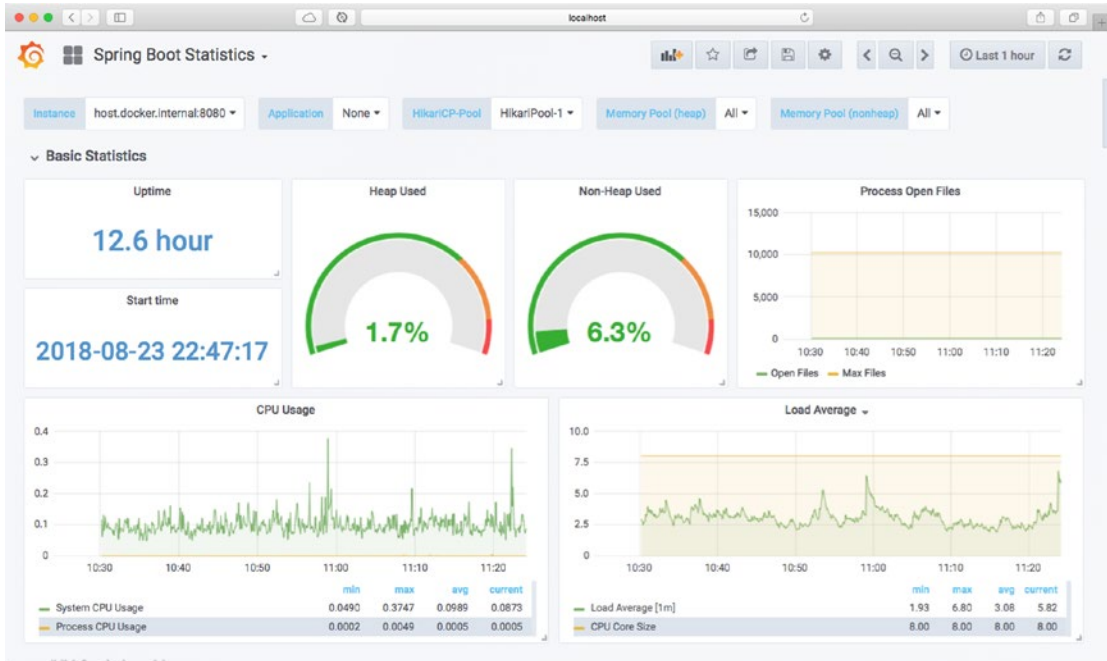
In the left corner of the Grafana homepage (<http://localhost:3000>), click the Grafana icon, which opens a side bar. Click the + symbol and choose Import.



Level the default value, but in the Prometheus field, choose todo-app (the data source that you configured earlier).



Click Import—and *voilà!* You have a complete dashboard with all the Spring Boot Actuator metrics and monitoring!



Take a moment to review every single graph. All the data comes from the /actuator/prometheus endpoint.

You can shut down docker-compose by executing the following in another terminal window.

```
$ docker-compose down
```

Note You can find the solution to this section in the book’s source code on the Apress website or on GitHub at <https://github.com/Apress/pro-spring-boot-2>, or on my personal repository at <https://github.com/felipeg48/pro-spring-boot-2nd>.

Summary

This chapter discussed Spring Boot Actuator, including its endpoints and how customizable it is. With the Actuator module, you can monitor your Spring Boot application, from using the `/health` endpoint to using `/httptrace` for more granular debugging.

You learned that you can use Micrometer and plug in any third-party tool to use the Spring Boot actuator metrics.

In the next chapter, you take a step forward and see how to use Spring Integration and Spring Cloud Stream.

CHAPTER 11

Spring Integration and Spring Cloud Stream with Spring Boot

In this chapter, I show you one of the best integration frameworks for the Java community: the Spring Integration project, which is based on the Spring Framework. I also present the Spring Cloud Stream, which is based on Spring Integration. It creates robust and scalable event-driven microservices connected to shared messaging systems—all done with Spring Boot.

If we take a look at software development and business needs, as a developer or an architect, we are always looking at how to integrate components and systems, either internal or external to our architecture, and probe what is fully functional, highly available, and easy to maintain and enhance.

The following are the main uses cases that developers or architects typically face.

- Creating a system that does a reliable file transfer or file analysis. Most of the applications out there need to read information from a file and then process it, so we need to create robust file systems that save and read data but also share and deal with the size of the files.
- The ability to use data in a shared environment where multiple clients (systems or users) need access to the same database or the same table and do operations and deal with inconsistency, duplication, and more.
- Remote access to different systems, from executing remote procedures, to sending a lot of information. We always want to have this in real time and in an asynchronous way. The ability to get a

response as fast as possible without forgetting that the remote system always needs to be reachable; in other words, have the fault tolerance and high availability required.

- **Messaging**—from a basic internal call to billions of messages per second to remote brokers. Normally, we do messaging in an asynchronous way, so we need to deal with concurrency, multithreading, speed (network latency), high availability, fault tolerance, and so forth.

How can we solve or implement all of these use cases? Almost 15 years ago, software engineers Gregor Hohpe and Bobby Woolf wrote *Enterprise Integration Patterns: Designing, Building and Deploying Messaging Solutions* (Addison-Wesley, 2003). This book exposes all the messaging patterns needed to solve the use cases that I mentioned. It gives a better understanding on how systems interconnect and work, and how you can create a robust integration system with application architecture, object-oriented design, and message-oriented.

In the following sections, I'll show you some of these patterns using the Spring Integration project from the Spring Framework.

Spring Integration Primer

Spring Integration is a simple model for implementing enterprise integration solutions. It facilitates the asynchronous and message-driven within a Spring Boot application. It implements all the enterprise integration patterns for creating enterprise, robust, and portable integration solutions.

The Spring Integration project offers a way to have components that are loosely coupled for modularity and testability. It helps to enforce the separation of concerns between your business and integration logic.

Spring Integration exposes the following main components.

- *Message*. This is a generic wrapper for any Java object. It consists of headers and a payload. The headers normally have important information like ID, timestamp, correlation ID, and return address; and of course, you can add your own. The payload can be any type of data, from an array of bytes to custom objects. You can find its definition in the spring-messaging module in the `org.springframework.messaging` package.

```
public interface Message<T> {
    T getPayload();
    MessageHeaders getHeaders();
}
```

As you can see, there's nothing fancy in the definition.

- *Message channel.* Pipes and filters architecture, very similar to the command you use in a UNIX system. To use it, you need to have producers and consumers; the producer sends the message to the message channel and a consumer receives it (see Figure 11-1).

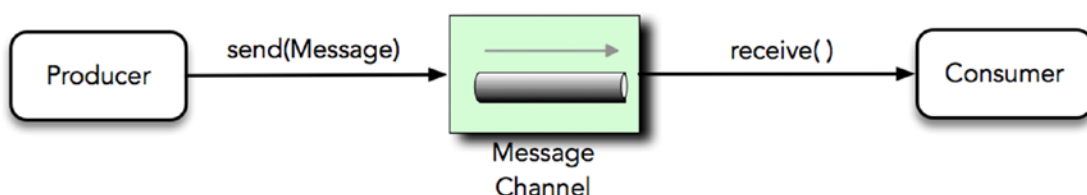


Figure 11-1. Message channel

This message channel follows messaging patterns, such as the Point-to-Point and Publish/Subscribe models. Spring Integration offers several message channels, like pollable channels (that allows you to have buffering messages within a queue) or subscribable channels for the consumers.

- *Message endpoint.* A filter that connects the application code to the messaging framework. Most of these endpoints are part of the *Enterprise Integration Patterns* implementations.
 - *Filter.* A message filter determines when a message should be passed to the output channel.
 - *Transformer.* A message transformer modifies the content or structure of a message and passes it to the output channel.
 - *Router.* A message router decides what to do and where to send the message based on rules. These rules can be in the headers or in the same payload. This message router has many patterns that can be applied. I'll show you at least one of them.

- *Splitter*. A message splitter accepts a message (input channel), and it splits and returns new multiple messages (output channel).
- *Service activator*. This is an endpoint that acts as a service by receiving a message (input channel) and processes it. It can either end the flow of the integration or it can return the same message or an entirely new one (output channel).
- *Aggregator*. This message endpoint received multiple messages (input channel); it combines them in a new single message (base on a release strategy) and sends it out (output channel).
- *Channel adapters*. This is a particular endpoint that connects a message channel to other systems or transports. Spring Integration offers inbound or outbound adapters. Where a response is required, it offers a gateway adapter. You see that these are the most commonly used. Why? If your solution is looking to connect to RabbitMQ, JMS, FTP, a File System, HTTP or any other technology, Spring Integration has the adapter to connect to it without you coding any client.

It would take a whole new book to write about Spring Integration and message patterns, messaging channels, adapters, and more, but if you are interesting in this technology, I recommend *Pro Spring Integration* by Dr. Mark Lui (Apress, 2011).

In the next section, I'll show you some of the components and patterns, which are enough to get you started.

Programming Spring Integration

With Spring Integration, there are several ways to configure all the components (message, message channel and message endpoints): XML, JavaConfig classes, annotations, and the new Integration DSL.

ToDo App with Spring Integration

Let's start with the well-known ToDo App and use Spring Integration right away. You can start from scratch or you can follow along in the next sections to learn what you need to do. If you are starting from scratch, then you can go to Spring Initializr (<https://start.spring.io>) and add the following values to the fields.

- Group: com.apress.todo
- Artifact: todo-integration
- Name: todo-integration
- Package Name: com.apress.todo
- Dependencies: Spring Integration, Lombok

You can select either Maven or Gradle as the project type. Then you can press the Generate Project button, which downloads a ZIP file. Uncompress it and import the project in your favorite IDE (see Figure 11-2).

The screenshot shows the Spring Initializr web application interface. At the top, it says "SPRING INITIALIZR bootstrap your application now". Below this, there is a header "Generate a" followed by a dropdown menu set to "Maven Project", the word "with", another dropdown menu set to "Java", and "and Spring Boot" followed by a dropdown menu set to "2.0.4".

The main content is divided into two columns:

- Project Metadata:** This section contains several input fields:
 - Artifact coordinates:** Group: com.apress.todo; Artifact: todo-integration; Name: todo-integration; Description: Demo project for Spring Boot; Package Name: com.apress.todo.
 - Packaging:** A dropdown menu set to "Jar".
 - Java Version:** A dropdown menu set to "8".
- Dependencies:** This section includes a search bar with the text "Web, Security, JPA, Actuator, Devtools...". Below the search bar, there is a section titled "Selected Dependencies" which shows two green buttons: "Spring Integration" and "Lombok", each with a small 'x' icon to its right.

At the bottom of the form, there is a green button labeled "Generate Project" with a small icon of a gear and a plus sign.

At the bottom left of the form, there is a link: "Too many options? [Switch back to the simple version.](#)"

Figure 11-2. *Spring Initializr*

As you can see from the dependencies, we are now using Spring Integration. You can reuse or copy the `ToDo` class (see Listing 11-1).

Listing 11-1. `com.apress.todo.domain.ToDo.java`

```
package com.apress.todo.domain;

import lombok.Data;
import java.time.LocalDateTime;
import java.util.UUID;

@Data
public class ToDo {

    private String id;
    private String description;
    private LocalDateTime created;
    private LocalDateTime modified;
    private boolean completed;

    public ToDo(){
        this.id = UUID.randomUUID().toString();
        this.created = LocalDateTime.now();
        this.modified = LocalDateTime.now();
    }

    public ToDo(String description){
        this();
        this.description = description;
    }

    public ToDo(String description,boolean completed){
        this(description);
        this.completed = completed;
    }
}
```

Listing 11-1 shows you well-known `ToDo` class. There's nothing new about it. Next, let's create a `ToDoIntegration` class that has the first Spring Integration flow using DSL (see Listing 11-2).

Listing 11-2. `com.apress.todo.integration.ToDoIntegration.java`

```
package com.apress.todo.integration;

import com.apress.todo.domain.ToDo;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.integration.channel.DirectChannel;
import org.springframework.integration.config.EnableIntegration;
import org.springframework.integration.dsl.IntegrationFlow;
import org.springframework.integration.dsl.IntegrationFlows;
import org.springframework.integration.dsl.channel.MessageChannels;

@EnableIntegration
@Configuration
public class ToDoIntegration {

    @Bean
    public DirectChannel input(){
        return MessageChannels.direct().get();
    }

    @Bean
    public IntegrationFlow simpleFlow(){
        return IntegrationFlows
            .from(input())
            .filter(ToDo.class, ToDo::isCompleted)
            .transform(ToDo.class,
                todo -> todo.getDescription().toUpperCase())
            .handle(System.out::println)
            .get();
    }
}
```


Listing 11-2 shows a basic example. This example receives a message from the input channel (a `ToDo` instance), filters this message if only the `ToDo` is completed, and then transforms the message by uppercasing the description and handles it by printing it on the console. All of this is called an *integration flow*. But let's take a deeper look inside.

- `IntegrationFlow`. Exposes the DSL as a bean (it is required to have a `@Bean` annotation). This class is a factory for the `IntegrationFlowBuilder` and defines the flow of the integration. It registers all the components, such as message channels, endpoints, and so forth.
- `IntegrationFlows`. This class exposes a fluent API that helps building the integration flow. It's easy to incorporate endpoints such as `transform`, `filter`, `handle`, `split`, `aggregate`, `route`, `bridge`. With these endpoints, you can use any Java 8 (and above) lambda expressions as an argument.
- `from`. This is an overloaded method where you normally pass the message source; in this case, we are calling the input method that returns a `DirectChannel` instance through the `MessageChannels` fluent API.
- `filter`. This overloaded method populates `MessageFilter`. The `MessageFilter` delegates to a `MessageSelector` that sends the message to the filter's output channel if the selector accepts the message.
- `transform`. This method can receive a lambda expression, but actually receives `GenericTransformer<S, T>`, where `S` is the source and the `T` is the type that it is converted to. Here we can use out-of-the-box transformers, like `ObjectToJsonTransformer`, `FileToStringTransformer`, and so forth. In this example, we are the class type (`ToDo`) and a lambda is executed; in this case, getting the `ToDo`'s description and transforming it to uppercase.
- `handle`. This is an overloaded method that populates `ServiceActivatingHandler`. Normally, we can use a POJO that allows you to receive the message and either return a new message or trigger another call. This is a useful endpoint that we are going to see in this chapter and in the next one as a service activator endpoint.

- `@EnableIntegration`. Here we are using a new annotation that sets up all the Spring Integration beans that we need for our flow. This annotation registers different beans, like `errorChannel`, `LoggingHandler`, `taskScheduler`, and more. These beans complement our flow in an integration solution. This annotation is necessary when using Java configurations, annotations, and DSL in a Spring Boot application.

Don't worry too much if this looks different from what you have probably done in the past with integration solutions. You will get more comfortable with all the examples that I show you next, and it gets even easier.

Next, let's create the `ToDoConfig` class in which a `ToDo` is sent through the input channel (see Listing 11-3).

Listing 11-3. `com.apress.todo.config.ToDoConfig.java`

```
package com.apress.todo.config;
import com.apress.todo.domain.ToDo;
import org.springframework.boot.ApplicationRunner;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.messaging.MessageChannel;
import org.springframework.messaging.support.MessageBuilder;

@Configuration
public class ToDoConfig {

    @Bean
    public ApplicationRunner runner(MessageChannel input){
        return args -> {
            input.send(
                MessageBuilder
                    .withPayload(new ToDo("buy milk today",true))
                    .build());
        };
    }
}
```

Listing 11-3 shows the `ApplicationRunner` bean, where it is executed when the application starts (see that the `MessageChannel` is injected—the one declared in the `ToDoIntegration` class). This method is using a `MessageBuilder` class that offers a fluent API that creates messages. In this case, the class is using the `withPayload` method that creates a new `ToDo` instance, marked as completed.

Now it's time to run our application. If you run it, you should see something similar to the following output.

```
...
INFO 39319 - [main] o.s.i.e.EventDrivenConsumer: started simpleFlow.org.
springframework.integration.config.ConsumerEndpointFactoryBean#2
INFO 39319 - [main] c.a.todo.ToDoIntegrationApplication      : Started
ToDoIntegrationApplication in 0.998 seconds (JVM running for 1.422)
GenericMessage [payload=BUY MILK TODAY, headers={id=c245b7a3-3191-641b-
7ad8-1f6eb950f62e, timestamp=1535772540543}]
...
```

Remember that a message is about *headers* and *payload*, which is why we get the `GenericMessage` class with a payload where the final message is “BUY MILK TODAY” and headers that include the ID and the timestamp. This is the result of applying a filter and transforming the message.

Using XML

Next, let's modify the classes to use another type of configuration, an XML, and see how to configure your integration flow. Create the `src/main/resources/META-INF/spring/integration/todo-context.xml` file (see Listing 11-4).

Listing 11-4. `src/main/resources/META-INF/spring/integration/todo-context.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:int="http://www.springframework.org/schema/integration"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd
```

```
http://www.springframework.org/schema/integration http://www.
springframework.org/schema/integration/spring-integration.xsd">
```

```
<int:channel id="input" />
<int:filter input-channel="input"
    expression="payload.isCompleted()"
    output-channel="filter" />
<int:channel id="filter" />
<int:transformer input-channel="filter"
    expression="payload.getDescription().toUpperCase()"
    output-channel="log" />
<int:channel id="log" />
<int:logging-channel-adapter channel="log" />
</beans>
```

Listing 11-4 shows the XML version for configuring the `ToDo` integration flow. I think that it is very straightforward. If you are using the STS IDE, you can use the drag-and-drop panel for Spring Integration flows (integration graph), or generate the graph if you are using IDEA IntelliJ (see Figure 11-3).

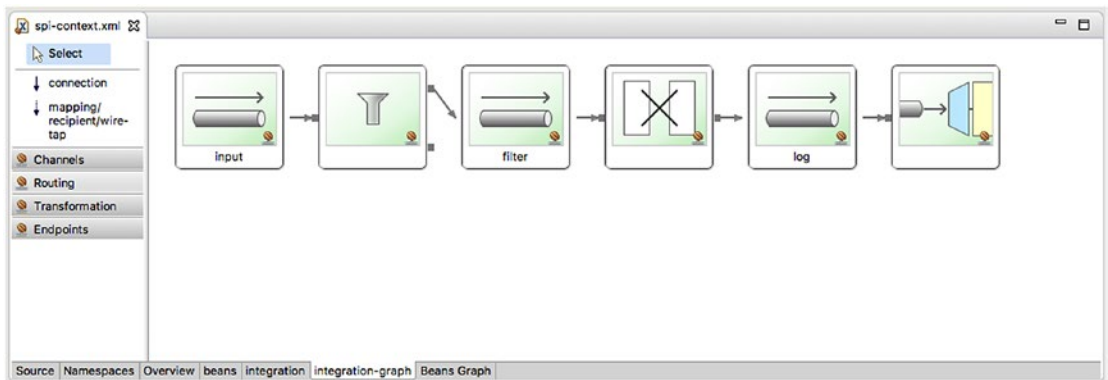


Figure 11-3. Spring integration-graph panel from STS

Figure 11-3 shows the integration-graph panel, where you can create your flows graphically by using the drag-and-drop components. This feature is only for the STS IDE. IDEA IntelliJ generates a graph based on the XML (right-click).

As you can see in Figure 11-X, there are channels, routing, transformations, endpoints, and more. Figure 11-3 is actually the translation of the XML; in other words, you can start by using the XML, and if you switch to the integration graph, it shows you what you have so far, and vice versa. You can use this feature and switch to the source to have the XML. It is a very cool way to create flows, don't you think?

To run this example, it is necessary to comment out all the bean declarations from the `ToDoIntegration` class. Then you need to use the `@ImportResource` annotation to indicate where is the XML you created is located. It should look like the snippet shown in Listing 11-5.

Listing 11-5. `com.apress.todo.integration.ToDoIntregation.java - v2`

```
package com.apress.todo.integration;

import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.ImportResource;
import org.springframework.integration.config.EnableIntegration;

@ImportResource("META-INF/spring/integration/todo-context.xml")
@EnableIntegration
@Configuration
public class ToDoIntegration {

}
```

Listing 11-5 shows the new version of the `ToDoIntegration` class (there is practically no code). We added the `@ImportResource` annotation. This tells Spring Boot that there is a configuration file that needs to be processed. If you run it, you should have the following output.

```
...
INFO 43402 - [main] o.s.i.channel.PublishSubscribeChannel : Channel
'application.errorChannel' has 1 subscriber(s).
2018-09-01 07:23:20.668 INFO 43402 --- [ main] o.s.i.endpoint.
EventDrivenConsumer : started _org.springframework.integration.
errorLogger
INFO 43402 - [main] c.a.todo.ToDoIntegrationApplication : Started
ToDoIntegrationApplication in 1.218 seconds (JVM running for 1.653)
INFO 43402 - [main] o.s.integration.handler.LoggingHandler : BUY MILK TODAY
...
```

Using Annotations

Spring Integration has integration annotations that help you use POJO (Plain Old Java Object) classes, so you can add more business logic to your flow and have a little more control.

Let's modify the `ToDoIntegration` class to look like Listing 11-6.

Listing 11-6. `com.apress.todo.integration.ToDoIntegration.java` – v3

```
package com.apress.todo.integration;

import com.apress.todo.domain.ToDo;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.integration.annotation.Filter;
import org.springframework.integration.annotation.MessageEndpoint;
import org.springframework.integration.annotation.ServiceActivator;
import org.springframework.integration.annotation.Transformer;
import org.springframework.integration.channel.DirectChannel;
import org.springframework.integration.config.EnableIntegration;
import org.springframework.integration.dsl.channel.MessageChannels;
import org.springframework.messaging.MessageChannel;

@EnableIntegration
@Configuration
public class ToDoIntegration {

    @Bean
    public MessageChannel input(){
        return new DirectChannel();
    }

    @Bean
    public MessageChannel toTransform(){
        return new DirectChannel();
    }
}
```

```

@Bean
public MessageChannel toLog(){
    return new DirectChannel();
}

@MessageEndpoint
class SimpleFilter {
    @Filter(inputChannel="input"
            ,outputChannel="toTransform")
    public boolean process(ToDo message){
        return message.isCompleted();
    }
}

@MessageEndpoint
class SimpleTransformer{
    @Transformer(inputChannel="toTransform"
                ,outputChannel="toLog")
    public String process(ToDo message){
        return message.getDescription().toUpperCase();
    }
}

@MessageEndpoint
class SimpleServiceActivator{
    Logger log = LoggerFactory
        .getLogger(SimpleServiceActivator.class);
    @ServiceActivator(inputChannel="toLog")
    public void process(String message){
        log.info(message);
    }
}
}

```

Listing 11-6 shows you the same flow as before, where now we are using integration annotations. Also take a look at the inner classes to facilitate the example. Let's see this code in detail.

- `MessageChannel`. This is an interface that defines methods for sending messages.
- `DirectChannel`. This is a message channel that invokes a single subscriber for each message sent. It is normally used when you don't require any message queue.
- `@MessageEndpoint`. This is a useful annotation that marks a class as the endpoint.
- `@Filter`. This annotation marks a method to do the functionality of a message filter. Normally, you need to return a boolean value.
- `@Transformer`. This annotation marks a method to do the functionality of transforming a message, its header, and/or payload.
- `@ServiceActivator`. This annotation marks a method capable of handling a message.

To run this example, comment out the `@ImportResource` annotation. That's it. You should have logs similar to the following output.

```
...
INFO 43940 - [main] c.a.todo.TODOIntegrationApplication      : Started
TODOIntegrationApplication in 1.002 seconds (JVM running for 1.625)
INFO 43940 - [main] i.TODOIntegration$SimpleServiceActivator : BUY MILK
TODAY
...
```

Using JavaConfig

JavaConfig is very similar to what we just did. What we are going to do next is change the last part of the flow. So, comment out the `SimpleServiceActivator` inner class message endpoint and replace it with the following code.

```
@Bean
@ServiceActivator(inputChannel = "toLog")
public LoggingHandler logging() {
    LoggingHandler adapter = new
        LoggingHandler(LoggingHandler.Level.INFO);
```



```

    adapter.setLoggerName("SIMPLE_LOGGER");
    adapter.setLogExpressionString
("headers.id + ': ' + payload");
    return adapter;
}

```

This code creates a `LoggingHandler` object, which is actually the same object that the XML generates from the `logging-channel-adapter` tag. It logs the `SIMPLE_LOGGER` message with the header’s ID and the payload, in this case, the “BUY MILK TODAY” message.

Again, I know that this is just a trivial example, but at least it gives you an idea of how Spring Integration works and how it can be configured. Clients often ask me if it is possible to mix configurations. Absolutely! We are going to see that very soon.

ToDo with File Integration

Next, let’s see how to integrate file reading. It is a very common task to integrate systems. This is one of the use cases most commonly used. Let’s start by creating a `ToDoProperties` class that helps external properties read the path and the name of the file (see Listing 11-7).

Listing 11-7. `com.apress.todo.config.ToDoProperties.java`

```

package com.apress.todo.config;

import lombok.Data;
import org.springframework.boot.context.properties.ConfigurationProperties;

@Data
@ConfigurationProperties(prefix="todo")
public class ToDoProperties {

    private String directory;
    private String filePattern;

}

```

As you can see in Listing 11-7, there is nothing new. Because this app reads from a file, it is necessary to create a converter that reads a String entry, parses it, and returns a new `ToDo` instance. Create the `ToDoConverter` class (see Listing 11-8).

Listing 11-8. `com.apress.todo.integration.ToDoConverter.java`

```
package com.apress.todo.integration;

import com.apress.todo.domain.ToDo;
import org.springframework.core.convert.converter.Converter;
import org.springframework.stereotype.Component;

import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

@Component
public class ToDoConverter implements Converter<String, ToDo> {
    @Override
    public ToDo convert(String s) {
        List<String> fields = Stream.of(s.split(",")).map(String::trim).
            collect(Collectors.toList());
        return new ToDo(fields.get(0), Boolean.parseBoolean(fields.get(1)));
    }
}
```

There is nothing special in Listing 11-8. The only requirement here is to implement the generic `Converter` interface. I'll talk about it in the next section. Another necessary class is a handler that processes the `ToDo` instance. Create the `ToDoMessageHandler` class (see Listing 11-9).

Listing 11-9. `com.apress.todo.integration.ToDoMessageHandler.java`

```
package com.apress.todo.integration;

import com.apress.todo.domain.ToDo;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Component;
```

```

@Component
public class ToDoMessageHandler {
    private Logger log = LoggerFactory.getLogger(ToDoMessageHandler.class);

    public void process(ToDo todo){
        log.info(">>> {}", todo);
        // More process...
    }
}

```

As Listing 11-9 is a simple POJO class; a method that receives the `ToDo` instance.

Next, let's create the main flow. Create the `ToDoFileIntegration` class (see Listing 11-10).

Listing 11-10. `com.apress.todo.integration.ToDoFileIntegration.java`

```

package com.apress.todo.integration;

import com.apress.todo.config.ToDoProperties;
import org.springframework.boot.context.properties.
EnableConfigurationProperties;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.integration.dsl.IntegrationFlow;
import org.springframework.integration.dsl.IntegrationFlows;
import org.springframework.integration.dsl.Pollers;
import org.springframework.integration.dsl.Transformers;
import org.springframework.integration.file.dsl.Files;
import org.springframework.integration.file.splitter.FileSplitter;

import java.io.File;

@EnableConfigurationProperties(ToDoProperties.class)
@Configuration
public class ToDoFileIntegration {

    private ToDoProperties props;
    private ToDoConverter converter;

```

```

public TodoFileIntegration(TodoProperties props,
TodoConverter converter){
    this.props = props;
    this.converter = converter;
}

@Bean
public IntegrationFlow fileFlow(){
    return IntegrationFlows
        .from(
            Files.inboundAdapter(
                new File(this.props.getDirectory()))
                .preventDuplicates(true)
                .patternFilter(this.props.getFilePattern())
                , e ->
                    e.poller(Pollers.fixedDelay(5000L))
            )
        .split(Files.splitter().markers())
        .filter(
            p -> !(p instanceof FileSplitter.FileMarker))
        .transform(Transformers.converter(converter))

        .handle("todoMessageHandler", "process")
        .get();
}
}

```

Listing 11-10 shows the main integration flow, which reads a file's contents (from the file system), converts the content into an object (in this case, into a `ToDo` object using the `ToDoConverter` class), and handles the message for any extra logic. Let's analyze this in detail.

- `from`. This is an overloaded method where you normally pass the `MessageSource`; in this case, we are passing two values: `Files.inboundAdapter` (that I'll explain next) and a consumer that receives `SourcePollingChannelAdapterSpec`; in this case, we are using a lambda expression to poll in the file system for new files every 5 seconds by using the `Pollers` class.

- **Files.** This is a protocol adapter that works out of the box; you just need to configure it. This adapter is used to pick up files from the file system. The `Files` class belongs to the Spring Integration Java DSL and provides several useful methods:
 - `inboundAdapter`. This adapter brings a fluent API that returns `FileInboundChannelAdapterSpec` that has methods such as
 - `preventDuplicates`. This means that you can avoid reading the same file more than one time by setting this to true.
 - `patternFilter`. This looks for files that name patterns.

In this example, we read from the directory (from the `todo.directory` property value) and the name based on the pattern (from the `todo.file-pattern` property value), both from the `ToDoProperties` class.

- `split`. This method call indicates that the parameter provided (it could be a bean, service, handler, etc.) can split a single message or message payload and produce multiple messages or payloads; in this case, we are using `FileMarker`, which delimits the file data when there is a sequential file process.
- `filter`. Because we are using markers to see each message start and end, we receive the content of the file as a `FileMarker` start, then the actual content and finally the `FileMarker` end, so that's why we are saying here, "pass me the payload or content, not the marker."
- `transform`. Here we are using a `Transformers` class with several implementations for transforming a message, and the converter (a custom converter, the `ToDoConverter` class, see Listing 11-8).
- `handle`. Here we are using a class that handles the message by passing as first parameter the name of the bean (`todoMessageHandler`) and the method that takes care of the process (look at the code in the `ToDoMessageHandler` class, see Listing 11-9) of the message. The `ToDoMessageHandler` class is a POJO marked using the `@Component` annotation.

Note Spring Integration Java DSL supports (just for now) the following protocol adapter classes: AMQP, JMS, Files, SFTP, FTP, HTTP, Kafka, Mail, Scripts, and Feed. These classes are in the `org.springframework.integration.dsl.*` package.

In the `application.properties`, add the following content.

```
todo.directory=/tmp/todo
todo.file-pattern=list.txt
```

Of course, you can add any directory and/or file pattern. The `list.txt` can be anything you want. If you reviewed `ToDoConverter`, it's expecting only two values: the description and the boolean value. So, the `list.txt` file is like this:

```
buy milk today, true
read a book, false
go to the movies, true
workout today, false
buy some bananas, false
```

To run the code, comment out all the code from the `ToDoIntegration` class. Once you run it, you should have something similar to the following output.

```
INFO 47953 - [          main] c.a.todo.TODOIntegrationApplication :
Started TODOIntegrationApplication in 1.06 seconds (JVM running for 1.633)
INFO 47953 - [ask-scheduler-1] c.a.todo.integration.TODOMessageHandler :
>>> ToDo(id=3037a45b-285a-4631-9cfa-f89251e1a634, description=buy milk
today, created=2018-09-01T19:29:38.309, modified=2018-09-01T19:29:38.310,
completed=true)
INFO 47953 - [ask-scheduler-1] c.a.todo.integration.TODOMessageHandler :
>>> ToDo(id=7eb0ae30-294d-49d5-92e2-d05f88a7befd, description=read a
book, created=2018-09-01T19:29:38.320, modified=2018-09-01T19:29:38.320,
completed=false)
INFO 47953 - [ask-scheduler-1] c.a.todo.integration.TODOMessageHandler :
>>> ToDo(id=5380decb-5a6f-4463-b4b6-1567361c37a7, description=go to the
movies, created=2018-09-01T19:29:38.320, modified=2018-09-01T19:29:38.320,
completed=true)
```

```
INFO 47953 - [ask-scheduler-1] c.a.todo.integration.ToDoMessageHandler :
>>> ToDo(id=ac34426f-83fc-40ae-b3a3-0a816689a99a, description=workout
today, created=2018-09-01T19:29:38.320, modified=2018-09-01T19:29:38.320,
completed=false)
INFO 47953 - [ask-scheduler-1] c.a.todo.integration.ToDoMessageHandler :
>>> ToDo(id=4d44b9a8-92a1-41b8-947c-8c872142694c, description=buy some
bananas, created=2018-09-01T19:29:38.320, modified=2018-09-01T19:29:38.320,
completed=false)
```

As you can see, it is a very simple way to integrate with files, read its content, and do any business logic with the data.

Remember that previously I told you that you could mix the way that you can configure Spring Integration? So, what do you need to do if you want to use an actual annotation to handle the message? You can use the `@ServiceActivator` annotation as part of the configuration.

```
@ServiceActivator(inputChannel="input")
public void process(ToDo message){
}
}
```

To use this service activator method, you need to change the flow. Replace this line: `handle("todoMessageHandler", "process")`

With this one:

```
.channel("input")
```

If you re-run the example, you get the same results. Did you realize that there is no input channel defined? The best part is that Spring Integration figures out that you need this channel, and it creates one behind the scenes for you.

It requires a complete book to explain all the Spring Integration goodies; of course, this book is a primer—a basic introduction on the power of integrating multiple systems.

Spring Cloud Stream

So far, you have seen all the messaging techniques that are available, and that using the Spring Framework and Spring Boot makes it easy for developers and architects to create very robust messaging solutions. In this section, we take a new step forward; we enter into cloud-native application development, which is an introduction for the following chapter.

In the next section, I talk about Spring Cloud Stream and how this new technology can help us write message-driven microservices applications.

Spring Cloud

Before I start talking about Spring Cloud Stream internals and usage, let's talk about its umbrella project, Spring Cloud.

Spring Cloud is a set of tools that allows developers to create applications that use all the common patterns in distributed systems: configuration management, service discovery, circuit breakers, smart routing, micro-proxy, control bus, global locks, distributed sessions, service-to-service calls, distributed messaging, and much more.

Based on Spring Cloud, there are several projects, including Spring Cloud Config, Spring Cloud Netflix, Spring Cloud Bus, Spring Cloud for Cloud Foundry, Spring Cloud Cluster, Spring Cloud Stream, and Spring Cloud Stream App Starters.

If you want to start with any of these technologies, you can add the following sections and dependencies to your `pom.xml` file if you are using Maven.

- Add the `<dependencyManagement/>` tag with a GA release; for example

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>Finchley.SR1</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```


- Add the technologies you want to use in the `<dependencies/>` tag; for example,

```
<dependencies>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
  </dependency>

  <!--MORE Technologies here -->

</dependencies>
```

If you are using Gradle, you can add the following content to your `build.gradle` file.

```
ext {
    springCloudVersion = 'Finchley.SR1'
}
dependencyManagement {
    imports {
        mavenBom "org.springframework.cloud:spring-cloud-dependencies:${springCloudVersion}"
    }
}
dependencies {
    // ...
    compile ('org.springframework.cloud:spring-cloud-starter-stream-rabbit')
    // ...
}
```

If you take a deep look into the `pom.xml` file of a Spring Cloud annotation, you see that the naming convention is now `spring-cloud-starter-<technology to use>`. Also note that we are adding a dependency management tag that allows you to deal with transitive dependencies and library version management.

Spring Cloud Stream

It's time to talk about Spring Cloud Stream. Why am I not covering other technologies? What is special about it? Well, Spring Cloud Stream is a lightweight messaging-driven microservices framework based on Spring Integration and Spring Boot (providing the opinionated runtime for easy configuration). You can create enterprise-ready, messaging and integration solution applications with ease. It provides a simple declarative model for sending and receiving messages using either RabbitMQ or Apache Kafka.

I think one of the most important features of Spring Cloud Stream is the decoupling of messaging between producers and consumers by creating bindings that can be used out of the box. In other words, you don't need to add any broker-specific code to your application for producing or consuming messages. Add the required binding (I'll explain this later on) dependencies to your application, and Spring Cloud Stream takes care of the messaging connectivity and communication.

So, let's go ahead and look at the main components of Spring Cloud Stream and learn how to use this framework.

Spring Cloud Stream Concepts

The following are the main components of Spring Cloud Stream.

- *Application model.* The application model is a middleware-neutral core, which means that the application communicates using input and output channels to external brokers (as a way of transporting messages) through binder implementations.
- *Binder abstraction.* Spring Cloud Stream provides at the Kafka and RabbitMQ binder implementations. This abstraction makes it possible for Spring Cloud Stream apps to connect to the middleware. But, how does this abstraction know about the destinations? It can dynamically choose at runtime the destinations based on channels. Normally, we need to provide this through the application. properties file as `spring.cloud.stream.bindings.[input|output].destination` properties. I'll discuss this when we take a look at the examples.

- *Persistent Publish/Subscribe.* The application communication is through the well-known Publish/Subscribe model. If Kafka is used, it follows its own Topic/Subscriber model, and if RabbitMQ is used, it creates a topic exchange and the necessary bindings for each queue. This model reduces any producer and consumer complexity.
- *Consumer groups.* You find out that your consumers probably need to be able to scale up at some point. That’s why scalability is done using the concept of consumer groups (this is similar to the Kafka consumer groups feature), where you can have multiple consumers in a group for a load-balancing scenario, making the scale very easy to set up.
- *Partitioning support.* Spring Cloud Stream supports data partitions, which allows multiple producers to send data to multiple consumers and ensure that common data is processed by the same consumer instances. This is a benefit for performance and consistency of data.
- *Binder API.* Spring Cloud Stream provides an API interface— a Binder SPI (service provider interface) where you can extend the core by modifying the original code, so it’s easy to implement a specific binder like JMS, WebSockets, and so forth.

In this section, we talk more about the programming model and the binders. If you want to learn more about the other concepts, take a look at the Spring Cloud Stream reference. The idea is to show you how to start creating event-driven microservices with Spring Cloud Stream. To show you what we are going to cover, take a look at Figure 11-4.

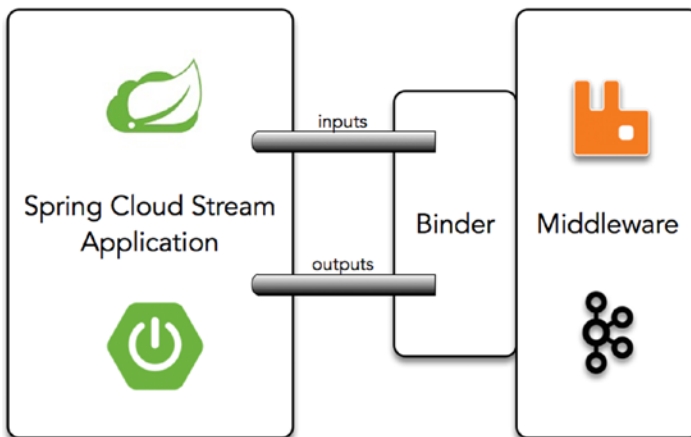


Figure 11-4. *Spring Cloud Stream application*

Spring Cloud Stream Programming

Looking at Figure 11-4, what do you need to create a Spring Cloud Stream app?

- `<dependencyManagement/>`. You need to add this tag with the latest Spring Cloud library dependencies.
- *Binder*. You need to choose which kind of a binder you need.
 - *Kafka*. If you choose Kafka as your binder, then you need to add the following dependency in your `pom.xml` if you are using Maven.

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-stream-kafka</artifactId>
</dependency>
```

If you are using Gradle, then you need to add the following dependency in the `build.gradle` file.

```
compile('org.springframework.cloud:spring-cloud-starter-stream-
kafka')
```

- *RabbitMQ*. If you choose RabbitMQ as your binder, then you need to add the following dependency in your `pom.xml` if you are using Maven.

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
</dependency>
```

If you are using Gradle, then you need to add the following dependency in the `build.gradle` file.

```
compile('org.springframework.cloud:spring-cloud-starter-stream-
rabbit')
```

Have Kafka or RabbitMQ up and running. Can you use both at the same time? Yes, you can. You can configure them in the `application.properties` file.

- `@EnableBinding`. This is a Spring Boot application, so adding the `@EnableBinding` is enough to convert the app to a Spring Cloud Stream.

In the following sections, we send and receive messages from one application to another using RabbitMQ as a transport layer without knowing any specifics about the brokers' API or how to configure to producer or consumer messages.

Spring Cloud Stream uses channels (input/output) as a mechanism to send and receive messages. A Spring Cloud Stream application can have any number of channels, so it defines two annotations, `@Input` and `@Output`, which identify consumers and producers. Normally, a `SubscribableChannel` class is marked with the `@Input` annotation to listen to incoming messages, and the `MessageChannel` class is marked with `@Output` to send messages.

Remember that I told you that Spring Cloud Stream is based on Spring Integration?

If you don't want to deal directly with these channels and annotations, Spring Cloud Stream simplifies things by adding three interfaces that cover the most common messaging use cases: `Source`, `Processor`, and `Sink`. Behind the scenes, these interfaces have the channels (input / output) that your application needs.

- `Source`. A `Source` is used in an application where you are ingesting data from an external system (by listening into a queue, a REST call, file system, database query, etc.) and sending it through an output channel. This is the actual interface from Spring Cloud Stream:

```
public interface Source {
    String OUTPUT = "output";
    @Output(Source.OUTPUT)
    MessageChannel output();
}
```

- `Processor`. You can use a `Processor` in an application when you want to start listening for new incoming messages from the input channel, make processing to the message received (enhancement,

transformation, etc.), and send a new message to the output channel. This is the actual interface in Spring Cloud Stream:

```
public interface Processor extends Source, Sink {
}

```

- Sink. You can use a Sink application when you want to start listening for a new incoming message from the input channel, do processing, and end the flow (saving data, fire a task, log into the console, etc.). This is the actual interface in Spring Cloud Stream:

```
public interface Sink {
    String INPUT = "input";

    @Input(Sink.INPUT)
    SubscribableChannel input();
}

```

Figures 11-5 and 11-6 are the models that we work with.

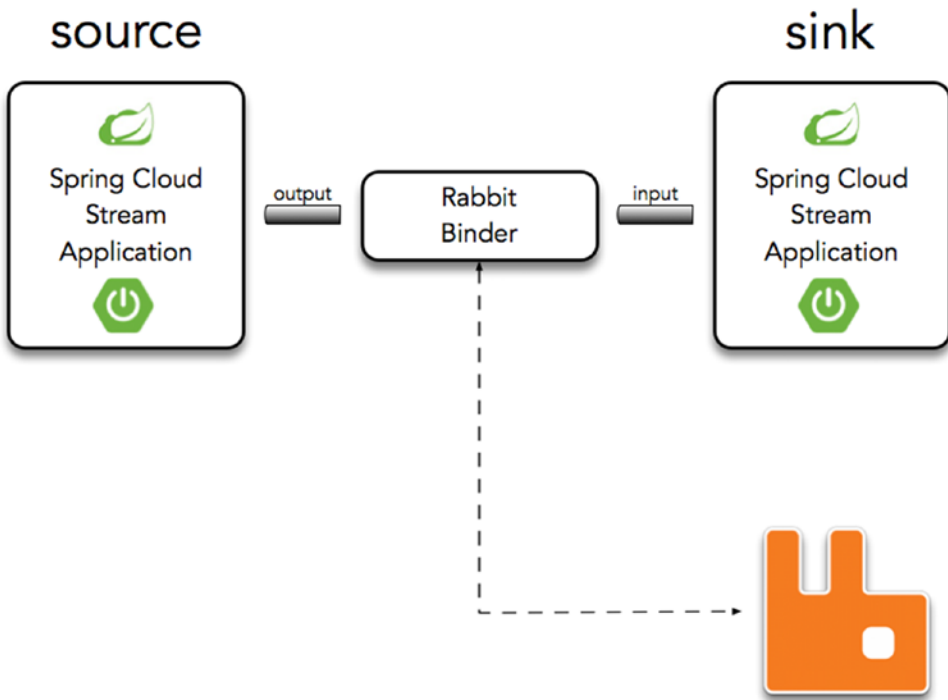


Figure 11-5. Source ➤ Sink

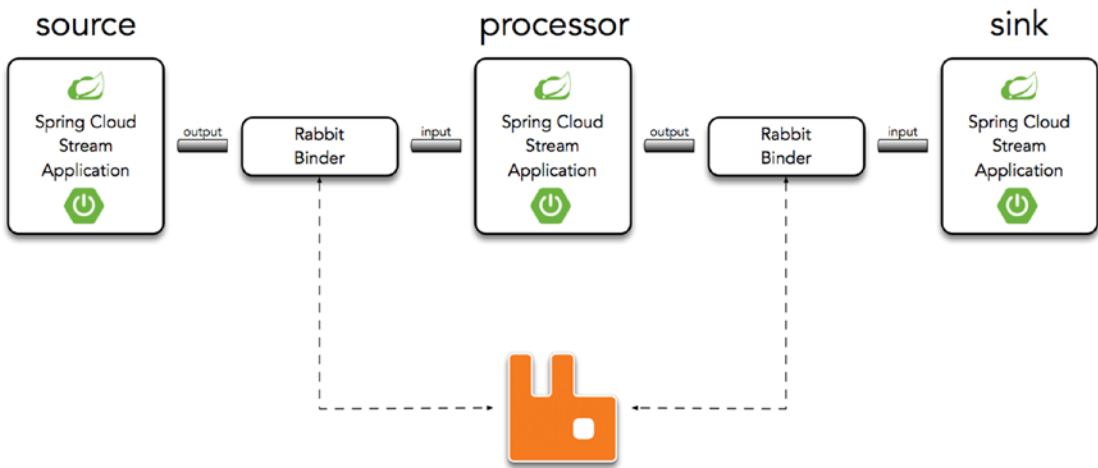


Figure 11-6. *Source > Processor > Sink*

ToDo App with Spring Cloud Stream

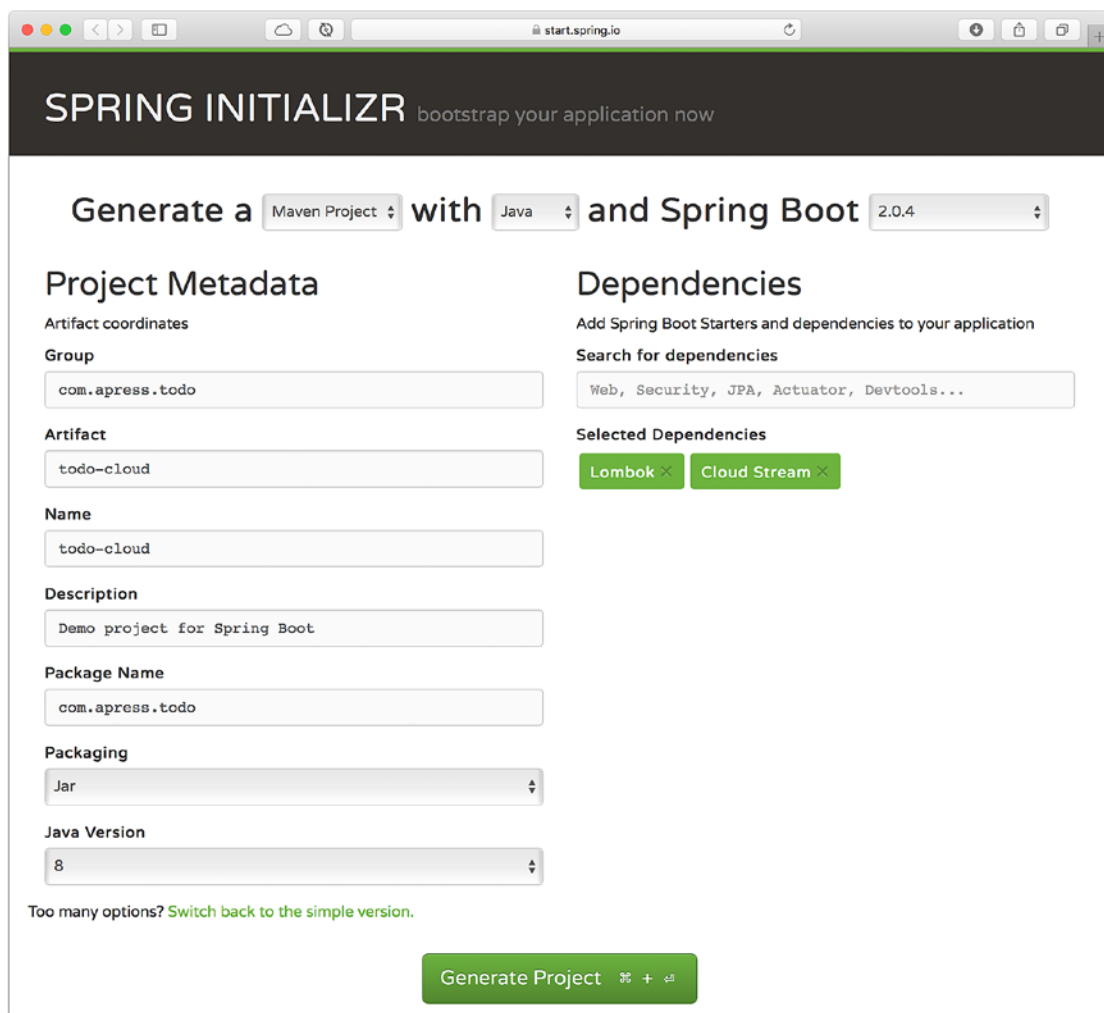
The purpose of this project is to show how you can create a Source interface and send a message through its output channel; a Processor interface and how to receive and send messages from the input and output channels respectively; and a Sink interface and how to receive messages from the input channel. I'm showing what Figure 11-6 describes, but taking each Stream app at a time.

Right now, the communication between these applications is manual, meaning that we need to do some steps in between because I want you to learn how each of the applications work. In the next section, we see how the whole flow works.

You can start from scratch or you can follow along in the next sections to learn what you need to do. If you are starting from scratch, then you can go to Spring Initializr and add the following values to the fields.

- Group: `com.apress.todo`
- Artifact: `todo-cloud`
- Name: `todo-cloud`
- Package Name: `com.apress.todo`
- Dependencies: Cloud Stream, Lombok

You can select either Maven or Gradle as the project type. Then you can press the Generate Project button, which downloads a ZIP file. Uncompress it and import the project in your favorite IDE (see Figure 11-7).



The screenshot shows the Spring Initializr web application in a browser window. The URL is `start.spring.io`. The page has a dark header with the text "SPRING INITIALIZR bootstrap your application now". Below the header, there is a main heading "Generate a" followed by a dropdown menu set to "Maven Project", the word "with", another dropdown menu set to "Java", and the word "and Spring Boot" followed by a dropdown menu set to "2.0.4".

The page is divided into two main sections: "Project Metadata" and "Dependencies".

Project Metadata:

- Artifact coordinates:**
 - Group:** `com.apress.todo`
 - Artifact:** `todo-cloud`
 - Name:** `todo-cloud`
 - Description:** `Demo project for Spring Boot`
 - Package Name:** `com.apress.todo`
 - Packaging:** `Jar`
 - Java Version:** `8`

Dependencies:

- Add Spring Boot Starters and dependencies to your application**
- Search for dependencies:** `Web, Security, JPA, Actuator, Devtools...`
- Selected Dependencies:** `Lombok` and `Cloud Stream`

At the bottom of the form, there is a green button labeled "Generate Project" with a plus sign and a download icon. Below the button, there is a link: "Too many options? [Switch back to the simple version.](#)"

Figure 11-7. Spring Initializr

You can use the `ToDo` domain class from the previous `todo-integration` project (see Listing 11-1).

Source

We are going to start by defining a `Source`. Remember that this component has an output channel. Create the `ToDoSource` class. It should look like Listing 11-11.

Listing 11-11. `com.apress.todo.cloud.ToDoSource.java`

```
package com.apress.todo.cloud;

import com.apress.todo.domain.ToDo;
import org.springframework.cloud.stream.annotation.EnableBinding;
import org.springframework.cloud.stream.messaging.Source;
import org.springframework.context.annotation.Bean;
import org.springframework.integration.annotation.InboundChannelAdapter;
import org.springframework.integration.core.MessageSource;
import org.springframework.messaging.support.MessageBuilder;

@EnableBinding(Source.class)
public class ToDoSource {

    @Bean
    @InboundChannelAdapter(channel=Source.OUTPUT)
    public MessageSource<ToDo> simpleToDo(){
        return () -> MessageBuilder
            .withPayload(new ToDo("Test Spring Cloud Stream"))
            .build();
    }
}
```

Listing 11-11 shows the simplest `Source` stream application that you can have. Let's take a look.

- `@EnableBinding`. This annotation enables this class as a Spring Cloud Stream application, and it enables the necessary configuration for sending or receiving message through the binder provided.

- `Source`. This interface marks the Spring Cloud Stream app as a `Source` stream. It creates the necessary channels; in this case, the output channel to send messages to the binder provided.
- `@InboundChannelAdapter`. This annotation is part of the Spring Integration framework. It polls over the `simpleToDo` method every second, which means that a new message is sent every second. You can actually change the frequency and the number of messages by adding a poller and modifying the default settings; for example,


```
@InboundChannelAdapter(value = Source.OUTPUT, poller =
    @Poller(fixedDelay = "5000", maxMessagesPerPoll = "2"))
```

The important part in this declaration is the channel, where in this case is pointing to `Source.OUTPUT` means that it uses the output channel (`MessageChannel output()`).

- `MessageSource`. This is an interface that sends back `Message<T>`, a wrapper that has payload and headers.
- `MessageBuilder`. You are already familiar with this class, which sends a `MessageSource` type; in this case, we are sending a `ToDo` instance message.

Before you run the example, remember that it is necessary to have the binder dependency, and because we are going to use RabbitMQ, it is necessary to add to your `pom.xml` file if you are using Maven.

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
</dependency>
```

If you are using Gradle, add the following dependency to your `build.gradle` file.

```
compile('org.springframework.cloud:spring-cloud-starter-stream-rabbit')
```

Make sure that you have RabbitMQ up and running. Next, run the example. You probably won't see much, but it is actually doing something. Now, we go into RabbitMQ.

1. Open the RabbitMQ Web Management in a browser. Go to <http://localhost:15672>. The username is guest and the password is guest.
2. Go to the Exchanges tab (see Figure 11-8).

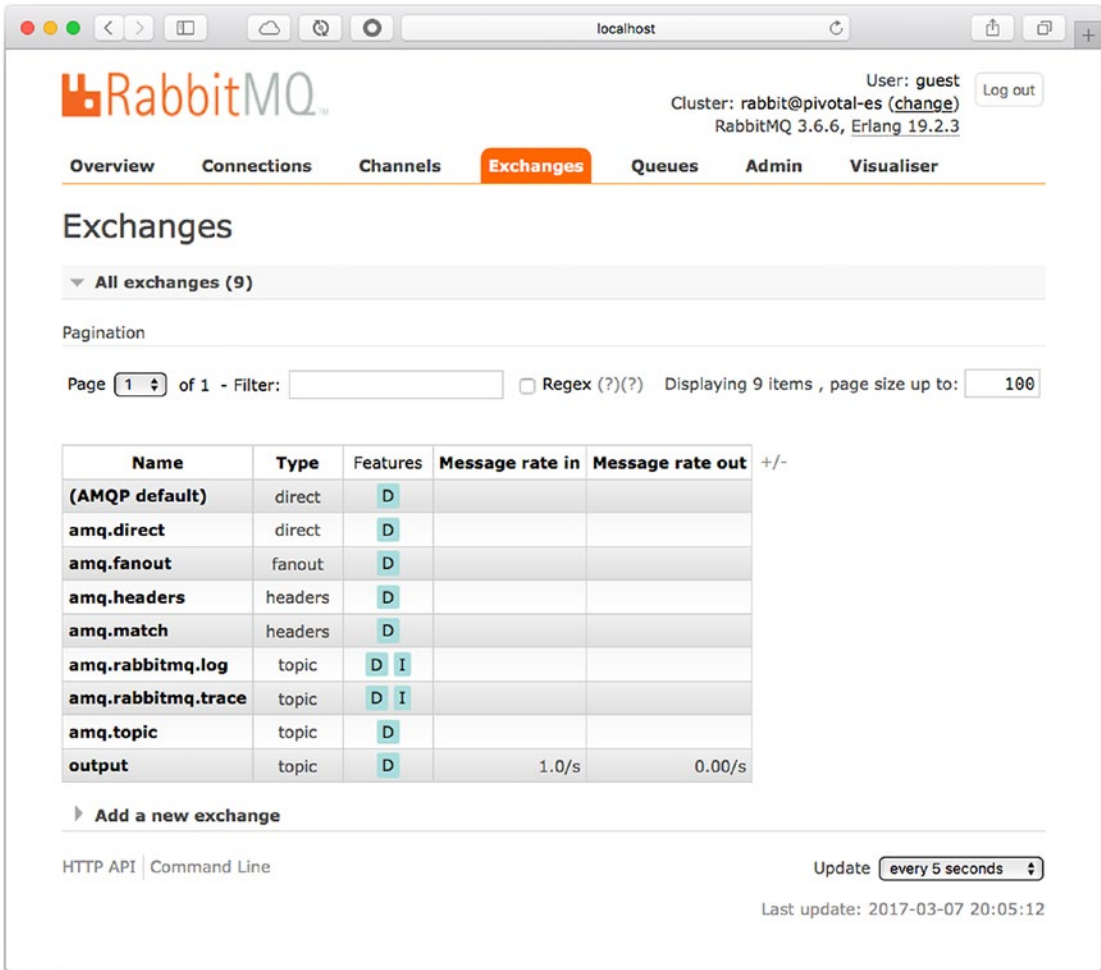


Figure 11-8. RabbitMQ Exchanges tab

Note that an output (a topic Exchange) was created and the message rate is 1.0/s.

- Next, let's bind this exchange to a queue; but first, let's create a queue. Go to the Queues tab and create a new queue named my-queue (see Figure 11-9).

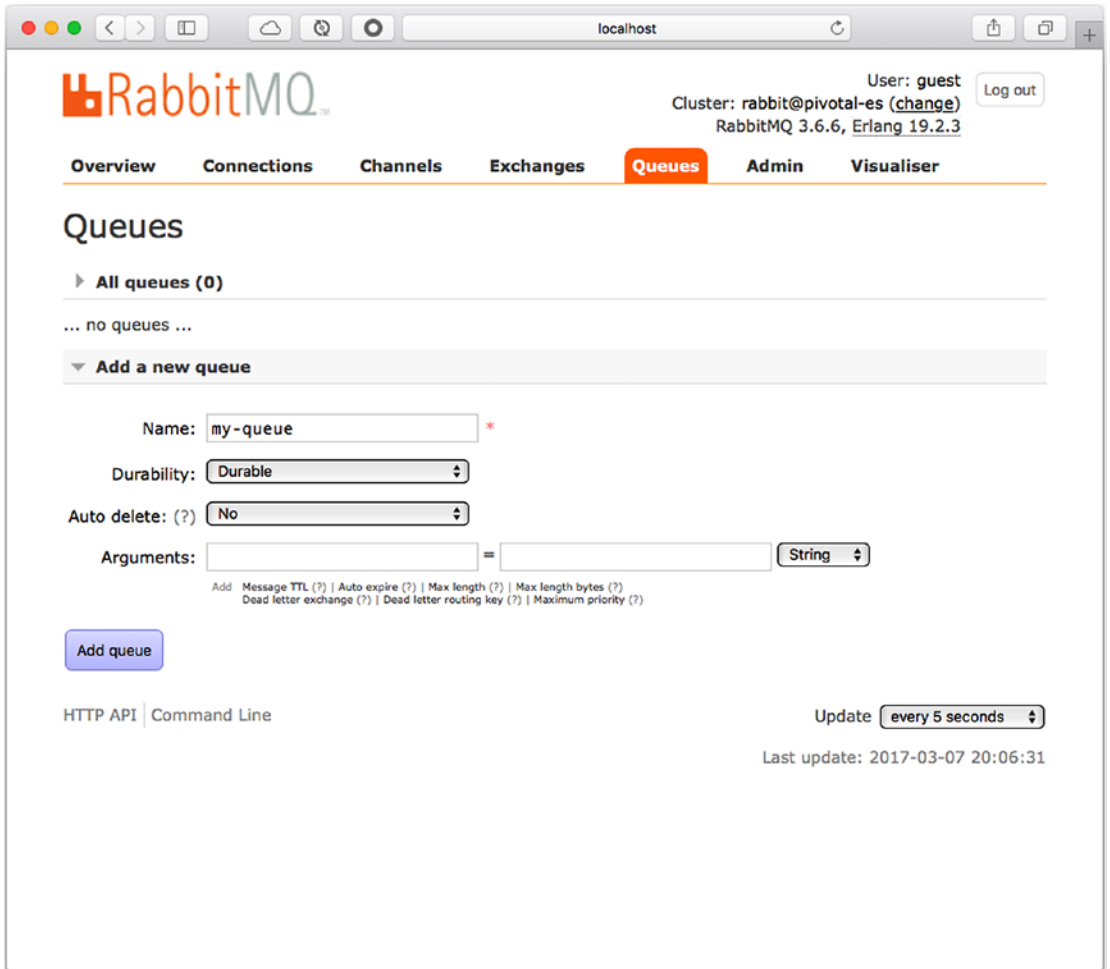


Figure 11-9. Create a Queue: my-queue

- Once the queue is created, it appears in the list. Click my-queue. Go to the Bindings section and add the binding. See Figure 11-10 for the values.

Queue my-queue

▶ Overview

▶ Consumers

▼ Bindings

From	Routing key	Arguments	
(Default exchange binding)			

⇓

This queue

Add binding to this queue

From exchange: *

Routing key:

Arguments: = String ▾

Figure 11-10. Bindings

- Fill out the From Exchange field with the value **output** (this is the name of the exchange). The Routing Key field has the value #, which allows any message to get into my-queue.
- After you bind the output exchange to my-queue, you start seeing several messages. Open the Overview panel (see Figure 11-11).

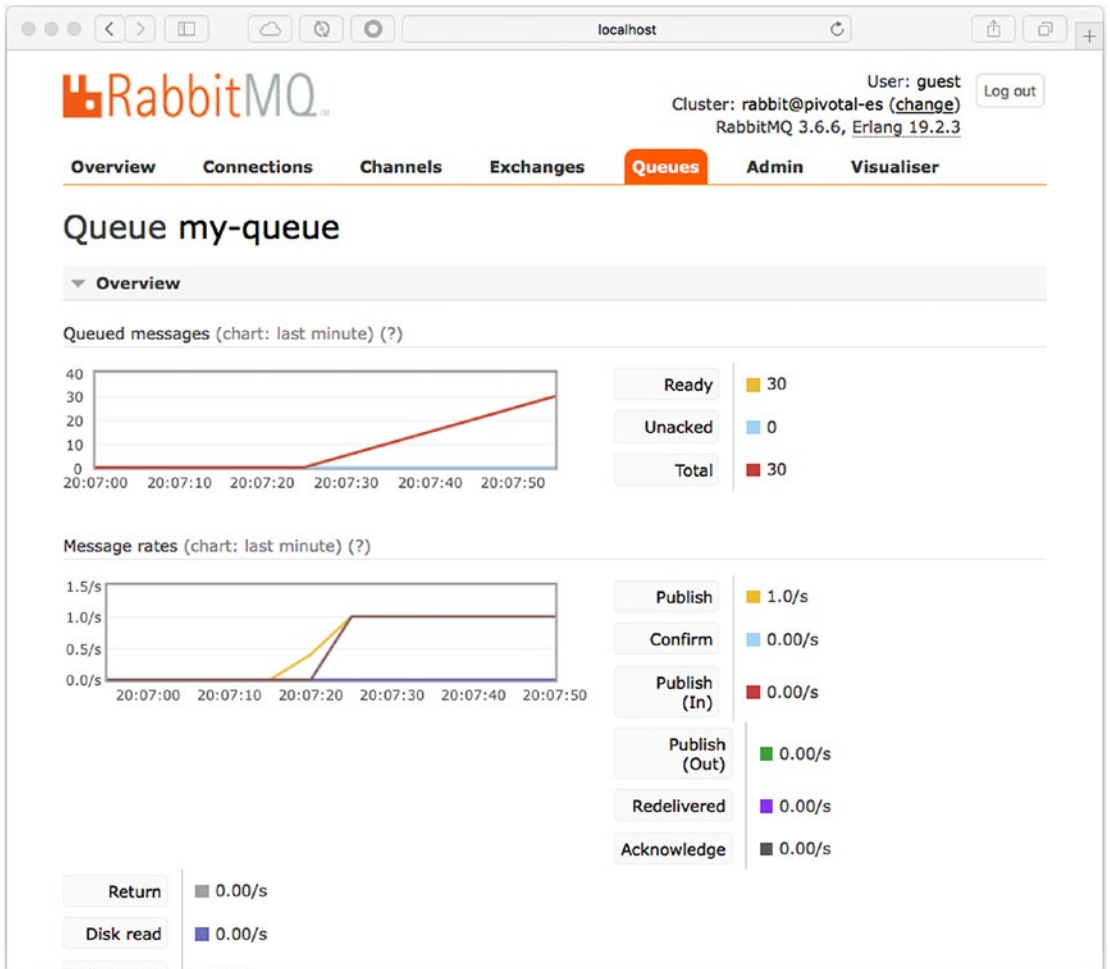


Figure 11-11. Overview

- Let's review a message by opening the Get Messages panel. You can get any number of messages and see its contents (see Figure 11-12).

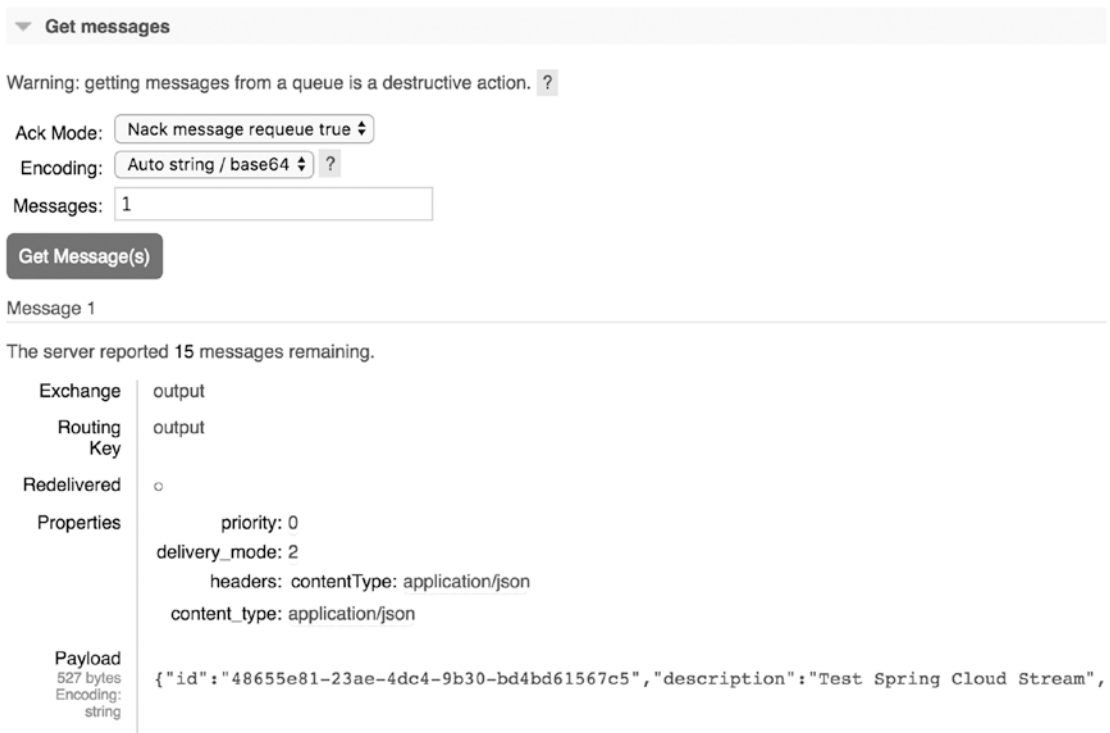


Figure 11-12. Get messages

If you chose several messages, take a look at the payload. You have a message every second. (Note that the format by default is a JSON payload. Also note that the message has properties, such as headers with `contentType: application/json` and `delivery_mode: 2`, which means that the message is being persisted). This is how Spring Cloud Stream and its binder connect to RabbitMQ to publish messages.

If you take a look at the message, you see that the dates are exposed with all the details.

```
{ "id": "68d4100a-e706-4a51-a254-d88545ffe7ef", "description": "Test Spring Cloud Stream", "created": { "year": 2018, "month": "SEPTEMBER", "hour": 21, "minute": 9, "second": 5, "nano": 451000000, "monthValue": 9, "dayOfMonth": 2, "dayOfWeek": "SUNDAY", "dayOfYear": 245, "chronology": { "id": "ISO", "calendarType": "iso8601" } }, "modified": { "year": 2018, "month": "SEPTEMBER", "hour": 21, "minute": 9, "second": 5, "nano": 452000000, "monthValue": 9, "dayOfMonth": 2, "dayOfWeek": "SUNDAY", "dayOfYear": 245, "chronology": { "id": "ISO", "calendarType": "iso8601" } }, "completed": false }
```

You can see a very verbose date serialization, but you can change this by adding the following dependency in the `pom.xml` file if you are using Maven.

```
<dependency>
  <groupId>com.fasterxml.jackson.datatype</groupId>
  <artifactId>jackson-datatype-jsr310</artifactId>
</dependency>
```

If you are using Gradle, add the following dependency to the `build.gradle` file.

```
compile('com.fasterxml.jackson.datatype:jackson-datatype-jsr310')
```

Re-run the app. Now you should see the following messages.

```
{"id":"37be2854-91b7-4007-bf3a-d75c805d3a0a","description":"Test Spring
Cloud Stream","created":"2018-09-02T21:12:12.415","modified":"2018-09-
02T21:12:12.416","completed":false}
```

Processor

This part uses a Listener for the channel input (where all new incoming messages arrive). It gets a `ToDo` message. It converts into an uppercase description, marks the `ToDo` as completed, and then sends it to the output channel.

Create the `ToDoProcessor` class. It should look like [Listing 11-12](#).

Listing 11-12. `com.apress.todo.cloud.ToDoProcessor.java`

```
package com.apress.todo.cloud;

import com.apress.todo.domain.ToDo;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.cloud.stream.annotation.EnableBinding;
import org.springframework.cloud.stream.annotation.StreamListener;
import org.springframework.cloud.stream.messaging.Processor;
import org.springframework.messaging.handler.annotation.SendTo;

import java.time.LocalDateTime;
```


@EnableBinding(Processor.class)

```

public class ToDoProcessor {

    private Logger log = LoggerFactory.getLogger(ToDoProcessor.class);

    @StreamListener(Processor.INPUT)
    @SendTo(Processor.OUTPUT)
    public ToDo transformToUpperCase(ToDo message) {
        log.info("Processing >>> {}", message);
        ToDo result = message;
        result.setDescription(message.getDescription().toUpperCase());
        result.setCompleted(true);
        result.setModified(LocalDateTime.now());
        log.info("Message Processed >>> {}", result);
        return result;
    }
}

```

Listing 11-12 shows a simple Processor stream. Let's review it.

- **@EnableBinding.** This annotation enables this class as a Spring Cloud Stream application. It enables the necessary configuration for sending or receiving messages through the binder provided.
- **Processor.** This interface marks the Spring Cloud Stream app as a Processor stream. It creates the necessary channels; in this case, the input (for listening to new incoming messages) and output channels (for sending messages to the binder provided).
- **@StreamListener.** This annotation is part of the Spring Cloud Stream Framework, very similar to **@RabbitListener** or **@JmsListener**. It listens for new incoming messages in the **Processor.INPUT** channel (**SubscribableChannel input()**).
- **@SendTo.** You already know this annotation; it is the same one used in a previous chapter. Its task is the same; you can see it as a reply or as a producer. It sends a message to the **Processor.OUTPUT** channel (**MessageChannel output()**).

I think this is a trivial but good example of what you can do with a Processor stream. So before you run it, make sure to comment out the `@EnableBinding` annotation from the `ToDoSource` class, and delete the output exchange and the `my-queue` queue.

Run the example. Again, the application is not doing too much, but let's go to RabbitMQ web management.

1. Go to your browser and hit `http://localhost:15672` (username: guest, password: guest).
2. Click the Exchanges tab, and you see the same output exchange and a new input exchange being created. Remember that the Processor stream uses input and output channels (see Figure 11-13).

The screenshot shows the RabbitMQ web management interface. The 'Exchanges' tab is active, displaying a table of all exchanges. The table has columns for Name, Type, Features, Message rate in, and Message rate out. The 'input' exchange is highlighted in blue. Below the table, there is a link to 'Add a new exchange' and a refresh button set to 'every 5 seconds'.

Name	Type	Features	Message rate in	Message rate out	+/-
(AMQP default)	direct	D			
amq.direct	direct	D			
amq.fanout	fanout	D			
amq.headers	headers	D			
amq.match	headers	D			
amq.rabbitmq.log	topic	D I			
amq.rabbitmq.trace	topic	D I			
amq.topic	topic	D			
input	topic	D			
output	topic	D			

Figure 11-13. Exchanges

Note that now there is no message rate in any of the new exchanges.

- Next, go to the Queues tab. You notice a new queue named `input.anonymous.*` has been created (see Figure 11-14).

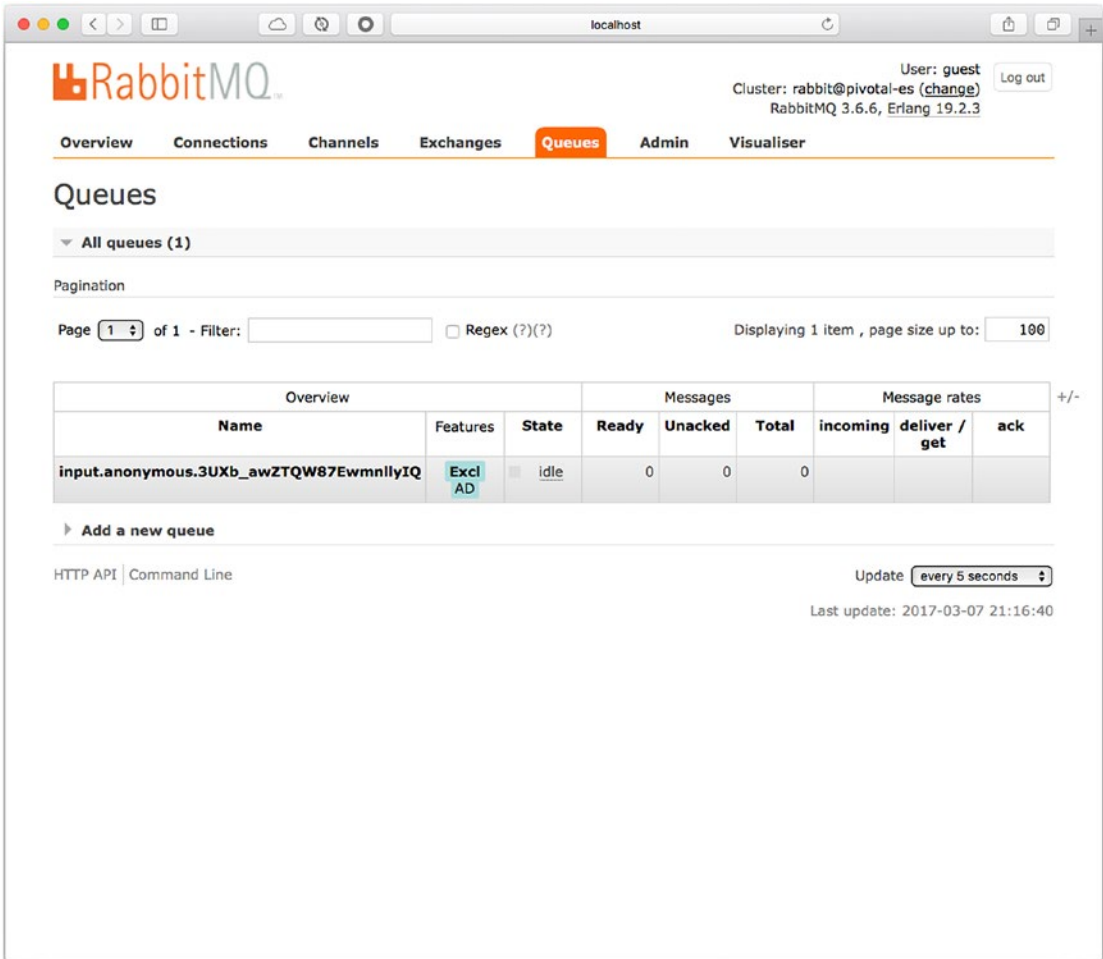


Figure 11-14. Queues

That’s it. The `ToDoProcessor` stream creates the output exchange and the `input.anonymous.*` queue, which means that the stream is connected to the binder, in this case, RabbitMQ. Now the question is how to send a message, right? There are different ways to do it: emulate a message using the RabbitMQ or do it programmatically. We are going to do both.

We are going to create a queue named `my-queue` and bind it to the output, very similar to what we did in the Source stream part.

4. Go to the Queues tab and create a queue named `my-queue` and bind it to the output exchange with a routing key, `#`. This is similar to steps 2 and 3 from the Source stream. Also note that the input `anonymous.*` queue has a binding to the input exchange.
5. Now, we are going to send a message using the input exchange. Go to the Exchanges tab. Click the input exchange and select the Publish Message panel.
6. Enter the following in the Payload field.

```
{ "id": "37be2854-91b7-4007-bf3a-d75c805d3a0a", "description":
  "Test Spring Cloud Stream", "created": "2018-09-02T21:12:12.415",
  "modified": "2018-09-02T21:12:12.416", "completed": false }
```

Enter **content-type=application/json** in the Properties field (see Figure 11-15).

The screenshot shows the 'Publish message' interface. It has a 'Routing key' field, a 'Delivery mode' dropdown set to '1 - Non-persistent', and a 'Headers' section with a dropdown set to 'String'. The 'Properties' section has a key-value pair: 'content_type' = 'application/json'. The 'Payload' field contains the following JSON:

```
{ "id": "37be2854-91b7-4007-bf3a-d75c805d3a0a", "description": "Test Spring Cloud Stream", "created": "2018-09-02T21:12:12.415", "modified": "2018-09-02T21:12:12.416", "completed": false }
```

A 'Publish message' button is located at the bottom left of the form.

Figure 11-15. Publish message

Then click the Publish Message button. It should appear as a message saying “Message published.”

- 7. Next, take a look at the app’s logs. You should have something similar to the following output.

```
...  
Processing >>> ToDo(id=37be2854-91b7-4007-bf3a-  
d75c805d3a0a, description=Test Spring Cloud Stream,  
created=2018-09-02T21:12:12.415, modified=2018-09-  
02T21:12:12.416, completed=false)  
Message Processed >>> ToDo(id=37be2854-91b7-4007-bf3a-  
d75c805d3a0a, description=TEST SPRING CLOUD STREAM,  
created=2018-09-02T21:12:12.415, modified=2018-09-  
02T21:54:55.048, completed=true)  
...
```

And if you take a look at the my-queue queue, and get the message, you should see practically the same results (see Figure 11-16).



Figure 11-16. Get messages

This is simple but it is not the right way. You are never going to send a message using the RabbitMQ console, except maybe for a small test.

I mentioned that we are able to send messages programmatically. Create the `ToDoSender` class (see Listing 11-13).

Listing 11-13. `com.apress.todo.sender.ToDoSender.java`

```
package com.apress.todo.sender;

import com.apress.todo.domain.ToDo;
import org.springframework.boot.ApplicationRunner;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.messaging.MessageChannel;
import org.springframework.messaging.support.MessageBuilder;

@Configuration
public class ToDoSender {

    @Bean
    public ApplicationRunner send(MessageChannel input){
        return args -> {
            input
                .send(MessageBuilder
                    .withPayload(new ToDo("Read a Book"))
                    .build());
        };
    }
}
```

If you run the application, now you have a `ToDo` with the description in uppercase and set as completed in the logs and in the `my-queue` queue. As you can see, we are using a class that you know from Spring Integration and using the `MessageChannel` interface. What is interesting is that Spring knows which channel to inject. Remember that the `@Processor` annotation exposes the input channel.

Sink

The Sink stream creates an input channel to listen for new incoming messages. Let's create the `ToDoSink` class (see Listing 11-14).

Listing 11-14. `com.apress.todo.cloud.ToDoSink.java`

```
package com.apress.todo.cloud;

import com.apress.todo.domain.ToDo;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.cloud.stream.annotation.EnableBinding;
import org.springframework.cloud.stream.annotation.StreamListener;
import org.springframework.cloud.stream.messaging.Sink;

@EnableBinding(Sink.class)
public class ToDoSink {

    private Logger log = LoggerFactory.getLogger(ToDoSink.class);

    @StreamListener(Sink.INPUT)
    public void process(ToDo message){
        log.info("SINK - Message Received >>> {}",message);
    }
}
```

Listing 11-14 shows a Sink stream, and you already know the annotations. The `@EnableBinding` converts this class into a Source stream, and it listens for new incoming messages through the `@StreamListener` and the `Sink.INPUT` channel. `Sink.INPUT` creates an input channel (`SubscribableChannel input()`).

If you use Listing 11-13 to comment out the `@EnableBinding` from the `ToDoProcessor` class and run the application, take a look at the RabbitMQ management, you see the input exchange and the `input.anonymous.*` created and bound to each other. You should get the same `ToDo` with the Sink logs.

Remember, the Sink stream does extra work with the message received but it ends the flow.

What I've explained so far doesn't do too much in the sense that are kind of probe of concept, and actually that was my intention because I want you to understand how does work internally. Now, let's use a real-life scenario, where we actually create a complete flow and see how these Streams can communicate with each other without going into the RabbitMQ management.

Microservices

I want to talk about this new way to create scalable and highly available applications using microservices. The most important part of this section is the ability to communicate between streams using messaging. At the end, you should consider and see each stream (source, processor, and sink) as a microservice.

ToDo App: A Complete Flow

Let's list some of the requirements for this new ToDo app.

- Create a Source that reads any ToDo's declared from a file and filter the ones that are completed returning a ToDo instance.
- Create a Processor that accepts a ToDo and create a text message.
- Create a Sink that receives the text and sends an email to a recipient.

Do you think can you do it? See Figure 11-17.

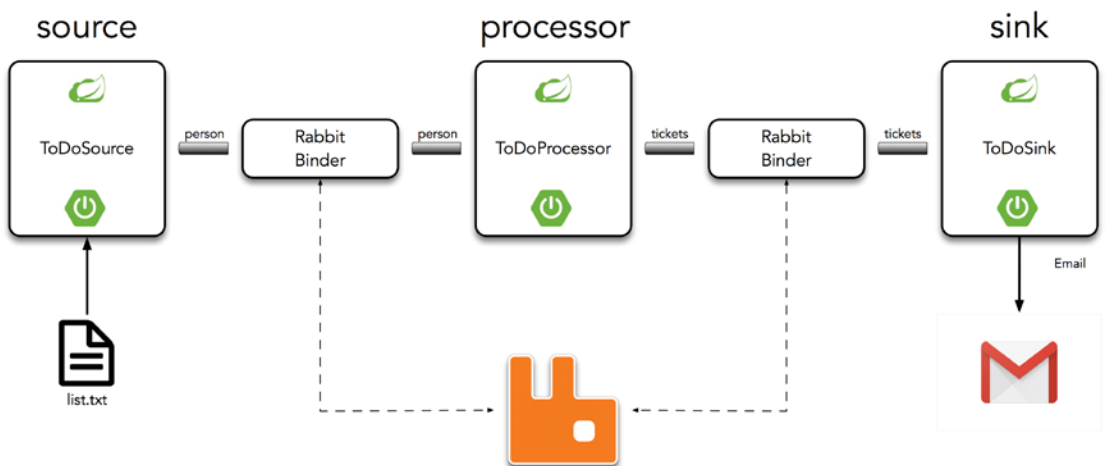


Figure 11-17. ToDo Flow

Figure 11-17 shows the real flow (note that each part is an independent app). In other words, you create `todo-source`, `todo-processor` and `todo-sink`.

Take a look at the source code for Chapter 11 to find every project. This is your homework. Make them work. Change the properties according to your settings, in this case, in the `todo-sink` project.

Spring Cloud Stream App Starters

What if I told you that we could avoid creating the previous example and use the Spring Cloud Stream app starters?

Spring Cloud Stream provides out-of-the-box applications starters that run. The Spring Cloud team already implemented about 52 applications that you can download, configure, and execute. These application starters are divided by Source, Processor, and Sink models.

- Source: `file`, `ftp`, `gemfire`, `gemfire-cq`, `http`, `jdbc`, `jms`, `load-generator`, `loggregator`, `mail`, `mongodb`, `rabbit`, `s3`, `sftp`, `syslog`, `tcp`, `tcp-client`, `time`, `trigger`, `triggertask`, `twitterstream`
- Processor: `bridge`, `filter`, `groovy-filter`, `groovy-transform`, `httpclient`, `pmml`, `scriptable-transform`, `splitter`, `tcp-client`, `transform`, and more
- Sink: `aggregate-counter`, `cassandra`, `counter`, `field-value-counter`, `file`, `ftp`, `gemfire`, `gpfdist`, `hdfs`, `hdfs-dataset`, `jdbc`, `log`, `rabbit`, `redis-pubsub`, `router`, `s3`, `sftp`, `task-launcher-local`, `task-launcher-yarn`, `tcp`, `throughput`, `websocket`, and many more

Note If you need to get the latest release of the app starters, you can get them from <http://repo.spring.io/libs-release/org/springframework/cloud/stream/app/>.

If you want to use the other Spring Cloud Stream application starters and see their configuration, take a look at <http://docs.spring.io/spring-cloud-stream-app-starters/docs/current/reference/html/> for reference.

Summary

In this chapter, you learned how to use Spring Integration and Spring Cloud Stream with Spring Boot.

You learned how Spring Integration helps you create robust and scalable applications that can be integrated with other systems.

You learned how Spring Cloud Stream gives the facility to create microservices with ease. And you learned how to use this framework and any transport method you want. It is an agnostic transport protocol framework, hiding all the messaging details; in other words, you don't need to learn RabbitMQ or Kafka to use this framework.

In the next chapter, you see how Spring Boot can live in the cloud.

CHAPTER 12

Spring Boot in the Cloud

Cloud computing is one of the most important concepts in the IT industry. Companies that want to be on the cutting-edge of the latest technologies are looking to be fast by increasing the speed of their services. They want to be safe by recovering from errors or mistakes as fast as possible without the client knowing about it. They want to be scalable by growing horizontally (typically refers to scaling infrastructure capacity outward, such as spawning more servers to share the load) instead of vertically (refers to the ability to increase available resources (CPU, memory, disk space, etc.) for an existing entity like a server). But what kind of technology can provide all of these concepts?

The term *cloud-native architecture* is beginning to emerge. It allows developers to follow patterns that provide speed, safety, and scalability with ease. In this chapter, I show you how you can create and deploy Spring Boot applications for the cloud by following some of these patterns.

The Cloud and Cloud-Native Architecture

I imagine you have heard about these companies: Pivotal, Amazon, Google, Heroku, Netflix, and Uber. They are applying all the concepts I mentioned. But how do these companies accomplish being fast, safe, and scalable at the same time?

One of the first pioneers of cloud computing was Amazon, which started using virtualization as primary tool to create resource elasticity; this means that any deployed application can have more computer power by increasing the number of virtual boxes, memory, processors, and so forth, without any IT person involved. All of these new ways to scale an application was the result of satisfying growing user demand.

How can Netflix satisfy all of their user demands? We are talking about millions of users who are streaming media content daily.

All of these companies have the IT infrastructure required for the cloud era, but don't you think that any application that wants to be part of the cloud needs to be adaptable to this new technology? You need to start thinking about how scaling resources impact

an application. You need to start thinking more about distributed systems, right? How applications communicate with legacy systems or between each other in these kinds of environments. What happened if one of your systems is down? How do you recover? How do users (and if millions) take advantage of the cloud?

The new cloud-native architecture responds to all of these questions. Remember that your applications need to be fast, safe, and scalable.

First, you need to have visibility in this new cloud environment, meaning that you need to have a better way to monitor your applications—set alerts, have dashboards, and so forth. You need fault isolation and tolerance, which means applications that are context-bounded, and that the applications shouldn't have any dependencies between each other. If one of your applications is down, the other apps should keep running. If you are continuously deploying an application, it shouldn't affect the entire system. This means that you need to think about auto-recovery, where the entire system is capable of identifying the failure and recover.

Twelve-Factor Applications

The engineers at Heroku identified a lot of patterns that became the twelve-factor application guide (<https://12factor.net>). This guide shows how an application (a single unit) needs to focus on declarative configuration, being stateless and deployment independent. Your application needs to be fast, safe, and scalable.

The following is a summary of the twelve-factor application guide.

- *Codebase.* One codebase tracked in VCS/many deploys. One app has a single code base that is tracked by a version control system (VCS) like Git, Subversion, Mercurial, and so forth. You can do many deployments (from the same code base) for development, testing, staging, and production environments.
- *Dependencies.* Explicitly declare and isolate dependencies. Sometimes environments don't have an Internet connection (if is a private system), so you need to think about packaging your dependencies (JARs, gems, shared libraries, etc.). If you have an internal repository of libraries, you can declare a manifest like poms, gemfile, bundles, and so forth. Never rely on everything in your final environment.

- *Configuration.* Store config in the environment. You shouldn't hardcode anything that varies. Use the environment variables or a configuration server.
- *Backing services.* Treat backing services as attached resources. Connect to services via URL or configuration.
- *Build, release, run.* Strictly separate build and run stages. Related to a CI/CD (continuous integration, continuous delivery).
- *Processes.* Execute the app as one or more stateless processes. Processes should not store internal states. Share nothing. Any necessary state should be considered a backing service.
- *Port binding.* Export services via port binding. Your application is self-contained, and these apps are exposed via port binding. An application can become another app's service.
- *Concurrency.* Scale out via the process model. Scale by adding more application instances. Individual processes are free to multithread.
- *Disposability.* Maximize robustness with fast startup and graceful shutdown. Processes should be disposable (remember, they are stateless). Fault tolerant.
- *Environment parity.* Keep development, staging, and production environments as similar as possible. This is a result of high quality and ensures continuous delivery.
- *Logs.* Treat logs as event streams. Your apps should write to *stdout*. Logs are streams of aggregated, time-ordered events.
- *Admin processes.* Run admin and management tasks as one-off processes. Run admin processes on the platform: DB migrations, one time scripts, and so forth.

Microservices

The term *microservices* is a new way to create applications. You need to see microservices as a way to decompose monolithic applications into different and independent components that follow the twelve-factor app guide. When deployed, they work (see Figure 12-1).

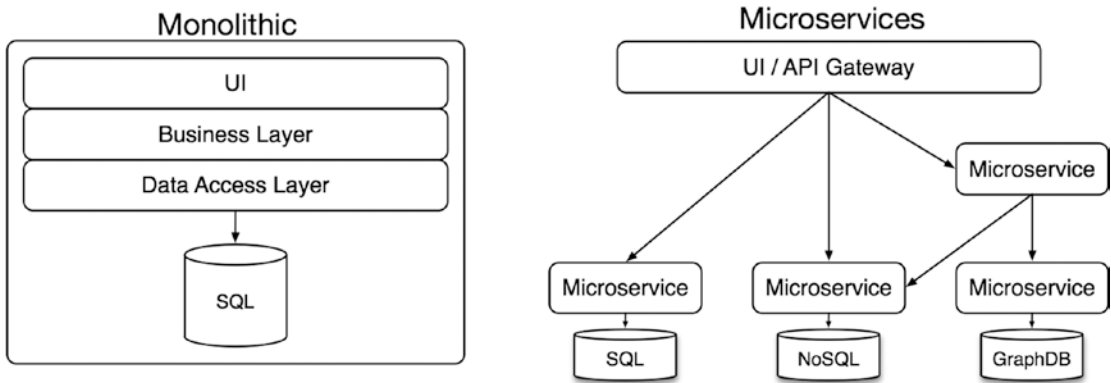


Figure 12-1. *Monolithic vs. microservices*

I think microservices have around since the invention of UNIX, because you can use one of the command-line tools, for example, `grep`, which is a single unit that does its job well. And if you combine several of these commands (e.g., `find . -name microservices.txt | grep -i spring-boot`), you can create a better app or system. But these commands are independent of each other and communicate through the UNIX pipe (`|`). This analogy can be the same within your applications.

Microservices help you accelerate development. Why? Because you can designate a small team that works on only one feature of the application with a bounded-context that follows the twelve-factor application guidelines.

There is a lot to say about microservices and the guides on how to migrate existing architectures into microservices, but the idea here is to explore Spring Boot and learn how to deploy it into a cloud environment.

Preparing the ToDo App as a Microservice

What would you need to do to convert the Spring Boot ToDo app to a microservice? Actually, nothing! Yes, nothing, because Spring Boot is a way to create microservices with ease. So, you are going to use the same ToDo app to deploy to a cloud platform. Which platform? The Pivotal Cloud Foundry!

You can choose the *todo-rest* project from previous chapters. Review it if you modify it, and make sure that you can run it.

It is important to make sure that you have these dependencies; if you are using Maven, you should have the following dependencies in your `pom.xml` file.

```
...
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
  <scope>runtime</scope>
</dependency>
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <scope>runtime</scope>
</dependency>
...
```

If you are using Gradle, see if you have these dependencies in your `build.gradle` file.

```
...
runtime('com.h2database:h2')
runtime('mysql:mysql-connector-java')
...
```

Why are these dependencies important? You learn that in the next sections. Next, go to your `application.properties` file and make sure that it is like the following content.

```
# JPA
spring.jpa.generate-ddl=true
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

What changed here was the `ddl-auto` property; before you used `create-drop`, which creates and destroys the schema at the end of the session. And you are changing this property to `update`, which means that it updates the schema if necessary. There is a point here, but see it in action in the next sections.

Let's prepare the app by executing the following command, where the source code is. (You can also execute a Maven goal or Gradle task inside your IDE; take a look at the documentation for how to do it.) If you are using Maven, you can execute

```
$ ./mvnw clean package
```

If you are using Gradle, you can execute

```
$ ./gradlew clean build
```

These commands generate the JAR that is deploying very soon. So, keep it safe; we are going back to it.

Note If you have issues with a Java constructor for the `ToDo` domain class, you are using an old version of Lombok (because in the domain class is the `@NoArgsConstructor` annotation). The Spring Boot team hasn't updated yet this library, so use Lombok version 1.18.2 or higher.

Pivotal Cloud Foundry

Cloud Foundry has been around since 2008; it began as an open source project by VMWare, then it was moved to Pivotal in 2013. Since then, Cloud Foundry has been the most used open source PaaS. Cloud Foundry, as an open source solution, has the largest community support. It's backed up by several large IT companies, including IBM (with Bluemix), Microsoft, Intel, SAP, and of course, Pivotal (with Pivotal Cloud Foundry—PAS and PKS) and VMware.

Cloud Foundry is the only open source solution that you can actually download and run without any problems. You can find two versions of Cloud Foundry: open source at www.cloudfoundry.org and the Pivotal Cloud Foundry PAS and PKS (commercial version) at <http://pivotal.io/platform>. If you are interested in downloading the commercial version, you can actually do it without any trials or limited time at

<https://network.pivotal.io/products/pivotal-cf>. Actually, this is a free version, but if you want to have support or help on how to install it, that's when you need to contact a Pivotal sales representative.

At the beginning of 2018, Pivotal released version 2.0 of Platform, which has more options for the end user. It brings the Pivotal Application Service (PAS) and the Pivotal Container Service (PKS, based on Kubernetes) solutions to the market (see Figure 12-2).

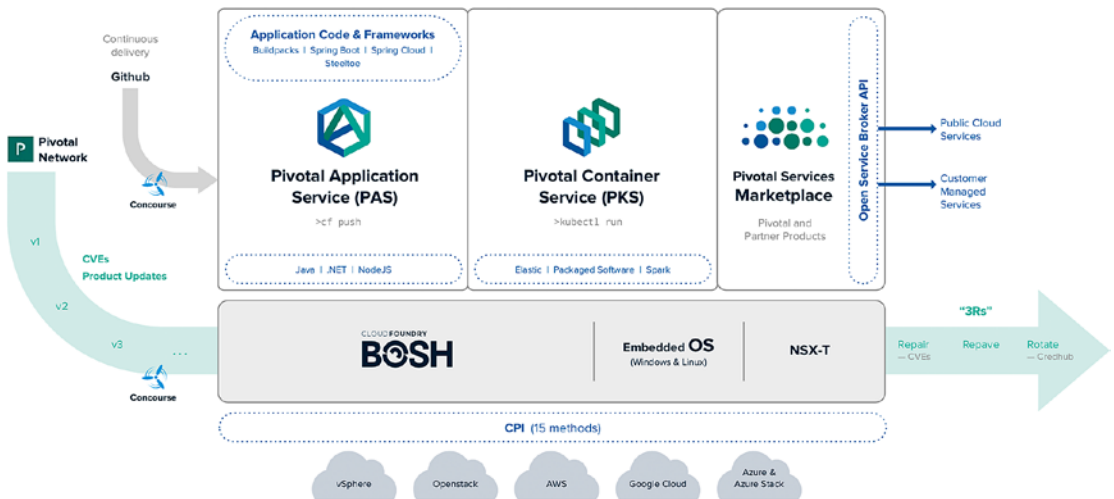


Figure 12-2. Pivotal Cloud Foundry 2.x

In the following sections, I cover only PAS and easy way to start with a cloud-native development, because you only need to care about your app, the data, and nothing else!

PAS: Pivotal Application Service

Pivotal Application Service (PAS) is built on open architecture, and it offers the following features.

- *Router.* Routes incoming traffic to the appropriate component, usually the cloud controller or a running application on a DEA node.
- *Authentication.* The OAuth2 server and login server work together to provide identity management.

- *Cloud controller.* The cloud controller is responsible for managing the life cycle of application.
- *Monitoring.* Monitors, determines, and reconciles applications to determine their state, version, and number of instances, and redirects to the cloud controller to take action to correct any discrepancies.
- *Garden/Diego Cells.* Manages application instances, tracks started instances, and broadcasts state messages.
- *Blob store.* Resources, application code, build packs, and droplets.
- *Service brokers.* When a developer provisions and binds a service to an application, the service broker is responsible for providing the service instance.
- *Message bus.* Cloud Foundry uses NATS (different from the network nats), a lightweight publish-subscribe and distributed queueing messaging system, for internal communication between components.
- *Logging and statistics.* The metrics collector gathers metrics from the components. Operators can use this information to monitor an instance of Cloud Foundry.

PAS Features

PAS, powered by Cloud Foundry (open source), delivers a turnkey PaaS experience on multiple infrastructures with leading application and data services.

- Commercially supported release based on Cloud Foundry open source.
- Fully automated deployment, updates, and one-click horizontal and vertical scaling on vSphere, vCloud Air, AWS, Microsoft Azure, Google Cloud, or OpenStack with minimal production downtime.
- Instant horizontal application-tier scaling.

- Web console for resource management and administration of applications and services.
- Applications benefit from built-in services, like load balancing and DNS, automated health management, logging, and auditing.
- Java Spring support through provided Java buildpack.
- Optimized developer experience for the Spring Framework.
- MySQL service for rapid development and testing.
- Automatic application binding and service provisioning for Pivotal services, such as Pivotal RabbitMQ, Pivotal Redis, Pivotal Cloud Cache (based on GemFire), and MySQL for Pivotal Cloud Foundry.

What is the difference between the open source version and the commercial version? Well, all the features listed. In the open source version, you need to do everything manually using the command line mostly (to install, configure, upgrade, etc.), but in the commercial version, you can use a web console to manage your infrastructure and run your applications. It's important to know that you can install Cloud Foundry on Amazon AWS, OpenStack, Google Cloud, Microsoft Azure, and vSphere. Pivotal Cloud Foundry (PAS and PKS) is IaaS agnostic!

Using PWS/PAS

To use PWS/PAS, you need to open an account in Pivotal Web Services at <https://run.pivotal.io>. You can get a trial account (see Figure 12-3).

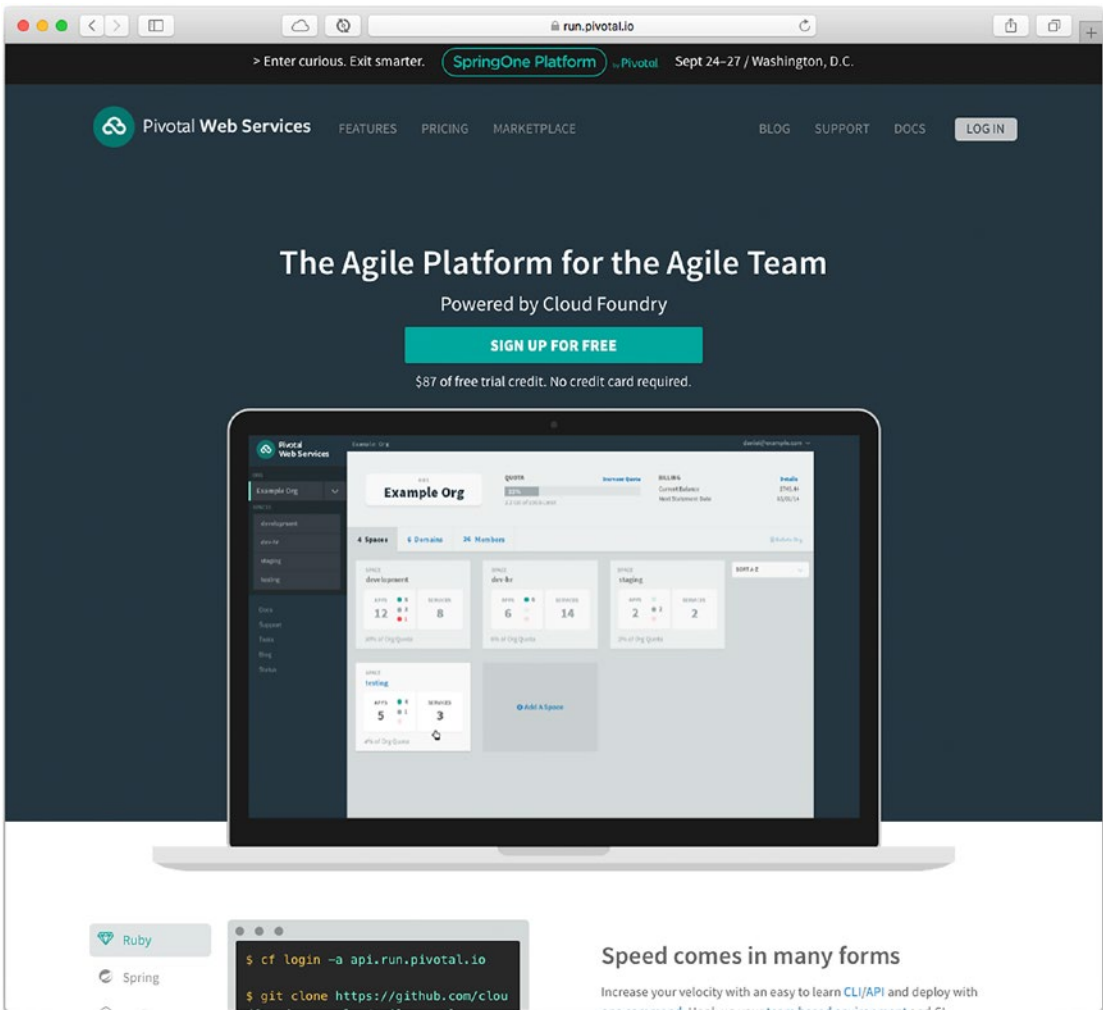


Figure 12-3. Pivotal Web Services - <https://run.pivotal.io>

When you sign up, you are prompted for a phone number, where you receive a code to start your trial. It also asks you for an organization, which could be your name with a -org; mine is fg-org, for example. By default, it creates the space (named development) where you will work (see Figure 12-4).

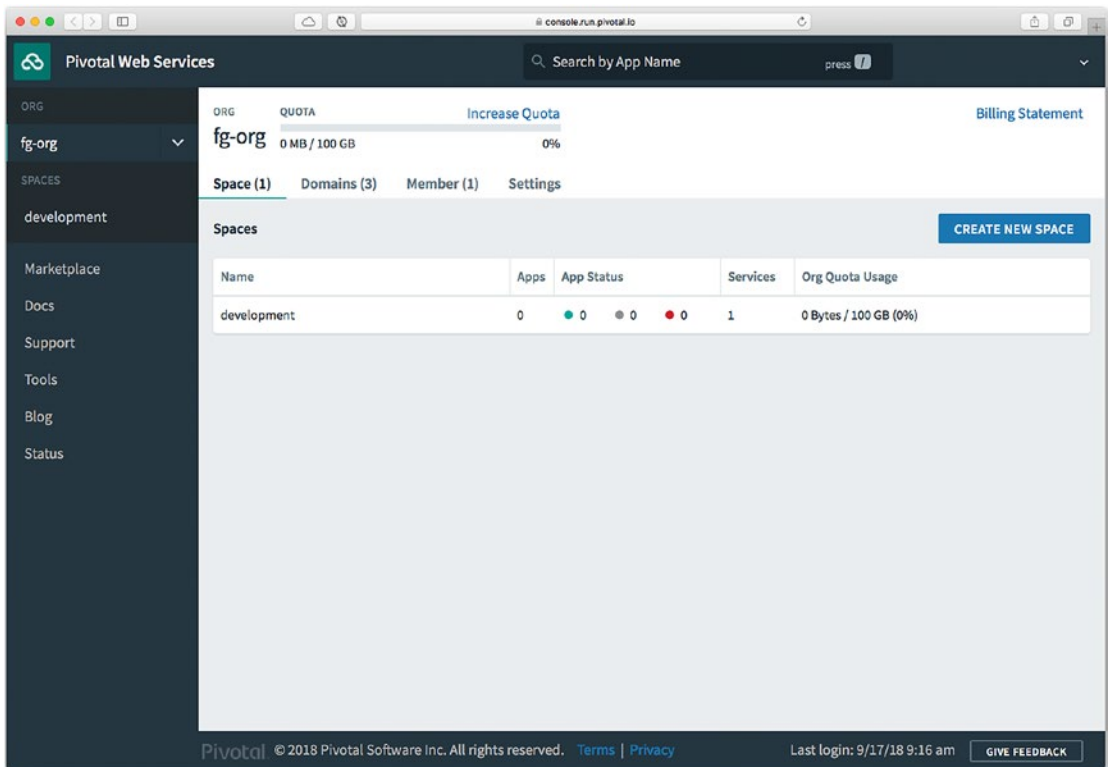


Figure 12-4. Pivotal Web Services

Now you are ready to deploy apps. By default, and because it is a trial account, you only have 2GB of memory, but this is enough to deploy the `ToDo` App. You can explore the tabs on the left.

The `Tools` tab shows links for downloading the CLI tool (which you install in the next section) and how to log in to the PWS instance.

Note In the next sections, I use PWS/PAS indistinctly, but it refers to Cloud Foundry.

Cloud Foundry CLI: Command-Line Interface

Before you start using PAS, you must install a command-line tool that is useful for deploying and do a lot of other tasks. If you are using a Windows OS, you can get the latest version from <https://github.com/cloudfoundry/cli#downloads> or you can use the Tools tab (from the previous section) and install based on your OS.

If you are using Mac OS/Linux, you can use brew.

```
$ brew update
$ brew tap cloudfoundry/tap
$ brew install cf-cli
```

After you install it, you can test it by running

```
$ cf --version
cf version 6.39.0+607d4f8be.2018-09-11
```

Now you are ready to use Cloud Foundry. Don't worry too much. I will show you the basic commands to get the ToDo App deployed and running.

Log in to PWS/PAS Using the CLI Tool

To log in to PWS and your account, you can execute the following command.

```
$ cf login -a api.run.pivotal.io
API endpoint: api.run.pivotal.io
Email> your-email@example.org
Password>
Authenticating...
OK

Targeted org your-org
Targeted space development

API endpoint: https://api.run.pivotal.io (API version: 2.121.0)
User: your-email@example.org
Org: your-org
Space: development
```

By default, you are placed in the development space. You are ready to execute commands for creating, deploying, scaling, and so forth to PWS (a PAS instance).

Deploying the ToDo App into PAS

It's time to deploy the ToDo App in PAS. It's important to know that the application that you deploy must have a unique subdomain. I'll talk about it later on.

Locate your JAR file (`todo-rest-0.0.1-SNAPSHOT.jar`). If you use Maven, it should be in the target directory. If you use Gradle, it should be in the `build/libs` directory.

To push an application, you need to use the following command.

```
$ cf push <name-of-the-app> [options]
```

So, to deploy the ToDo app, you can execute the following command.

```
$ cf push todo-app -p todo-rest-0.0.1-SNAPSHOT.jar -n todo-app-fg
```

```
Pushing app todo-app to org your-org / space development as your-email@
example.org...
```

```
Getting app info...
```

```
Creating app with these attributes...
```

```
+ name:      todo-app
  path:      /Users/Books/pro-spring-boot-2nd/ch12/todo-rest/target/todo-
            rest-0.0.1-SNAPSHOT.jar
```

```
  routes:
```

```
+  todo-app-fg.cfapps.io
```

```
Creating app todo-app...
```

```
Mapping routes...
```

```
Comparing local files to remote cache...
```

```
Packaging files to upload...
```

```
...
```

```
...
```

```
      state      since      cpu      memory      disk
#0  running  T01:25:10Z  33.0%  292.7M of 1G  158.3M of 1G
```

The `cf` command offers several options.

- `-p`. Tell the `cf` command that it uploads a file or all the content of a specific directory.
- `-n`. Creates a subdomain that must be unique. By default, every app has the `<sub-domain>.cfapps.io` URI, which must be unique. You can omit the `-n` option, but `cf` takes the name of the app, and it can collide with other names. In this example, I use the `todo-app-[my-initials]` (`todo-app-fg`). I suggest that you do this.

Behind the scenes, the `ToDo` app is running in a container (not a Docker container). This container is created by `RunC` (<https://github.com/opencontainers/runc>), which uses the host's resources and is isolated without compromising security. Now, you can go to your browser and use the URI given; in this example, <https://todo-app-fg.cfapps.io/todos>.

Take a look at the PWS to see your app (see Figure 12-5).

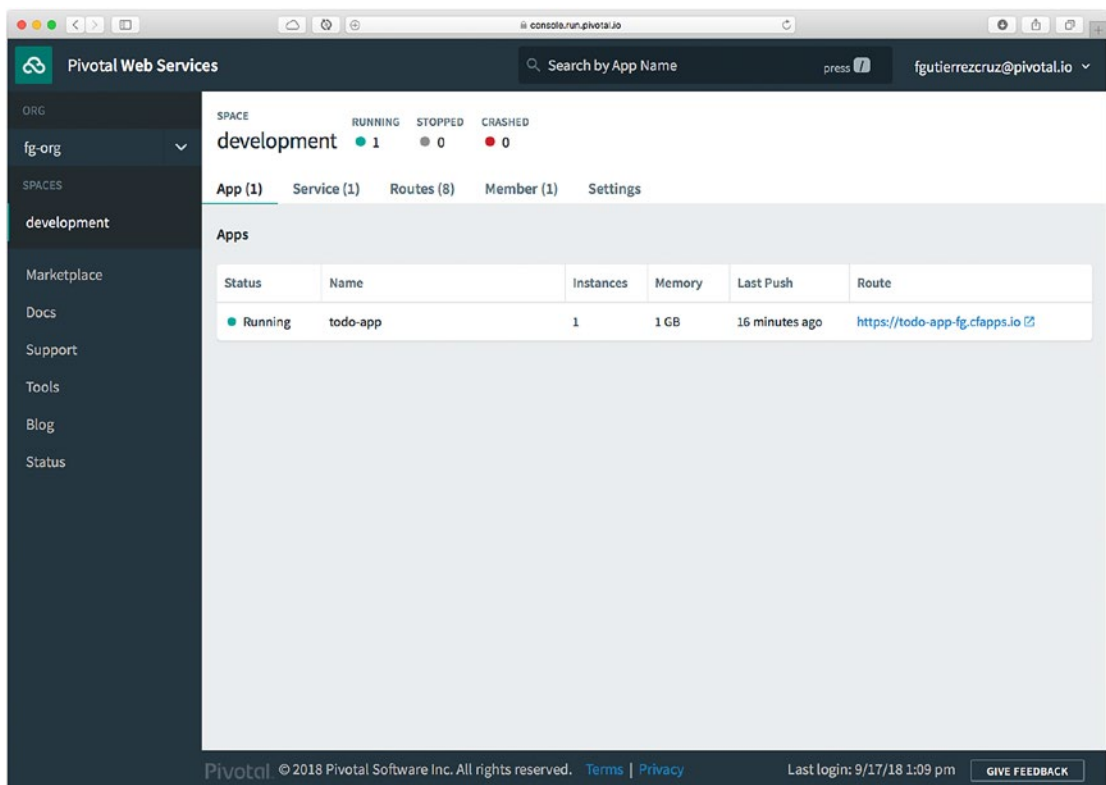


Figure 12-5. PWS `ToDo` App

If you hover over the name of the `todo-app`, you see what's shown in Figure 12-6.

The screenshot shows the Pivotal Web Services (PWS) console interface. The top navigation bar includes the PWS logo, a search bar for app names, and the user's email address. The left sidebar shows the organization 'fg-org' and the space 'development'. The main content area displays the details for the 'todo-app', which is currently 'Running'. The 'Overview' tab is active, showing a list of events, an app summary table, and a table of processes and instances.

Events (Last Push: 07:24 PM 09/17/18)

- Started app: fgutierrezcruz@pivotal.io 09/17/2018 at 07:24:16 PM
- Mapped route to app: fgutierrezcruz@pivotal.io 09/17/2018 at 07:23:56 PM
- Created app: fgutierrezcruz@pivotal.io 09/17/2018 at 07:23:55 PM

App Summary

Instances / Allocated	Memory / Allocated	Disk / Allocated
1 / 1	0.36 / 1.00 GB	0.15 / 1.00 GB

Processes and Instances (View in PCF Metrics)

web						
Instances	1	Memory Allocated	1 GB	Disk Allocated	1 GB	SCALE
<input type="checkbox"/> Autoscailing						
#	CPU	Memory	Disk	Uptime		
0	0%	365.44 MB	158.26 MB	16 min		

Footer: Pivotal © 2018 Pivotal Software Inc. All rights reserved. Terms | Privacy Last login: 9/17/18 1:09 pm GIVE FEEDBACK

Figure 12-6. PWS ToDo App details

You can inspect each link. You can check out the logs by clicking the Logs tab. You can get metrics by clicking the View in PCF Metrics link to learn more about your app's memory, requests per minute, CPU usage, disk, and so forth.

Another way to see the logs is by executing the following command.

```
$ cf logs todo-app
```

This command tails the logs. You can refresh or send requests to the app to see logs. It's a useful way to debug your application.

Creating Services

You can add ToDo's to the app by executing a command like the following.

```
$ curl -XPOST -d '{"description":"Learn to play Guitar"}' -H "Content-Type: application/json" https://todo-app-fg.cfapps.io/todos
```

```
{
  "description" : "Learn to play Guitar",
  "created" : "2018-09-18T01:58:34.211",
  "modified" : "2018-09-18T01:58:34.211",
  "completed" : false,
  "_links" : {
    "self" : {
      "href" : "https://todo-app-fg.cfapps.io/todos/8a70ee1f65ea47de0165ea668de30000"
    },
    "todo" : {
      "href" : "https://todo-app-fg.cfapps.io/todos/8a70ee1f65ea47de0165ea668de30000"
    }
  }
}
```

So, where has the previous ToDo been saved? Remember that the app has two drivers: one is the H2 (in-memory) and the other is MySQL, right? Deploying this app to PWS uses the same H2 driver as local. Why? Because we haven't specified any external MySQL service.

PAS offers a way to create services. If you review the section of the twelve-factor principles, you see that there is an item that talks about using services as attached resources. PAS helps with that by provisioning a service so you don't need to worry about installing it, hardening it, or managing it. PAS calls this a *managed service*.

Let's see how many services PWS has. You can execute the following command.

```
$ cf marketplace
```

This command prints out all available managed services that were installed and are provisioned by PAS. In this case, we are going to use the ClearDB service that has a MySQL service.

To tell PAS that we are going to create a `cleardb` instance service, you need to execute the command.

```
$ cf create-service <provider> <plan> <service-name>
```

So, to use a MySQL service, execute the following command.

```
$ cf create-service cleardb spark mysql
Creating service instance mysql in org your-org / space development as
your-email@example.org...
OK
```

The plan you chose was the `spark` plan, which is a free plan. If you choose something different, you need to add your credit card and expect to be charged every month.

You can review the services with the following command.

```
$ cf services
Getting services in org your-org / space development as your -email@
example.org...
```

name	service	plan	bound apps	last operation
mysql	cleardb	spark		create succeeded

See from previous command that the *bound apps* column is empty. Here we need to tell the `ToDo` app to use this service (`mysql`). To bound the app with the service, you can execute the following command.

```
$ cf bind-service todo-app mysql
Binding service mysql to app todo-app in org your-org / space development
as your-email@example.org...
OK
```

TIP: Use `'cf restage todo-app'` to ensure your env variable changes take effect

Behind the scenes, the container in which the `ToDo` app is running creates an environment variable with all the credentials, `VCAP_SERVICES`; , so it is easy for the `ToDo` app to connect to the `mysql` service. For the `ToDo` app to recognize this environment variable, it is necessary to restart the app. You can execute the following command.

```
$ cf restart todo-app
```

After the app restarts, take a look if it's working. Go to the browser and add `ToDo`'s. Let's take a look at the `VCAP_SERVICES` environment variable. Execute the following command.

```
$ cf env todo-app
```

```
Getting env variables for app todo-app in org your-org / space development
as your-email@example.org...
```

```
OK
```

```
System-Provided:
```

```
{
  "VCAP_SERVICES": {
    "cleardb": [
      {
        "binding_name": null,
        "credentials": {
          "hostname": "us-cdbr-iron-east-01.cleardb.net",
          "jdbcUrl": "jdbc:mysql://us-cdbr-iron-east-01.cleardb.net/ad_9a533ebf2
            e8e79a?user=b2c041b9ef8f25\u0026password=30e7a38b",
          "name": "ad_9a533ebf2e8e79a",
          "password": "30e7a38b",
          "port": "3306",
          "uri": "mysql://b2c041b9ef8f25:30e7a38b@us-cdbr-iron-east-01.cleardb.
            net:3306/ad_9a533ebf2e8e79a?reconnect=true",
          "username": "b2c041b9ef8f25"
        },
        "instance_name": "mysql",
        "label": "cleardb",
        "name": "mysql",
        "plan": "spark",...
```

```
...
```

```
....
```

See that the `VCAP_SERVICES` variable has the `hostname`, `username`, `password`, and the `jdbcUrl` properties. Actually, you can connect to it. You can use any MySQL client and use those properties. For example, if you have the `mysql` client command line, you can execute

```
$ mysql -h us-cdbr-iron-east-01.cleardb.net -ub2c041b9ef8f25 -p30e7a38b
ad_9a533ebf2e8e79a
...
...
mysql> show tables;
+-----+
| Tables_in_ad_9a533ebf2e8e79a |
+-----+
| to_do                          |
+-----+
1 row in set (0.07 sec)

mysql> select * from to_do \G
***** 1. row *****
      id: 8a72072165ea86ef0165ea887cd10000
    completed:
      created: 2018-09-18 02:35:38
description: Learn to play Guitar
      modified: 2018-09-18 02:35:38
1 row in set (0.07 sec)

mysql>
```

As you can see, now the `ToDo` app is using the MySQL service. But how? Cloud Foundry uses *buildpacks* that inspect your app and knows which programming language you are trying to run. Cloud Foundry is programming-language agnostic; so, it recognizes that you are using a Spring Boot app. It also sees that you have a bounded service (`mysql`), so it checks if you have the right drivers (in this case, it is the MySQL driver that is embedded in the fat JAR) so it can connect to it. But what is the best part? Well, you don't even need to change anything in your code!! Cloud Foundry and the Java buildpack auto-configures the datasource on your behalf. That's simple!

Cloud Foundry helps you focus only on your application, without worrying about infrastructure, services, and more.

Everything that you did to create a service can be done using the web console. You can client in the Marketplace tab on the PWS page, and select the service that you need (see Figure 12-7).

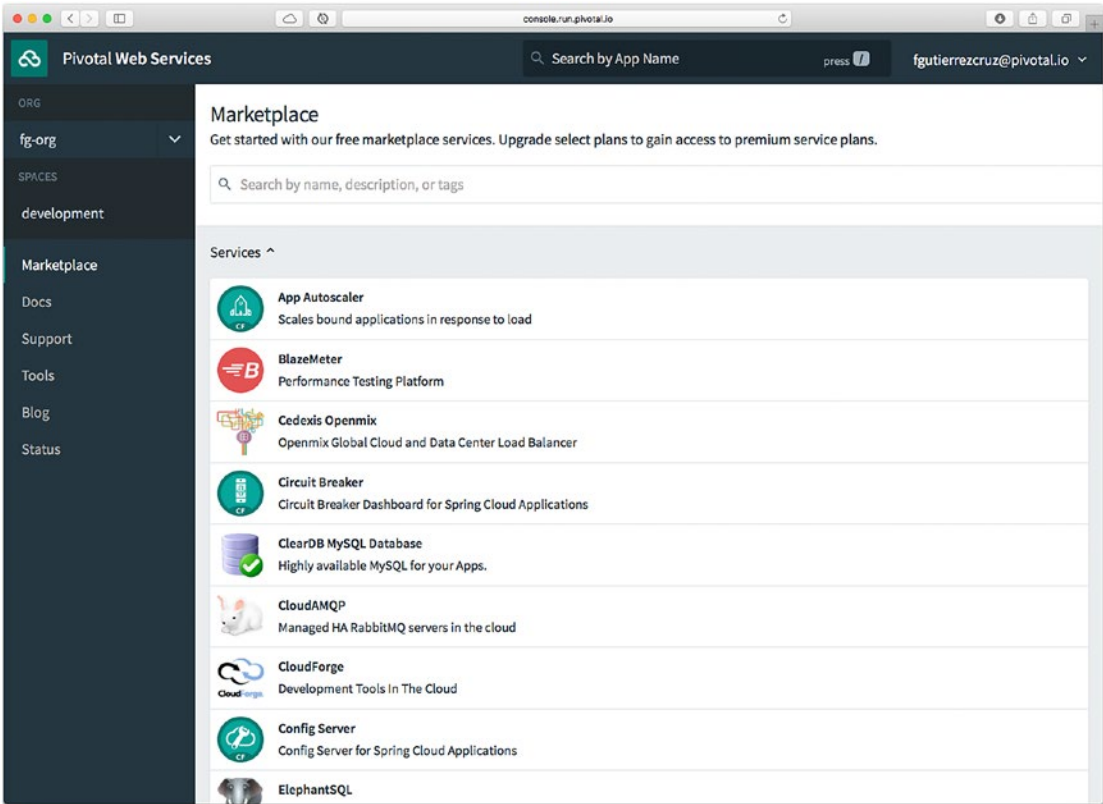


Figure 12-7. PWS Marketplace

You can click the ClearDB MySQL Database icon and select the Spark plan to configure it.

Congratulations! Now you know how to deploy Spring Boot apps into the cloud. Can you deploy the other examples? How about the `todo-mongo`, `todo-redis`, `todo-rabbitmq` projects?

Cleaning Up

It's important to clean your services and your apps. This helps you to use more credits when you need it. Let's start by unbinding the service from the app. Execute the following command.

```
$ cf unbind-service todo-app mysql
Unbinding app todo-app from service mysql in org your-org / space
development as your-email@example.org...
OK
```

Then, let's delete this service with the following command.

```
$ cf delete-service mysql

Really delete the service mysql?> y
Deleting service mysql in org your-org / space development as your-email@
exmaple.org...
OK
```

As you can see, you are prompted to confirm that you want to remove the service. You can use the `-f` flag to avoid this.

Lastly, let's remove the app. Execute the following command.

```
$ cf delete -f todo-app
Deleting app todo-app in org your-org / space development as your-email@
example.org...
OK
```

You can execute

```
$ cf apps
Getting apps in org your-org / space development as your-email@example.org...
OK
```

No apps found

to see if your current apps running.

Note Remember that you can get the book source code from the Apress website or on GitHub at <https://github.com/Apress/pro-spring-boot-2>.

Summary

In this chapter, you learned more about microservices and the cloud. You learned more about the twelve-factor principles that help you to create cloud-native applications.

You also learned about Cloud Foundry and what it offers, including the Pivotal Application Service and the Pivotal Container Service. You learned that Cloud Foundry is programming-language agnostic and that buildpacks inspect your app and auto-configures it.

In the next chapter, you learn how to extend and create your own spring-boot-start technology.

CHAPTER 13

Extending Spring Boot

Developers and software architects are often looking for design patterns to apply, new algorithms to implement, reusable components that are easy to use and maintain, and new ways to improve development. It's not always easy to find a unique or perfect solution, and it's necessary to use different technologies and methodologies to accomplish the goal of having an application that runs and never fails.

This chapter explains how the Spring and Spring Boot teams created a pattern for reusable components that are easy to use and implement. Actually, you have been learning about this pattern throughout the entire book, especially in the Spring Boot configuration chapter.

This chapter covers auto-configuration in detail, including how you can extend and create new Spring Boot modules that are reusable. Let's get started.

Creating a spring-boot-starter

In this section, I show you how to create a custom `spring-boot-starter`, but let's discuss some of the requirements first. Because you are working in the `ToDo` app, this custom starter is a client that you can use to do any operations for the `ToDo`'s, such as `create`, `find`, and `findAll`. This client needs a host that connects to a `ToDo` REST API service.

Let's start by setting up the project. So far, there is no template that sets up a baseline for a custom `spring-boot-starter`, so, we need to do this manually. Create the following structure.

```
todo-client-starter/  
├── todo-client-spring-boot-autoconfigure  
└── todo-client-spring-boot-starter
```

You need to create a folder named `todo-client-starter`, where you create two subfolders: `todo-client-spring-boot-autoconfigure` and `todo-client-spring-boot-starter`. Yes, there is a naming convention here. The Spring Boot team suggests that any custom starter follow this naming convention: `<name-of-starter>-spring-boot-starter` and `<name-of-starter>-spring-boot-autoconfigure`. The `autoconfigure` module has all the code and necessary dependencies that the starter needs; don't worry, I give you the information on what is needed.

First, let's create a main `pom.xml` file that has two modules: `autoconfigure` and `starter`. Create a `pom.xml` file in the `todo-client-starter` folder. Your structure should look like this:

```
todo-client-starter/
├── pom.xml
├── todo-client-spring-boot-autoconfigure
└── todo-client-spring-boot-starter
```

The `pom.xml` file looks Listing 13-1.

Listing 13-1. `todo-client-starter/pom.xml`

```
<?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.todo</groupId>
  <artifactId>todo-client</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>pom</packaging>
  <name>todo-client</name>

  <modules>
    <module>todo-client-spring-boot-autoconfigure</module>
    <module>todo-client-spring-boot-starter</module>
  </modules>
```

```

<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-dependencies</artifactId>
      <version>2.0.5.RELEASE</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
</project>

```

Listing 13-1 shows the main `pom.xml` that has two modules. One important thing to mention is that the `<packaging/>` tag is a `pom`, because at the end it is necessary to install these jars into the local repo to be used later. Also important is that this `pom` is declaring a `<dependencyManagement/>` tag that allows us to use the Spring Boot jars and all its dependencies. At the end, we don't need to declare versions.

todo-client-spring-boot-starter

Next, let's create another `pom.xml` file in the `todo-client-spring-boot-starter` folder. You should have the following structure.

```

todo-client-starter/
├── pom.xml
├── todo-client-spring-boot-autoconfigure
└── todo-client-spring-boot-starter
    └── pom.xml

```

See Listing 13-2.

Listing 13-2. `todo-client-starter/todo-client-spring-boot-starter/pom.xml`

```

<?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.todo</groupId>
<artifactId>todo-client-spring-boot-starter</artifactId>
<version>0.0.1-SNAPSHOT</version>
<packaging>jar</packaging>

<name>todo-client-spring-boot-starter</name>
<description>Todo Client Spring Boot Starter</description>

<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-8</project.reporting.
    outputEncoding>
</properties>

<parent>
    <groupId>com.apress.todo</groupId>
    <artifactId>todo-client</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <relativePath>..</relativePath>
</parent>

<dependencies>
    <dependency>
        <groupId>com.apress.todo</groupId>
        <artifactId>todo-client-spring-boot-autoconfigure</artifactId>
        <version>0.0.1-SNAPSHOT</version>
    </dependency>
</dependencies>

</project>

```

As you can see, Listing 13-2 is nothing new. It is declaring a `<parent/>` tag that relates to the previous `pom.xml` file, and it is declaring the `autoconfigure` module.

That's it for the `todo-client-spring-boot-starter`; nothing else. You can see this as a marker where you declare the modules that do the heavy work.

todo-client-spring-boot-autoconfigure

Next, let's create the structure for within the `todo-client-spring-boot-autoconfigure` folder. You should have the following final structure.

```

todo-client-starter/
├── pom.xml
├── todo-client-spring-boot-autoconfigure
│   ├── pom.xml
│   └── src
│       ├── main
│       │   ├── java
│       │   └── resources
└── todo-client-spring-boot-starter
    └── pom.xml

```

Your `todo-client-spring-boot-autoconfigure` folder should look like this:

```

todo-client-spring-boot-autoconfigure/
├── pom.xml
└── src
    ├── main
    │   ├── java
    │   └── resources

```

A basic Java project structure. Let's start with the `pom.xml` file (see Listing 13-3).

Listing 13-3. `todo-client-starter/todo-client-spring-boot-autoconfigure`

```

<?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.todo</groupId>
<artifactId>todo-client-spring-boot-autoconfigure</artifactId>
<version>0.0.1-SNAPSHOT</version>
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <configuration>
        <source>8</source>
        <target>8</target>
      </configuration>
    </plugin>
  </plugins>
</build>
<packaging>jar</packaging>

<name>todo-client-spring-boot-autoconfigure</name>
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <project.reporting.outputEncoding>UTF-8</project.reporting.
  outputEncoding>
  <java.version>1.8</java.version>
</properties>

<parent>
  <groupId>com.apress.todo</groupId>
  <artifactId>todo-client</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <relativePath>..</relativePath>
</parent>

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

```

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

<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
  <version>1.18.2</version>
</dependency>

<dependency>
  <groupId>org.springframework.hateoas</groupId>
  <artifactId>spring-hateoas</artifactId>
</dependency>

<!-- JSON / Traverson -->
<dependency>
  <groupId>org.springframework.plugin</groupId>
  <artifactId>spring-plugin-core</artifactId>
</dependency>

<dependency>
  <groupId>com.jayway.jsonpath</groupId>
  <artifactId>json-path</artifactId>
</dependency>

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-configuration-processor</artifactId>
  <optional>>true</optional>
</dependency>

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

```

    <dependency>
      <groupId>org.springframework.security</groupId>
      <artifactId>spring-security-test</artifactId>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>

```

In this case, the `autoconfigure` project depends on the `web`, `Lombok`, `security`, `Hateoas`, and `JsonPath`.

spring.factories

If you remember from the first chapters, I told you about the Spring Boot way to auto-configure everything based on the classpath; that's the real magic behind Spring Boot. I mentioned that when the application starts, the Spring Boot auto-configuration loads all the classes from the `META-INF/spring.factories` files to do each auto-configuration class, which provides the app the defaults it needs to run. Remember, Spring Boot is an *opinionated runtime* for Spring applications.

Let's create the `spring.factories` file that contains the class that does the auto-configuration and sets some defaults (see Listing 13-4).

Listing 13-4. `src/main/resources/META-INF/spring.factories`

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=com.apress.
todo.configuration.ToDoClientAutoConfiguration
```

Note that the `spring.factories` file is declaring the `ToDoClientAutoConfiguration` class.

auto-configuration

Let's start by creating the `ToDoClientAutoConfiguration` class (see Listing 13-5).

Listing 13-5. `com.apress.todo.configuration.ToDoClientAutoConfiguration.java`

```
package com.apress.todo.configuration;

import com.apress.todo.client.ToDoClient;
import lombok.RequiredArgsConstructor;
```



```

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.autoconfigure.condition.ConditionalOnClass;
import org.springframework.boot.context.properties.
EnableConfigurationProperties;
import org.springframework.boot.web.client.RestTemplateBuilder;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.hateoas.Resource;
import org.springframework.web.client.RestTemplate;

@RequiredArgsConstructor
@Configuration
@ConditionalOnClass({Resource.class, RestTemplateBuilder.class})
@EnableConfigurationProperties(ToDoClientProperties.class)
public class ToDoClientAutoConfiguration {

    private final Logger log = LoggerFactory.getLogger(ToDoClientAutoConfig
uration.class);
    private final ToDoClientProperties toDoClientProperties;

    @Bean
    public ToDoClient client(){
        log.info(">>> Creating a ToDo Client...");
        return new ToDoClient(new RestTemplate(),this.toDoClientProperties);
    }
}

```

Listing 13-5 shows the auto-configuration that is executed. It is using the `@ConditionalOnClass` annotation, which says that if it finds in the classpath, `Resource.class` and `RestTemplateBuilder.class` will continue with the configuration. Of course, because one of the dependencies is `spring-boot-starter-web`, it has those classes. But what happens when somebody excludes these resources? That's when this class does its job.

This class is declaring a `ToDoClient` bean that uses a `RestTemplate` and the `ToDoClientProperties` instances.

That's it. Very simple auto-configuration. It sets the default `ToDoClient` bean if it finds those resource classes in your project where you are using this custom starter.

Helper Classes

Next, let's create the helper classes. Create the `ToDoClientProperties` and the `ToDoClient` classes (see Listings 13-6 and 13-7).

Listing 13-6. `com.apress.todo.configuration.ToDoClientProperties.java`

```
package com.apress.todo.configuration;

import lombok.Data;
import org.springframework.boot.context.properties.ConfigurationProperties;

@Data
@ConfigurationProperties(prefix="todo.client")
public class ToDoClientProperties {

    private String host = "http://localhost:8080";
    private String path = "/toDos";

}
```

As you can see, nothing new—just two fields that hold a default for the host and the path. This means that you can override them in the `application.properties` files.

Listing 13-7. `com.apress.todo.client.ToDoClient.java`

```
package com.apress.todo.client;

import com.apress.todo.configuration.ToDoClientProperties;
import com.apress.todo.domain.ToDo;
import lombok.AllArgsConstructor;
import lombok.Data;
import org.springframework.core.ParameterizedTypeReference;
import org.springframework.hateoas.MediaTypes;
import org.springframework.hateoas.Resources;
import org.springframework.hateoas.client.Traverson;
import org.springframework.http.MediaType;
import org.springframework.http.RequestEntity;
import org.springframework.http.ResponseEntity;
import org.springframework.web.client.RestTemplate;
```

```

import org.springframework.web.util.UriComponents;
import org.springframework.web.util.UriComponentsBuilder;

import java.net.URI;
import java.util.Collection;

@AllArgsConstructor
@Data
public class TodoClient {

    private RestTemplate restTemplate;
    private TodoClientProperties props;

    public Todo add(Todo todo){
        UriComponents uriComponents = UriComponentsBuilder.newInstance()
            .uri(URI.create(this.props.getHost())).path(this.props.
                getPath()).build();

        ResponseEntity<Todo> response =
            this.restTemplate.exchange(
                RequestEntity.post(uriComponents.toUri())
                    .body(todo)
                    ,new ParameterizedTypeReference<Todo>() {});

        return response.getBody();
    }

    public Todo findById(String id){
        UriComponents uriComponents = UriComponentsBuilder.newInstance()
            .uri(URI.create(this.props.getHost())).pathSegment(this.
                props.getPath(), "{id}")
            .buildAndExpand(id);

        ResponseEntity<Todo> response =
            this.restTemplate.exchange(
                RequestEntity.get(uriComponents.toUri()).
                    accept(MediaType.HAL_JSON).build()
                    ,new ParameterizedTypeReference<Todo>() {});

        return response.getBody();
    }
}

```

```

public Collection<ToDo> findAll() {
    UriComponents uriComponents = UriComponentsBuilder.newInstance()
        .uri(URI.create(this.props.getHost())).build();

    Traverson traverson = new Traverson(uriComponents.toUri(),
        MediaType.HAL_JSON, MediaType.APPLICATION_JSON_UTF8);
    Traverson.TraversalBuilder tb = traverson.follow(this.props.
        getPath().substring(1));
    ParameterizedTypeReference<Resources<ToDo>> typeRefDevices =
        new ParameterizedTypeReference<Resources<ToDo>>() {};

    Resources<ToDo> toDos = tb.toObject(typeRefDevices);

    return toDos.getContent();
}
}

```

The `ToDoClient` class is a very straightforward implementation. This class is using the `RestTemplate` in all the methods; even though the `findAll` method is using a *Traverson* (an implementation in Java of the JavaScript Traverson library (<https://github.com/traverson/traverson>), which is a way to manipulate all the HATEOAS links) instance; behind the scenes it is using the `RestTemplate`.

Take a few minutes to analyze the code. Remember that this is a client that is requesting and posting to a `ToDo`'s REST API server.

To use this client, it is necessary to create the `ToDo` domain class (see Listing 13-8).

Listing 13-8. `com.apress.todo.domain.ToDo.java`

```

package com.apress.todo.domain;

import com.fasterxml.jackson.annotation.JsonFormat;
import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
import com.fasterxml.jackson.databind.annotation.JsonDeserialize;
import com.fasterxml.jackson.datatype.jsr310.deser.
    LocalDateTimeDeserializer;
import lombok.Data;
import lombok.NoArgsConstructor;

import java.time.LocalDateTime;

```

```

@JsonIgnoreProperties(ignoreUnknown = true)
@NoArgsConstructor
@Data
public class ToDo {

    private String id;
    private String description;

    @JsonDeserialize(using = LocalDateTimeDeserializer.class)
    @JsonFormat(pattern = "yyyy-MM-dd HH:mm:ss")
    private LocalDateTime created;

    @JsonDeserialize(using = LocalDateTimeDeserializer.class)
    @JsonFormat(pattern = "yyyy-MM-dd HH:mm:ss")
    private LocalDateTime modified;

    private boolean completed;

    public ToDo(String description){
        this.description = description;
    }
}

```

Here we are introducing `@Json*` annotations. One ignores any of the links (provided by the HAL+JSON protocol) and one serializes the `LocalDateTime` instances.

We are almost done. Let's add a security utility that helps encrypt/decrypt `ToDo` descriptions.

Creating an `@Enable*` Feature

One of the cool features of the Spring and Spring Boot technologies is that they expose several `@Enable*` features that hide all the boilerplate configuring and does the heavy lifting for us.

So, let's create a custom `@EnableToDoSecurity` feature. Let's start by creating the annotation that is picked up by Spring Boot, auto-configuration (see Listing 13-9).

Listing 13-9. `com.apress.todo.annotation.EnableToDoSecurity.java`

```

package com.apress.todo.annotation;

import com.apress.todo.security.ToDoSecurityConfiguration;
import org.springframework.context.annotation.Import;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Import(ToDoSecurityConfiguration.class)
public @interface EnableToDoSecurity {
    Algorithm algorithm() default Algorithm.BCRYPT;
}

```

This annotation uses an algorithm enum; let's create it (see Listing 13-10).

Listing 13-10. `com.apress.todo.annotation.Algorithm.java`

```

package com.apress.todo.annotation;

public enum Algorithm {
    BCRYPT, PBKDF2
}

```

This means that we can have some parameters passed to the `@EnableToDoSecurity`. We choose either `BCRYPT` or `PBKDF2`, and if there is no parameter, by default it is `BCRYPT`.

Next, create the `ToDoSecurityConfiguration` class that triggers any configuration if the `@EnableToDoSecurity` is declared (see Listing 13-11).

Listing 13-11. `com.apress.todo.security.ToDoSecurityConfiguration.java`

```

package com.apress.todo.security;

import com.apress.todo.annotation.Algorithm;
import com.apress.todo.annotation.EnableToDoSecurity;

```

```

import org.springframework.context.annotation.ImportSelector;
import org.springframework.core.annotation.AnnotationAttributes;
import org.springframework.core.type.AnnotationMetadata;

public class ToDoSecurityConfiguration implements ImportSelector {
    public String[] selectImports(AnnotationMetadata annotationMetadata) {
        AnnotationAttributes attributes =
            AnnotationAttributes.fromMap(
                annotationMetadata.getAnnotationAttributes(EnableToDoSecurity.class.getName(), false));
        Algorithm algorithm = attributes.getEnum("algorithm");
        switch(algorithm){
            case PBKDF2:
                return new String[] {"com.apress.todo.security.Pbkdf2Encoder"};
            case BCRYPT:
            default:
                return new String[] {"com.apress.todo.security.BCryptEncoder"};
        }
    }
}

```

Listing 13-11 shows you that the auto-configuration is executed only if the `@EnableToDoSecurity` annotation is declared. Spring Boot also tracks every class that implements the `ImportSelector` interface, which hides all the boilerplate processing annotations.

So, if the `@EnableToDoSecurity` annotation is found, then this class is executed by calling the `selectImports` method that returns an array of Strings that are classes that have to be configured; in this case, either the `com.apress.todo.security.Pbkdf2Encoder` class (if you set the `PBKDF2` algorithm as parameter) or the `com.apress.todo.security.BCryptEncoder` class (if you set the `BCRYPT` algorithm as the parameter).

What is special about these classes? Let's create the `BCryptEncoder` and `Pbkdf2Encoder` classes (see Listing 13-12 and Listing 13-13).

Listing 13-12. com.apress.todo.security.BCryptEncoder.java

```

package com.apress.todo.security;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

@Configuration
public class BCryptEncoder {

    @Bean
    public TodoSecurity utils(){
        return new TodoSecurity(new BCryptPasswordEncoder(16));
    }
}

```

Listing 13-13. com.apress.todo.security.Pbkdf2Encoder.java

```

package com.apress.todo.security;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.crypto.password.Pbkdf2PasswordEncoder;

@Configuration
public class Pbkdf2Encoder {

    @Bean
    public TodoSecurity utils(){
        return new TodoSecurity(new Pbkdf2PasswordEncoder());
    }
}

```

Both classes are declaring the `TodoSecurity` bean. So, if you choose the PBKDF2 algorithm, then the `TodoSecurity` bean is created with the `Pbkdf2PasswordEncoder` instance; and if you choose the BCRYPT algorithm, the `TodoSecurity` bean is created with the `BCryptPasswordEncoder(16)` instance.

Listing 13-14 shows the `TodoSecurity` class.

Listing 13-14. `com.apress.todo.security.ToDoSecurity.java`

```

package com.apress.todo.security;

import lombok.AllArgsConstructor;
import lombok.Data;
import org.springframework.security.crypto.password.PasswordEncoder;

@AllArgsConstructor
@Data
public class ToDoSecurity {

    private PasswordEncoder encoder;
}

```

As you can see, nothing special in this class.

ToDo REST API Service

Let's prepare the ToDo REST API service. You can reuse the `todo-rest` project that uses `data-jpa` and `data-rest`, which you did in other chapters. Let's review it and see what we need to do (see Listings [13-15](#), [13-16](#), and [13-17](#)).

Listing 13-15. `com.apress.todo.domain.ToDo.java`

```

package com.apress.todo.domain;

import com.fasterxml.jackson.annotation.JsonFormat;
import com.fasterxml.jackson.databind.annotation.JsonDeserialize;
import com.fasterxml.jackson.datatype.jsr310.deser.
LocalDateTimeDeserializer;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.hibernate.annotations.GenericGenerator;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.PrePersist;

```

```

import javax.persistence.PreUpdate;
import javax.validation.constraints.NotBlank;
import javax.validation.constraints.NotNull;
import java.time.LocalDateTime;

@Entity
@Data
@NoArgsConstructor
public class ToDo {

    @NotNull
    @Id
    @GeneratedValue(generator = "system-uuid")
    @GenericGenerator(name = "system-uuid", strategy = "uuid")
    private String id;
    @NotNull
    @NotBlank
    private String description;

    @Column(insertable = true, updatable = false)
    @JsonDeserialize(using = LocalDateTimeDeserializer.class)
    @JsonFormat(pattern = "yyyy-MM-dd HH:mm:ss")
    private LocalDateTime created;

    @JsonDeserialize(using = LocalDateTimeDeserializer.class)
    @JsonFormat(pattern = "yyyy-MM-dd HH:mm:ss")
    private LocalDateTime modified;

    private boolean completed;

    public ToDo(String description){
        this.description = description;
    }

    @PrePersist
    void onCreate() {
        this.setCreated(LocalDateTime.now());
        this.setModified(LocalDateTime.now());
    }
}

```

```

@PreUpdate
void onUpdate() {
    this.setModified(LocalDate.now());
}
}

```

This `ToDo` class is nothing new; you already know about every annotation used here. The only difference is that it is using the `@Json*` annotations only for the dates with a specific format.

Listing 13-16. `com.apress.todo.repository.ToDoRepository.java`

```

package com.apress.todo.repository;

import com.apress.todo.domain.ToDo;
import org.springframework.data.repository.CrudRepository;

public interface ToDoRepository extends CrudRepository<ToDo,String> {

}

```

The same as before; nothing that you don't know about this interface.

Listing 13-17. `com.apress.todo.config.ToDoRestConfig.java`

```

package com.apress.todo.config;

import com.apress.todo.domain.ToDo;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.rest.core.config.
RepositoryRestConfiguration;
import org.springframework.data.rest.webmvc.config.
RepositoryRestConfigurerAdapter;

@Configuration
public class ToDoRestConfig extends RepositoryRestConfigurerAdapter {

    @Override
    public void configureRepositoryRestConfiguration(RepositoryRest
    Configuration config) {

```

```

        config.exposeIdsFor(ToDo.class);
    }
}

```

Listing 13-17 shows you a new class, the `ToDoRestConfig` that is extending from the `RepositoryRestConfigurerAdapter`; this class can help configure part of the `RestController` implementation from everything that is by default configured by the JPA repositories auto-configuration. It overrides the `configureRepositoryRestConfiguration` by exposing the IDs of the domain class. When we worked with REST in other chapters, the IDs never showed upon request; but with this overriding, we can make it happen. And we need this feature because we want to get back the ID of the `ToDo` instance.

In `application.properties`, you should have the following.

```

# JPA
spring.jpa.generate-ddl=true
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true

```

Again, nothing new.

Installing and Testing

Let's prepare everything to run on a new custom starter. Let's start by installing the `todo-client-spring-boot-starter`. Open a terminal and go to your `todo-client-starter` folder and execute the following command.

```
$ mvn clean install
```

This command installs your jar in the local `.m2` directory, which is ready to be picked up by another project that uses it.

Task Project

Now that you installed `todo-client-spring-boot-starter`, it is time to test it. You are going to create a new project. You can create the project as usual. Go to the Spring Initializr (<https://start.spring.io>) and set the fields with the following values.

- Group: `com.apress.task`
- Artifact: `task`
- Name: `task`
- Package Name: `com.apress.task`

You can select either Maven or Gradle. Then click the Generate Project button. This generates and downloads a ZIP file. You can uncompress it, and then import it into your favorite IDE (see Figure 13-1).

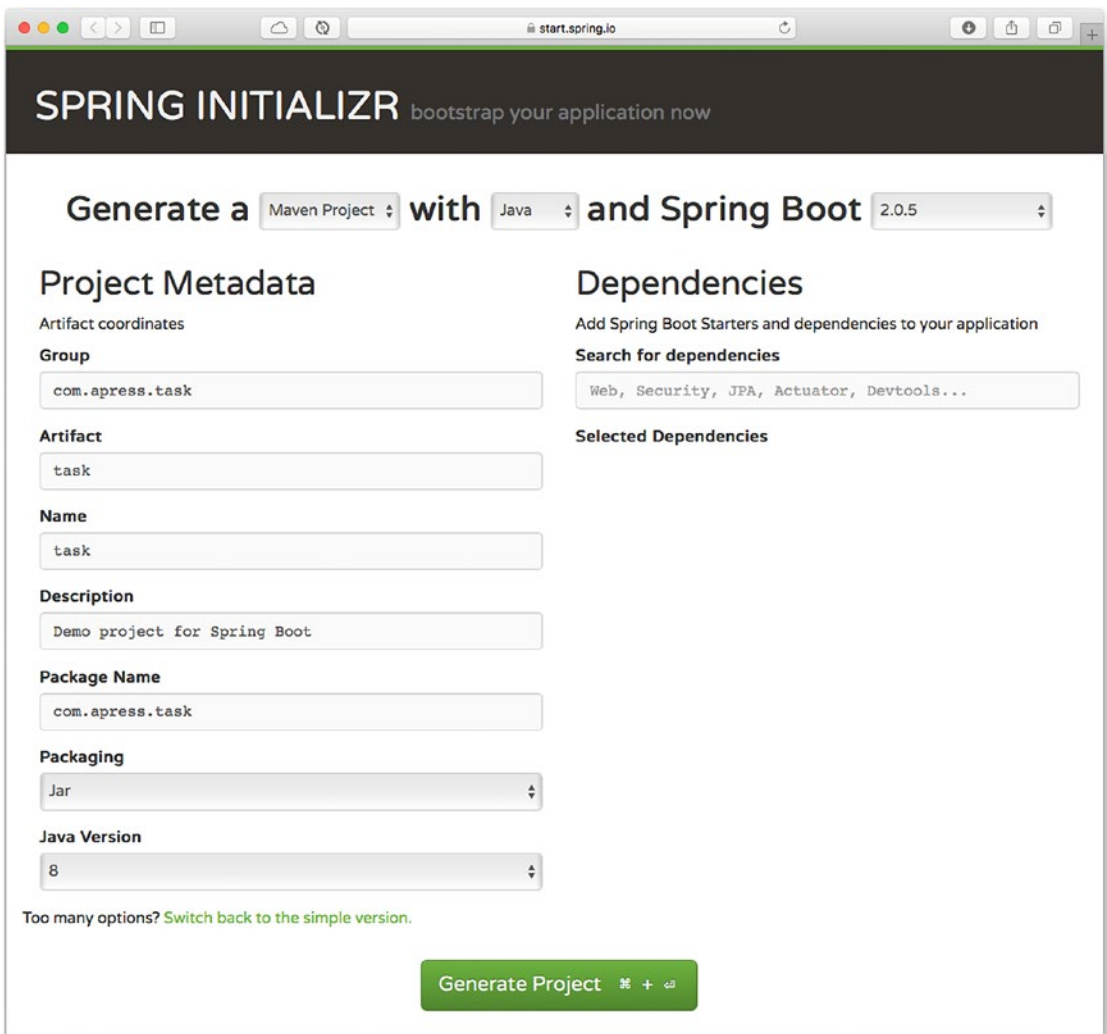


Figure 13-1. Spring Initializr

Next, you need to add `todo-client-spring-boot-starter`. If you are using Maven, go to your `pom.xml` file and add the dependency.

```
<dependency>
  <groupId>com.apress.todo</groupId>
  <artifactId>todo-client-spring-boot-starter</artifactId>
  <version>0.0.1-SNAPSHOT</version>
</dependency>
```

If you are using Gradle, add the dependency to your `build.gradle` file.

```
compile('com.apress.todo:todo-client-spring-boot-starter:0.0.1-SNAPSHOT')
```

That's it. Now open the main class, in which there is the code shown in Listing 13-18.

Listing 13-18. `com.apress.task.TaskApplication.java`

```
package com.apress.task;

import com.apress.todo.annotation.EnableToDoSecurity;
import com.apress.todo.client.ToDoClient;
import com.apress.todo.domain.ToDo;
import com.apress.todo.security.ToDoSecurity;
import org.springframework.boot.ApplicationRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.WebApplicationType;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

@EnableToDoSecurity
@SpringBootApplication
public class TaskApplication {

    public static void main(String[] args) {
        SpringApplication app = new SpringApplication(TaskApplication.class);
        app.setWebApplicationType(WebApplicationType.NONE);
        app.run(args);
    }
}
```

```

@Bean
ApplicationRunner createTodos(ToDoClient client){
    return args -> {
        ToDo todo = client.add(new ToDo("Read a Book"));
        ToDo review = client.findById(todo.getId());
        System.out.println(review);
        System.out.println(client.findAll());
    };
}

@Bean
ApplicationRunner secure(ToDoSecurity utils){
    return args -> {
        String text = "This text will be encrypted";
        String hash = utils.getEncoder().encode(text);
        System.out.println(">>> ENCRYPT: " + hash);
        System.out.println(">>> Verify: " + utils.getEncoder().
            matches(text,hash));
    };
}
}

```

There are two `ApplicationRunner` beans; each has a parameter. `createTodos` uses the `ToDoClient` bean instance (if you don't have the `RestTemplateBuilder` or `Resource`, it will fail). It is using the methods you know (`add`, `findById` and `findAll`).

The method `secure` is using the `ToDoSecurity` bean instance, which is possible thanks to `@EnableToDoSecurity`. If you remove it or comment it out, it tells you it can't find the `ToDoSecurity` bean.

Take a few minutes to analyze the code to see what's going on.

Running the Task App

To run the app, first make sure the `todo-rest` app is up and running. It should run on port 8080. Remember that you have already installed the `todo-client-spring-boot-starter` with the `mvn clean install` command.

So if you are running it, you see some responses, and the `ToDo` is saved in the `ToDo` REST service. It also shows you the encrypted text.

If you are running the `ToDo` REST API in a different IP, host, port, or path, you can change the defaults by using the `todo.client.*` properties in the application's `properties` file.

```
# ToDo Rest API
todo.client.host=http://some-new-server.com:9091
todo.client.path=/api/todos
```

Remember that if you don't override, it defaults to `http://localhost:8080` and `/todos`. After running the Task app, you should see something similar to the following output.

```
INFO - [ main] c.a.t.c.ToDoClientAutoConfiguration      : >>> Creating a
ToDo Client...
INFO - [ main] o.s.j.e.a.AnnotationMBeanExporter        : Registering beans
for JMX exposure on startup
INFO - [ main] com.apress.task.TaskApplication            : Started
TaskApplication in 1.047 seconds (JVM running for 1.853)
ToDo(id=8a8080a365f427c00165f42adee50000, description=Read a Book,
created=2018-09-19T17:29:34, modified=2018-09-19T17:29:34, completed=false)
[ToDo(id=8a8080a365f427c00165f42adee50000, description=Read a
Book, created=2018-09-19T17:29:34, modified=2018-09-19T17:29:34,
completed=false)]
>>> ENCRYPT: $2a$16$pVOI../twnLwN3GFiChdR.zRFfyCIZMEbwEXbAtRoIHqxeLB3gmUG
>>> Verify: true
```

Congratulations! You just have created your first custom Spring Boot starter and `@Enable` feature!

Note Remember that you can get the book source code from the Apress website or on GitHub at <https://github.com/Apress/pro-spring-boot-2>.

Summary

This chapter showed you how to create a module for Spring Boot by using the auto-configuration pattern. I showed you how to create your custom health monitor. As you can see, it's very simple to extend Spring Boot apps, so feel free to modify the code and experiment with them.

We didn't do much if any unit or integration testing. It would be good homework for you to practice all the details that I showed you. I think it will help you understand how Spring Boot works even better. Repeat and you will master!

APPENDIX A

Spring Boot CLI

This chapter discussed a Spring Boot tool that can help you create prototypes and production-ready applications. This tool is part of the Spring Boot installation that you performed in the first chapters. This is not a Maven or Gradle plugin or dependency.

I'm talking about the Spring Boot CLI (command-line interface). In the previous chapters, you learned that you can get the CLI from the binary installation at <http://repo.spring.io/release/org/springframework/boot/spring-boot-cli/>. Or, if you are using a Mac/Linux, you can use Homebrew (<http://brew.sh/>) with the following command.

```
$ brew tap pivotal/tap
$ brew install springboot
```

If you are using Linux, you can use the SDKMAN tool (<http://sdkman.io/>) and install it with the following command.

```
$ sdk install springboot
```

All the examples are in Java and Groovy. There is no real distinction between one language and another in the sense of compile, run, or package. The only difference is some extra parameters that pass to the command line. But don't worry too much; you'll see those in a moment.

Spring Boot CLI

When I first started to learn Spring Boot, which was around three years ago, it was the alpha release, and the only available command was `run`. What else do you need, right? It was amazing that with a few lines of code you could have a web application written in Groovy up and running. It was simple and awesome.

Since version 1.4, it has more options and an interactive shell that you'll see soon. To see the CLI in detail, you need to consider some simple examples. Let's start with the one in Listing A-1, which shows the Groovy example from other chapters.

Listing A-1. app.groovy

```
@RestController
class WebApp{

    @RequestMapping("/")
    String greetings(){
        "Spring Boot Rocks"
    }
}
```

Listing A-1 shows the simplest Groovy web application you can have and that you can run with Spring Boot. Now, let's look at the same web application but in Java (see Listing A-2).

Listing A-2. WebApp.java

```
package com.apress.spring;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@SpringBootApplication
public class WebApp {

    @RequestMapping("/")
    public String greetings(){
        return "Spring Boot Rocks in Java too!";
    }

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

}
```

Listing A-2 shows the Java version of the simplest web application. As I mentioned, Spring Boot enables you to choose Java or Groovy to create enterprise- and production-ready applications with ease.

Let's start using all the CLI commands.

The run Command

The run command allows you to run Java or Groovy Spring Boot applications. Its syntax is as follows.

```
spring run [options] <files> [--] [args]
```

The available options are shown in Table A-1.

Table A-1. *Spring Run Options*

Option	Description
--autoconfigure [Boolean]	Adds the auto-configuration compiler transformation. Remembers the auto-configuration features and how everything works by adding the <code>@EnableAutoConfiguration</code> or the composed <code>@SpringBootApplication</code> annotations. This is the same idea (default is true).
--classpath, -cp	Adds the classpath entries; it's useful when you have third-party libraries. As a recommendation, you can create a <code>lib/</code> folder in the root of your program and add all the classes or JARs there.
--no-guess- dependencies	Does not attempt to guess the dependencies. This is useful when you already use the <code>@Grab</code> annotation in your application.
--no-guess- imports	Does not attempt to guess the imports. This is useful when you have already included some of the imports in your Groovy application. For example, in a Java app you can use this option because you are importing the classes you need. There is more about this in Chapter 3 (in the auto-configuration section).
-q, --quiet	Quiets logging. In other words, it won't print anything to the console.
-v, --verbose	Logs everything. It is useful for seeing what's going on, because it shows the code introspection and what is added to the program. See Chapter 3 for more information.
--watch	Sets a watch to the file(s) for changes. It is useful when you don't want to stop and run the app again.

To run the Groovy application (shown in Listing A-1), you simply execute

```
$ spring run app.groovy
```

Executing this command has a web application up and running and listening to port 8080 by default, but you can override this by executing the following command.

```
$ spring run app.groovy -- --server.port=8888
```

This command runs the web application and it is listening in port 8888. Now, if you want to add some third-party library and load the dependencies, you simply execute

```
$ spring run -cp lib/mylib.jar app.groovy
```

If you want to run the Java application (see Listing A-2), you just execute

```
$ spring run WebApp.java
```

Note You can stop your application by pressing Ctrl+C in your keyboard. If you are running a Java application, it's important to add the package keyword. You don't need to have a hierarchy or create any directories. If you don't add a package to Spring Boot scanning, it will be impossible to run your app because it needs to scan all the available dependencies of Spring. It starts scanning all the dependencies used and start from the root of every dependency, so be careful!

If you have several files, you can use the wildcard * to compile all of them. Just execute this command:

```
$ spring run *.groovy
```

If, for some reason, you need to tweak the JVM and its options, you can execute the following command.

```
$ JAVA_OPTS=-Xmx2g spring run app.groovy
```

This command increases the memory heap up to 2GB for the app.groovy application.

The test Command

The test command runs a Spring Groovy script and Java tests. Its syntax is as follows.

```
spring test [options] files [--] [args]
```

The available options are shown in Table [A-2](#).

Table A-2. *Spring Test Options*

Option	Description
--autoconfigure [Boolean]	Adds the auto-configuration compiler transformation (default is true).
--classpath, -cp	Adds the classpath entries, which is useful when you have third-party libraries. As a recommendation, you can create a <code>lib/</code> folder in the root of your program and add all the classes or JARs there.
--no-guess- dependencies	Does not attempt to guess the dependencies. This is useful when you already use the <code>@Grab</code> annotation in your application.
--no-guess- imports	Does not attempt to guess the imports. This is useful when you already include some of the imports in your Groovy application. For example, in a Java app you can use this option because you are importing the classes you need. See more in Chapter 3 (in the auto-configuration section).

To run a test, you need a test, right? Listings [A-3](#), [A-4](#), and [A-5](#) show examples using the well-known JUnit and Spock frameworks.

Listing A-3. `test.groovy`

```
class MyTest{
    @Test
    void simple() {
        String str= "JUnit works with Spring Boot"
        assertEquals "JUnit works with Spring Boot",str
    }
}
```

Listing A-3 shows the simplest unit test. You don't need to use any imports; Spring Boot takes care of that. To run it, you execute

```
$ spring test test.groovy
```

Take a look at the Spock unit test shown in Listing A-4.

Listing A-4. `spock.groovy`

```
@Grab('org.spockframework:spock-core:1.0-groovy-2.4')
import spock.lang.Specification
import org.springframework.boot.test.OutputCapture

class SimpleSpockTest extends Specification {

    @org.junit.Rule
    OutputCapture capture = new OutputCapture()

    def "get output and capture it"() {
        when:
            print 'Spring Boot works with Spock'

        then:
            capture.toString() == 'Spring Boot works with Spock'
    }
}
```

Listing A-4 shows the use of the Spock Framework by extending the Specification class and defining the methods. To use the Spock Framework it's necessary to import the necessary dependencies and to include those dependencies by adding the @Grab annotation that includes the Spock dependency for Groovy. The intention of this section is to show the usage of Spock. But if you are looking for more information about it, you can go to <http://spockframework.org/>. All its documentation is found at <http://spockframework.github.io/spock/docs/1.0/index.html>.

Listing A-4 also shows one of the new features of Spring Boot, which is the OutputCapture class. It allows you to capture output from System.out and System.err. To run this test, you execute the same instruction but change the name of the file.

```
$ spring test spock.groovy
```

It's important to know that Spring Boot won't always figure out when you are using third-party libraries, so you must use the `@Grab` annotation and the correct `import`.

Take a look at the unit test in Java, shown in Listing A-5.

Listing A-5. MyTest.java

```
import org.junit.Rule;
import org.junit.Test;
import org.springframework.boot.test.OutputCapture;

import static org.hamcrest.Matchers.*;
import static org.junit.Assert.*;

public class MyTest {

    @Rule
    public OutputCapture capture = new OutputCapture();

    @Test
    public void stringTest() throws Exception {
        System.out.println("Spring Boot Test works in Java too!");
        assertThat(capture.toString(), containsString("Spring Boot
        Test works in Java too!"));
    }
}
```

Listing A-5 shows a unit test written in Java. The `assertThat` statement belongs to the `org.junit.Assert` class, which can be accessed as static. The `containsString` is a static method from the `org.hamcrest.Matchers` class, and it matches the capture string. This unit test also uses the `OutputCapture` class. To run it, you just execute this command:

```
$ spring test MyTest.java
```


If you want to test the `app.groovy` web application (see Listing A-1), you can create the code in Listing A-6.

Listing A-6. `test.groovy`

```
class SimpleWebTest {  
  
    @Test  
    void greetingsTest() {  
        assertEquals("Spring Boot Rocks", new WebApp().greetings())  
    }  
}
```

To test this, just execute the following command.

```
$ spring test app.groovy test.groovy
```

This command uses the previous class—the `WebApp` (see Listing A-1)—and it calls the `greetings` method to get the string back.

Although these examples are extremely simple, it's important to see how easy it is to create and run tests using the command-line interface. A special chapter includes a more elaborated unit and integration test using all the power of Spring Boot.

The `grab` Command

The `grab` command downloads all the Spring Groovy scripts and Java dependencies to the `./repository` directory. Its syntax is as follows.

```
spring grab [options] files [--] [args]
```

The available options are shown in Table A-3.

Table A-3. *Grab Options*

Option	Description
<code>--autoconfigure</code> [Boolean]	Adds the auto-configuration compiler transformation (default is true).
<code>--classpath, -cp</code>	Adds the classpath entries, which is useful when you have third-party libraries. As a recommendation, you can create a <code>lib/</code> folder in the root of your program and add all the classes or JARs there.
<code>--no-guess-dependencies</code>	Does not attempt to guess the dependencies. This is useful when you already use the <code>@Grab</code> annotation in your application.
<code>--no-guess-imports</code>	Does not attempt to guess the imports. This is useful when you already include some of the imports in your Groovy application. For example, in a Java app you can use this option because you are importing the classes you need. For more information, see Chapter 3 (in the auto-configuration section).

You can use any of the listings you’ve seen so far to execute the `grab` command. For [Listing A-4](#), you can execute

```
$ spring grab MyTest.java
```

If you check out the current directory, you see the repository subdirectory created with all the dependencies. The `grab` command is useful when you want to execute a Spring Boot application that doesn’t have an Internet connection and the libraries are needed. The `grab` command is also used to prepare your application before you can deploy it to the cloud. (You’ll see this useful command in [Chapter 13](#), “Spring Boot in the Cloud.”)

The jar Command

The `jar` command creates a self-contained executable JAR file from a Groovy or Java script. Its syntax is as follows.

```
spring jar [options] <jar-name> <files>
```

The available options are shown in Table A-4.

Table A-4. Jar Options

Option	Description
--autoconfigure [Boolean]	Adds the auto-configuration compiler transformation (default is true).
--classpath, -cp	Adds the classpath entries, which is useful when you have third-party libraries. As a recommendation, you can create a lib/ folder in the root of your program and add all the classes or JARs there.
--exclude	A pattern to find the files and exclude them from the final JAR file.
--include	A pattern to find the files and include them in the final JAR file.
--no-guess- dependencies	Does not attempt to guess the dependencies. This is useful when you already use the @Grab annotation in your application.
--no-guess- imports	Does not attempt to guess the imports. This is useful when you already include some of the imports in your Groovy application. For example, in a Java app you can use this option because you are importing the classes you need. For more information, see Chapter 3 (the auto-configuration section).

You can use Listing A-1 (app.groovy) and execute the following command.

```
$ spring jar app.jar app.groovy
```

Now you can check out your current directory and see that there are two files—one named app.jar.original and another named app.jar. The only difference between the files is that the app.jar.original is the one created by the dependency management (Maven) to create the app.jar. It’s a fat JAR that can be executed with the following.

```
$ java -jar app.jar
```

By executing this command, you have a web application up and running. The jar command enables application portability, because you can ship your application and run it in any system that has Java installed, without worrying about an application container. Remember that Spring Boot embeds the Tomcat application server in a Spring Boot web application.

The war Command

This is very similar to the previous command. The war command creates a self-contained executable WAR file from a Groovy or Java script. Its syntax is as follows.

```
spring war [options] <war-name> <files>
```

The available options are shown in Table A-5.

Table A-5. War Options

Option	Description
<code>--autoconfigure</code> [Boolean]	Adds the auto-configuration compiler transformation (default is true).
<code>--classpath,</code> <code>-cp</code>	Adds the classpath entries, which is useful when you have third-party libraries. As a recommendation, you can create a <code>lib/</code> folder in the root of your program and add all the classes or JARs there.
<code>--exclude</code>	A pattern to find the files and exclude them from the final JAR file.
<code>--include</code>	A pattern to find the files and include them in the final JAR file.
<code>--no-guess-</code> <code>dependencies</code>	Does not attempt to guess the dependencies. This is useful when you already use the <code>@Grab</code> annotation in your application.
<code>--no-guess-</code> <code>imports</code>	Does not attempt to guess the imports. This is useful when you already include some of the imports in your groovy application. For example, in a Java app you can use this option because you are importing the classes you need. For more information, see Chapter 3 (the auto-configuration section).

You can use Listing A-1 (`app.groovy`) to run the war command by executing the following.

```
$ spring war app.war app.groovy
```

After executing this command, you have in your current directory the `app.war` original and the `app.war` files. You can run it with the following command.

```
$ java -jar app.war
```

In the previous command I mentioned the word portability, right? So what would be the case for a WAR file? Well, you can use the WAR file in existing application containers like Pivotal tcServer, Tomcat, WebSphere, Jetty, etc.

Note You can use either command to create a portable and executable application. The only difference is that when you use the `war` command, it creates a “transportable” WAR, which means that you can run your application as a standalone or you can deploy it to a J2EE-compliant container. You are going to see a complete example in the following chapters.

The install Command

The `install` command is very similar to the `grab` command; the only difference is that you need to specify the library you want to install (in a coordinate format `groupId:artifactId:version`; the same as the `@Grab` annotation). It downloads it and the dependencies in a `lib` directory. Its syntax is as follows.

```
spring install [options] <coordinates>
```

The available options are shown in Table A-6.

Table A-6. *Install Options*

Option	Description
<code>--autoconfigure</code> [Boolean]	Adds the auto-configuration compiler transformation (default is true).
<code>--classpath,</code> <code>-cp</code>	Adds the classpath entries, which is useful when you have third-party libraries. As a recommendation, you can create a <code>lib/</code> folder in the root of your program and add all the classes or JARs there.
<code>--no-guess-dependencies</code>	Does not attempt to guess the dependencies. This is useful when you already use the <code>@Grab</code> annotation in your application.
<code>--no-guess-imports</code>	Does not attempt to guess the imports. This is useful when you already include some of the imports in your groovy application. For example, in a Java app you can use this option because you are importing the classes you need. For more information, see Chapter 3 (the auto-configuration section).

Take for example Listing A-4 (spock.groovy). If you execute the following command.

```
$ spring install org.spockframework:spock-core:1.0-groovy-2.4
```

You have in the `lib` directory the Spock library and its dependencies.

Note If you are using the SDKMAN tool (<http://sdkman.io/>), it downloads the libraries in the `$HOME/.sdkman/candidates/springboot/1.3.X.RELEASE/lib` directory.

The uninstall Command

The `uninstall` command uninstalls the dependencies from the `lib` directory. Its syntax is as follows.

```
spring uninstall [options] <coordinates>
```

The available options are shown in Table A-7.

Table A-7. *Uninstall Options*

Option	Description
<code>--autoconfigure</code> [Boolean]	Adds the auto-configuration compiler transformation (default is true).
<code>--classpath,</code> <code>-cp</code>	Adds the classpath entries, which is useful when you have third-party libraries. As a recommendation, you can create a <code>lib/</code> folder in the root of your program and add all the classes or JARs there.
<code>--no-guess-</code> <code>dependencies</code>	Does not attempt to guess the dependencies. This is useful when you already use the <code>@Grab</code> annotation in your application.
<code>--no-guess-</code> <code>imports</code>	Does not attempt to guess the imports. This is useful when you already include some of the imports in your groovy application. For example, in a Java app you can use this option because you are importing the classes you need. For more information, see Chapter 3 (the auto-configuration section).

You can test this command by executing the following command.

```
$ spring uninstall org.spockframework:spock-core:1.0-groovy-2.4
```

It removes all the Spock dependencies from the lib directory.

The init Command

The `init` command helps you initialize a new project by using the Spring Initializr (<http://start.spring.io/>). Whether or not you are using an IDE, this command helps you get everything ready to start developing Spring Boot applications. Its syntax is as follows.

```
spring init [options] [location]
```

The available options are shown in Table A-8.

Table A-8. *Init Options*

Option	Description
-a, --artifactId	The project coordinate; if it's not provided, the default name is demo.
-b, --boot-version	The Spring Boot version; if it's not provided, it gets the latest, defined as the parent-pom.
--build	The build system to use; the possible values are maven or gradle. If it's not specified, the default value is maven.
-d, --dependencies	A comma-separated list of dependency identifiers are included. For example, -d=web or -d=web,jdbc,actuator.
--description	The project description.
-f, --force	Overwrites existing files.
--format	A format of the generated content. Useful when you want to import your projects into an IDE like STS. The possible values are build and project. If it's not provided, the default value is project.
-g, --groupId	The project coordinates defining the group ID. If it's not provided, it defaults to com.example.

(continued)

Table A-8. (continued)

Option	Description
-j, --java-version	The language level. If it's not provided, it defaults to 1.8.
-l, --language	Specifies the programming language. The possible values are java and groovy. If it's not provided, it defaults to java.
-n, --name	The name of the application. If it's not provided, it defaults to demo.
-p, --packaging	The project packaging. The values are jar, war, and zip. If it's not provided, it generates a ZIP file.
--package-name	The package name. If it's not provided, it defaults to demo.
-t, --type	The project type. The values are maven-project, maven-build, gradle-project, and gradle-build. If it's not provided, it defaults to maven-project.
--target	The URL of the service to use. It defaults to https://start.spring.io . This means that you can create your own reference service.
-v, --version	The project version. If it's not provided, it defaults to 0.0.1-SNAPSHOT.
-x, --extract	Extracts the content of the project created in the current directory if the location is not specified.

You use this command very often (if you are not using an IDE such as the STS or IntelliJ), so you can get used to it with the following examples.

To create a default project, you just execute

```
$ spring init
```

It generates a demo.zip file. You can unzip it and take a look at the structure (a Maven project structure), as shown in Figure A-1, but the most important part is the pom.xml file. If you look at this file, you can see the minimal dependencies: spring-boot-starter and spring-boot-starter-test.

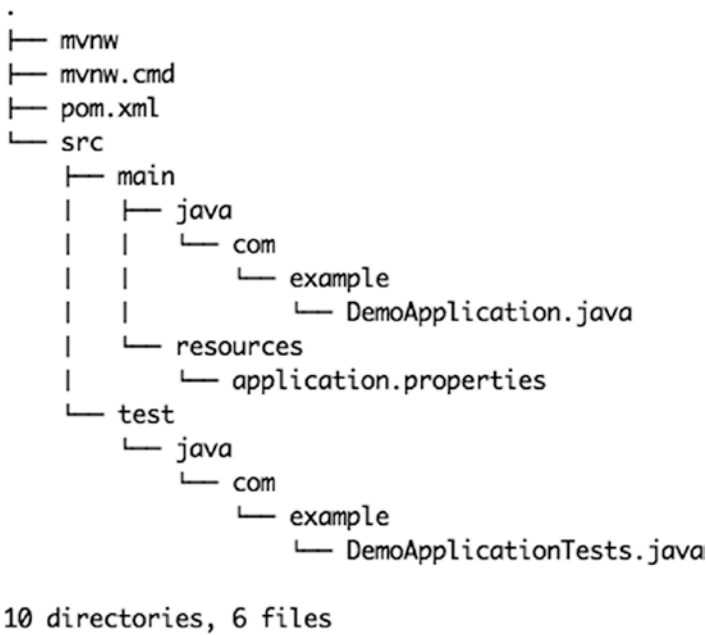


Figure A-1. *The demo.zip contents*

Figure A-1 shows the demo.zip structure. Take a look at the src folder, which contains the main/java/com/example/DemoApplication.java file and of course its unit test. Also you can see that it contains two additional files, mvnw (for UNIX) and mvnw.cmd (for Windows). These commands allow you to run a Maven project without actually having Maven installed on your system.

You can simply execute the following command.

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

This command downloads the Maven tool (in the .mvn subdirectory) and run it. If you take a look at the DemoApplication.java class, you'll see that it's not doing much. It's simply running the Spring Boot application. With all this you have a template that you can use over and over. If you want to create a web application, the only thing you need to do is add the spring-boot-starter-web dependency.

init Examples

This section includes more examples using the init command. The following command creates a web application with JDBC Gradle project.

```
$ spring init -d=web,jdbc --build=gradle
```

This command generates a `demo.zip` file, but with its contents using Gradle. It includes the Gradle wrapper so you don't have to install it.

If you want to generate only the `pom.xml` (for a Maven project) or `build.gradle` file (for a Gradle project), just add `--format=build` and `--build=[gradle|maven]`.

```
$ spring init -d=web,data-jpa,security --format=build --build=gradle
```

This command creates the `build.gradle` file with the web, JPA, and security dependencies.

```
$ spring init -d=jdbc,amqp --format=build
```

This command creates the `pom.xml` file. If you don't add the `--build` parameter, it defaults to Maven.

To create a project with the name, `groupId` and `artifactId`, you need to use the `-name`, `-g`, and `-a` parameters respectively.

```
$ spring init -d=amqp -g=com.apress.spring -a=spring-boot-rabbitmq
-name=spring-boot-rabbitmq
```

This command creates a `spring-boot-rabbitmq.zip` file (Maven project) with the `groupId` and `artifactId` specified.

By default, when the package name is not specified, it defaults to `com.example`. If you want to add a package convention, you need to add the `--package` parameter.

```
$ spring init -d=web,thymeleaf,data-jpa,data-rest,security -g=com.apress.
spring -a=spring-boot-journal-oauth --package-name=com.apress.spring
-name=spring-boot-journal-oauth
```

It's fine to have a ZIP file for portability, but you can uncompress directly into the current directory. You simply add the `-x` parameter.

```
$ spring init -d=web,thymeleaf,data-jpa,data-
rest,security,actuator,h2,mysql -g=com.apress.spring -a=spring-boot-
journal-cloud --package-name=com.apress.spring -name=spring-boot-journal-
cloud -x
```

This command uncompresses the ZIP file on the fly and the contents are written to the current directory.

If you are curious and want to know more about the dependencies or other parameter values, you can execute the following command.

```
$ spring init --list
```

You use the `spring init` command throughout the entire book, so take a moment and review all its options.

An Alternative to the `init` Command

There will be times when you need just the `pom.xml` or `build.gradle` files, perhaps to check out the dependencies and declarations or to look at the plugins declarations. You execute the following command:

```
$ curl -s https://start.spring.io/pom.xml -d packaging=war -o pom.xml
```

Yes, you read it right! Remember that the `init` command calls the Spring Initializr service at <https://start.spring.io>, so you can use the UNIX `cURL` command. This command generates only the `pom.xml` file. And if you are curious again to see what else you can do by using the UNIX `cURL` command, just execute the following.

```
$ curl start.spring.io
```

This command prints all the available options and some examples using `cURL` with the Spring Initializr. You learned in previous chapters that, within the STS (Spring Tool Suite) IDE, you can create a Spring Boot application by selecting Spring Starter Project. This wizard connects to the Spring Initializr, so either you use an IDE or the command line to get a Spring Boot project structure.

The `shell` Command

The `shell` command starts an embedded shell. Execute the following command.

```
$ spring shell
Spring Boot (v1.3.X.RELEASE)
Hit TAB to complete. Type 'help' and hit RETURN for help, and 'exit' to
quit.
$
```

As you can see from the output, you can type `help` to get more information about the shell. Actually the `spring shell` is the previous command, but just executed in an embedded shell. One of the benefits is that it has a TAB completion so you can get all the possible suggestions for the options.

The help Command

The help command is your best friend. You can execute it as follows.

```
spring help
```

```
usage: spring [--help] [--version]
       <command> [<args>]
```

Available commands are.

```
run [options] <files> [--] [args]
```

Run a spring groovy script

```
test [options] <files> [--] [args]
```

Run a spring groovy script test

```
grab
```

Download a spring groovy script's dependencies to `./repository`

```
jar [options] <jar-name> <files>
```

Create a self-contained executable jar file from a Spring Groovy script

```
war [options] <war-name> <files>
```

Create a self-contained executable war file from a Spring Groovy script

```
install [options] <coordinates>
```

Install dependencies to the lib directory

```
uninstall [options] <coordinates>
```

Uninstall dependencies from the lib directory

```
init [options] [location]
```

Initialize a new project using Spring Initializr (`start.spring.io`)

```
shell
```

Start a nested shell

Common options.

`-d, --debug` Verbose mode

Print additional status information for the command you are running

See `'spring help <command>'` for more information on a specific command.

As you can see from this output, you can also execute the `spring help <command>`, which is very handy because you get more information about the command ,and some examples of how to use it. For example, if you want to know about the `init` command, just execute the following.

```
$ spring help init
```

Remember, the `spring help` command is your best friend.

Summary

The chapter showed you how to use the Spring Boot Command Line Interface. It explained all the different commands and their options.

You learned mentioned that one of the most important commands is the `init` command and it is used in the entire book, either through a terminal in a command line or by using an IDE such as STS or IntelliJ.

Index

A

Abstract syntax tree (AST), 46

/actuator endpoint, 325–326

Advance Message Queuing Protocol
(AMQP), 282

bindings, 283–284

default exchange, 284

direct exchange, 284

exchanges, 283

fanout exchange, 284

headers exchange, 284

queues, 283–284

topic exchange, 284

ApplicationRunner and

 CommandLineRunner

 application configuration, 71, 73

 PropertyResolver interface, 72

 @PropertySource

 annotation, 71

 configuration properties

 application.properties/YAML
 file, 78

 browser, 73

 command-line arguments, 74

 contents of, 80

 location and name, 78

 overriding, 73

 profiles, 79

 relaxed binding, 77

 properties prefix, 81

 app class, 84

 application.properties file, 81

 build.gradle file, 81

 myapp properties, 84

 pom.xml file, 81

 SpringBootConfigApplication.
 java, 82–83

 run method, 68

 SpringBootSimpleApplication.java, 70

Asynchronous sequence

 collectList, 184

 doOnNext, 184

 EmitterProcessor, 183

 FluxExample class, 182

 Flux<T>, 181

 messaging-passing processor, 183

 MonoExample class, 179–180

 Mono<T>, 178

/autoconfig endpoint, 326–327

Auto-configuration

 app.groovy, 45–46, 48

 AutoConfigurationImportSelector
 class, 50

 CloudAutoConfiguration class, 52

 debug parameter, 47

 DemoApplication.java, 49

 @EnableAutoConfiguration, 48, 50

 getCandidateConfigurations method, 50

 modification, 46–47

B

/beans endpoint, [327–328](#)

Boot, [31](#)

app.groovy, [33](#)

application model

browser, [37](#)

build.gradle file, [40, 41](#)

components, [36](#)

DemoApplication.java class, [41](#)

features, [43](#)

Maven pom.xml, [38](#)

technology, [42](#)

web controller, [41](#)

web service, [37](#)

CLI, [34](#)

features of, [31](#)

home page, [32](#)

project webpage, [33](#)

SimpleWebApp.java, [34](#)

web-reactive modules, [31](#)

Boot web applications, [87](#)

MVC technology

auto-configuration, [88](#)

error handling, [90](#)

features of, [88](#)

HttpMessageConverters, [89](#)

JSON, [89](#)

path matching and content

negotiation, [89](#)

spring-web module, [87](#)

template engine, [90](#)

override, [110](#)

Jetty server, [115](#)

JSON date format, [112](#)

JSON/XML content-type, [112](#)

MVC, [114](#)

server, [110](#)

Tomcat, [114](#)

ToDo app (*see* [ToDo app](#))

C

CassandraHealthIndicator, [354](#)

Cloud computing, [433](#)

application.properties file, [437](#)

build.gradle file, [437](#)

Cloud Foundry, [438](#)

cleaning up, [453](#)

command-line tool, [444](#)

deployment, [445](#)

log in (PWS/PAS/CLI), [444](#)

PAS, [439](#)

PWS, [441](#)

service creation, [448](#)

2.x, [439](#)

cloud-native architecture, [433–434](#)

microservices, [436](#)

pom.xml file, [437](#)

twelve-factor applications, [434](#)

Cloud Stream

application, [408](#)

application.properties file, [410](#)

application starters, [430](#)

binder

Kafka, [409](#)

RabbitMQ, [409](#)

build.gradle file, [406](#)

components of, [407](#)

definition, [405](#)

features of, [407](#)

microservices, [429](#)

models, [411](#)

pom.xml file, [405](#)

- processor, 410
 - exchanges tab, 423
 - get messages, 426
 - output, 426
 - publish message, 425
 - queues, 424
 - RabbitMQ web management, 423
 - sink, 428
 - stream application, 422
 - ToDoProcessor class, 421
 - ToDoSender class, 427
 - programming, 409
 - Sink application, 411
 - source, 410
 - build.gradle file, 415
 - pom.xml file, 415
 - RabbitMQ, 415
 - stream application, 414
 - ToDoSource class, 414
 - Spring Initializr, 412–413
 - SubscribableChannel and
 - MessageChannel classes, 410
 - Command-line arguments (CLA), 74
 - Command-line interface
 - (CLI), 34, 444
 - app.groovy, 482
 - UNIX cURL command, 498
 - grab command, 488
 - help command, 499–500
 - init command, 494
 - build.gradle file, 497
 - demo.zip file, 495
 - JDBC Gradle project, 496
 - options and description, 494–495
 - Spring-boot-starter-web
 - dependency, 496
 - ZIP file, 497
 - install command, 492
 - jar command, 489–490
 - MyTest.java, 487
 - run command, 483–484
 - SDKMAN tool, 481
 - shell command, 498
 - test command
 - JUnit and Spock frameworks, 485
 - option and description, 485
 - OutputCapture class, 486–487
 - SimpleWebTest class, 488
 - Spock unit test, 486
 - uninstall command, 493–494
 - war command, 491–492
 - WebApp.java, 482
 - /configprops endpoint, 328–329
 - Cross-origin resource sharing
 - (CORS), 341
 - Custom actuator endpoint, 343
 - ToDo app
 - ToDoRepository interface, 346
 - ToDoStatsEndpoint, 343, 345
- ## D
- Data definition language (DDL), 148
 - @DataJpaTest, 214
 - @DataMongoTest, 216
 - DataSourceHealthIndicator, 354
 - demo.zip structure, 496
 - DiskSpaceHealthIndicator, 354
 - Docker-compose.yml, 367
 - /dump endpoint, 329–330
- ## E
- ElasticsearchHealthIndicator, 354
 - @EnableAutoConfiguration
 - annotation, 50

INDEX

- @Enable* feature, 467
 - algorithm enum, 468
 - BCryptEncoder and Pbkdf2Encoder classes, 469–470
 - EnableToDoSecurity feature, 467
 - ToDoSecurity class, 470–471
 - ToDoSecurityConfiguration class, 468
- @Enable<Technology> annotations, 50
- EndpointHandlerMapping, 341
- /env endpoint, 330–331
- eXtensible Markup Language (XML)
 - configuration file, 394
 - graph panel, 393
 - todo-context.xml, 392
 - ToDoIntegration class, 394

F

- Features, Spring Boot, 54
 - application arguments
 - executable JAR, 68
 - SpringApplication.run, 66
 - ApplicationEvent events, 65
 - banner
 - ASCII art text, 60, 61
 - banner.txt, 61, 62
 - classpath location, 62
 - SpringBootSimple
 - Application.java, 58
 - components, 57
 - Maven/Gradle project, 54–55
 - SpringApplicationBuilder, 63
 - SpringApplication class, 57
 - SpringBootSimpleApplication.java, 56
 - terminal and execute, 55–56
- File integration
 - application.properties, 403
 - integration flow, 401

- list.txt file, 403
- service activator method, 404
- ToDoConverter class, 399
- ToDoFileIntegration class, 400
- ToDoIntegration class, 403
- ToDoMessageHandler class, 399
- ToDoProperties class, 398

G

- Grafana, 363, 367, 378
- grafana.ini file, 368

H

- HandlerInterceptor interface, 364
- /health endpoint, 331–332, 353
 - indicators, 354–357
 - ToDo app, 358–362
 - code, 362
 - values, 353
- HealthIndicator interface, 354
- Hypermedia as the Engine of Application State, HAL +JSON (HATEOAS), 150

I

- InfluxDBHealthIndicator, 354
- /info endpoint, 333
- Integration
 - annotations, 395
 - components, 384
 - definition, 384
 - developers or architects, 383
 - file (*see* File integration)
 - IntegrationFlow, 390
 - JavaConfig, 397

- message
 - endpoints, 385
 - message channel, 385
 - spring-messaging module, 384
- Spring Initializr, 386–387
- ToDo class, 388
- ToDoConfig class, 391
- ToDoIntegration class, 389
- withPayload method, 392
- XML (*see* eXtensible Markup Language (XML))

J, K

- JavaConfig, 26, 397
- Java Database Connectivity (JDBC), 241
 - directory app
 - application.properties file, 248
 - cURL command, 249
 - data.sql file, 248
 - DirectorySecurityConfig class, 245
 - Person class, 242–244
 - PersonRepository interface, 244
 - Spring Initializr, 241–242
 - userDetailsService method, 246
 - JdbcTemplate class, 129
 - methods, 129
 - Spring Boot, 130
 - ToDo app, 131
 - application.properties file, 256
 - CommonRepository interface, 133–135
 - H2 console, 137–139
 - Person class, 250
 - schema.sql/data.sql files, 136
 - terminal window, 256
 - testing, 136

- ToDoProperties class, 251
- ToDoSecurityConfig class, 251–253, 255
- Java Message Service (JMS)
 - ActiveMQ, 282
 - configuration, 274
 - application.properties file, 280
 - producer and consumer, 275–276
 - running, 280
 - ToDoConfig class, 277
 - ToDoErrorHandler code, 279
 - ToDoProperties class, 279
 - ToDoValidator class, 278
- definition, 270
- Pub/Sub pattern, 281
- ToDo app, 270
 - build.gradle file, 272
 - consumer class, 273–274
 - pom.xml file, 272
 - Spring Initializr, 270–271
 - ToDoProducer class, 272
- Java Persistence API (JPA)
 - features, 139
 - Spring Boot, 140–141
 - ToDo app, 141
 - domain model, 143, 145
 - properties, 148–149
 - testing, 149
 - ToDoController class, 146–148
 - ToDoRepository interface, 142–143
- @JdbcTest, 215
- JmsHealthIndicator, 354
- @JsonTest annotation, 212

L

- /loggers endpoint, 333

M

- MailHealthIndicator, [354](#)
- MappedInterceptor bean, [365](#)
- /mappings endpoint, [336–337](#)
- Map-reduce/lower-level
 - operations, [164](#)
- Messaging
 - definition, [269](#)
 - JMS (*see* Java Message Service (JMS))
 - RabbitMQ (*see* RabbitMQ)
 - Redis (*see* Redis)
 - WebSockets (*see* WebSockets)
- MeterRegistry instance, [365](#)
- /metrics endpoint, [334–336](#)
- Micrometer, [363](#)
- Microservices, [436](#)
- Model-View-Controller (MVC), [87](#)
- MongoDB
 - embedded, [160–161](#)
 - features, [158](#)
 - installation, [159](#)
 - Spring Boot, [159](#)
 - ToDo app, [161](#)
 - domain model class, [163–164](#)
 - testing, [164](#)
- MongoDB reactive streams,
 - ToDo app, [196](#)
- browser, [196](#)
- build.gradle file, [198](#)
- command execution, [205](#)
- configuration, [204](#)
- domain class, [198](#)
- handler class, [200–202](#)
- pom.xml file, [198](#)
- ToDoConfig class, [202](#)
- ToDoRepository interface, [199](#)
- ToDoRouter class, [200](#)

- MongoHealthIndicator, [354](#)
- Monolithic *vs.* microservices, [436](#)
- Multiple data sources, [170](#)

N

- NamedParameterJdbcTemplate, [135](#)
- Neo4jHealthIndicator, [354](#)

O

- OAuth2 and ToDo app
 - application.properties file, [260](#)
 - build.gradle file, [259](#)
 - GitHub, [261](#)
 - application.properties, [263](#)
 - authorization process, [265–267](#)
 - client id and secret keys, [263](#)
 - new application, [262](#)
 - pom.xml file, [258](#)
 - Spring Initializr, [257–258](#)
 - ToDoRepository interface, [259](#)
 - ToDoRepository.java, [260](#)

P, Q

- Pivotal Application Service (PAS), [439](#)
- Pivotal Cloud Foundry, [438](#)
- Pivotal Web Services (PWS), [441](#)
- Prometheus, [363, 367](#)
- prometheus.yml file, [368](#)

R

- RabbitHealthIndicator, [354](#)
- RabbitMQ, [283, 415](#)
 - AMQP (*see* Advance Message Queuing Protocol (AMQP))
 - application.properties file, [292](#)

- bindings section, [418](#)
 - build.gradle file, [421](#)
 - configuration, [289](#)
 - consumer class, [288](#)
 - Consumer.java class, [289](#)
 - exchanges tab, [416](#)
 - get messages, [419–420](#)
 - installation, [283](#)
 - my-queue creation, [417](#)
 - overview panel, [418–419](#)
 - pom.xml file, [421](#)
 - Producer.java class, [286–287](#)
 - remote connection, [297](#)
 - sender class, [292](#)
 - ToDo app
 - Spring Initializr, [285–286](#)
 - ToDoSender class, [295, 297](#)
 - web console management, [293](#)
 - web console Queues tab, [295](#)
- Reactive systems, [173](#)
- characteristics, [173](#)
 - data MongoDB, [196](#)
 - Manifesto, [173](#)
 - message driven, [174](#)
 - project reactor
 - processors, operators and timers, [175](#)
 - programming paradigm, [175](#)
 - RxJava and project reactor, [196](#)
 - ToDo app
 - browser, [175](#)
 - build.gradle file, [177](#)
 - domain class, [178](#)
 - Flux<T>, [181](#)
 - initializr, [176](#)
 - MonoExample class, [179](#)
 - Mono<T>, [178](#)
 - onComplete/onError code, [178](#)
 - pom.xml, [176–177](#)
 - WebFlux (*see* WebFlux)
 - wrapper types, [196](#)
- Redis
- configuration, [302](#)
 - consumer, [301](#)
 - features, [165](#)
 - installation, [298](#)
 - RedisTemplate class, [165](#)
 - remote connection, [306](#)
 - server up and running, [304–305](#)
 - Spring Boot, [166](#)
 - Spring Initializr, [298–299](#)
 - ToDo app, [166](#)
 - domain model class, [168–169](#)
 - testing, [169](#)
 - ToDoProducer.java class, [300](#)
- @RedisHash annotation, [169](#)
- RedisHealthIndicator, [354](#)
- REST
- features, [150](#)
 - HATEOAS, [150](#)
 - NoSQL databases, [158](#)
 - Spring Boot, [150](#)
 - ToDo app, [151](#)
 - running, [152–153](#)
 - testing, [154–155](#)
 - testing, HAL Browser, [156–158](#)
- @RestClientTest, [217](#)
- RowMapper interface, [129](#)
- S**
- Security
- Boot use of, [220](#)
 - definition, [219](#)
 - features of, [219–220](#)
 - JDBC (*see* Java Database Connectivity (JDBC))

Security (*cont.*)

- OAuth2 (*see* OAuth2 and ToDo app)
- ToDo app
 - application.properties, 223
 - authentication, 226–227
 - build.gradle file, 232
 - dependencies, 232
 - domain class, 222
 - Http basic authentication, 231
 - index.mustache page, 234
 - index page, 238–239
 - login page, 225–226
 - .mustache extension, 232
 - overriding, 227
 - sign in, 238
 - Spring Initializr, 220–221
 - spring.security.* properties, 227
 - SQL statements, 224
 - ToDoRepository interface, 223
 - ToDoSecurityConfig
 - class, 229, 235–236, 240
 - ToDoWebConfig class, 237
 - WebFlux, 257
- /shutdown endpoint, 337, 339
- SolrHealthIndicator, 354
- Spring Actuator module, 319
 - /actuator endpoint, 325–326
 - /autoconfig endpoint, 326–327
 - /beans endpoint, 327
 - beans to run, 328
 - /configprops endpoint, 327–329
 - configuration, 342
 - CORS, 341
 - /dump endpoint, 329–330
 - endpoint ID, 340
 - /env endpoint, 330–331
 - //health endpoint, 331–332, 353
 - /info endpoint, 333
 - /loggers endpoint, 333
 - management, 341
 - /mappings endpoint, 336–337
 - /metrics, 363
 - Docker, 366
 - endpoint, 334–336
 - ToDo app, 363, 369
 - securing, 342
 - /shutdown endpoint, 337, 339
 - ToDo app, 320
 - endpoints, 323–324
 - Spring Initializr, 321
 - ToDoRepository
 - interface, 321–323
 - /trace endpoint, 339–340
- SpringApplicationBuilder class, 63
- Spring Boot
 - JDBC, 130
 - JPA, 140–141
 - MongoDB, 159
 - Redis, 166
 - REST, 150
- spring-boot-starter
 - pom.xml file, 456
 - structure, 455
 - todo-client-spring-boot-starter
 - (*see* todo-client-spring-boot-starter folder)
 - todo-client-starter folder, 456
- Spring-boot-starter-actuator, 323
- Spring data
 - data-driven frameworks, 128
 - features, 128
- Spring Framework, 1
 - context, 3
 - history of, 1
 - principles of, 2
 - version 3, 5

web application (*see* Web application)
 WebFlux module, 4
 Spring Initializr, 321
 Spring Tool Suite (STE), 498
 SQL databases, 127–128

T

Template design pattern, 128

Testing

annotations, 208
 ApplicationContext, 208
 definition, 207
 features of, 207
 framework, 207
 mocking beans, 211
 MockMvc class, 210
 slices, 212
 @DataJpaTest, 214
 @DataMongoTest, 216
 @JdbcTest, 215
 @JsonTest, 212
 @RestClientTest, 217
 @WebFluxTest, 214
 @WebMvcTest, 213
 source code, 210
 spring-boot-starter-test
 dependency, 209
 web endpoints, 210

ToDo app, 90

browser, 91–92
 client
 browser, 116
 domain model, 117
 run and test, 123–124
 TodoClientApplication
 class, 123
 ToDoErrorHandler, 118

ToDoRestClient, 120–122
 ToDoRestClientProperties, 119
 CommonRepository interface, 133–135
 controller
 @Autowired, 102
 @DeleteMapping, 102
 @ExceptionHandler, 103
 @GetMapping, 102
 @PatchMapping, 102
 @PathVariable, 103
 @RequestBody, 103
 @RequestMapping, 102
 ResponseEntity<T> class, 103
 @ResponseStatus, 103
 @RestController, 102
 ToDoController, 99–100, 102
 @Valid, 103
 domain model class, 93, 143, 145,
 163–164, 168–169
 H2 console, 137–139
 MongoDB reactive streams, 196
 properties, 148
 repository
 CommonRepository<T>, 95
 ToDoRepository, 96
 requirements, 90
 run option, 104
 schema.sql/data.sql file, 136
 testing, 107, 136, 149, 164, 169
 command execution, 107
 command-line tool, 106
 data (-d option), 109
 deleteToDo method, 108
 errors and errorMessage, 109
 terminal command, 105
 ToDoBuilder, 94
 ToDoController class, 146–148
 ToDoRepository interface, 142–143

INDEX

ToDo app (*cont.*)

validation

ToDoValidationError, 97

ToDoValidationErrorBuilder, 99

WebFlux

annotated controllers, 191–192

browser, 188

domain class, 189–190

functional endpoints, 193–195

Initializr, 189

ToDoHandler class, 194

ToDoRepository class, 190–191

todo-client-spring-boot-starter folder

pom.xml, 457

structure, 457

testing, 474

run app, 477

task project, 474

todo-client-spring-boot-autoconfigure

application.properties files, 464

auto-configuration, 462

helper classes, 464

pom.xml file, 459–462

spring.factories files, 462

structure, 459

ToDoClient classes, 464

ToDoClientProperties, 464

ToDo domain class, 466

ToDoMetricInterceptor class, 364

methods, 365

ToDo REST API service, 471

data-jpa and data-rest, 471–473

@Json* annotations, 473

ToDoRestConfig, 474

/trace endpoint, 339–340

Twelve-factor applications, 434

U, V

Undertow/Netty server, 115

W, X, Y, Z

Web application

app running, 23

classes, 17

ToDo domain class, 17

ToDoController class, 19

ToDo domain class, 18–19

ToDoRepository interface, 19

WEB-INF/views folder, 22

configuration, 9

components, 13–14

dispatcherServlet-servlet.xml

file, 10, 11, 13

logback.xml file, 16

persistence.xml file, 15

resources/META-INF/sql folder, 15

SQL table creation, 15

web.xml file, 9

creation, 4

dependencies, 5

Java config, 26

Maven, 4

ToDo project, 4

WebClient interface, 186

WebFlux

annotated controllers, 185

application security, 257

auto-configuration, 187

dependencies, 188

ToDo app, 188

features of, 185

functional points, 185

- module, [4](#)
- WebClient, [186](#)
- @WebFluxTest, [214](#)
- @WebMvcTest, [213](#)
- WebSecurityConfigurerAdapter, [342](#)
- WebSockets
 - application.properties file, [314](#)
 - build.gradle file, [309](#)
 - configuration, [312](#)
 - definition, [307](#)
- event handler, [311](#)
- producer class, [310](#)
- running execution, [316](#)
- SockJS and Stomp messages, [317](#)
- Spring Initializr, [307–308](#)
- ToDo and ToDoRepository
 - classes, [308](#)
- ToDoEventHandler.java class, [310](#)
- ToDoProperties class, [311](#)
- web client, [314](#)