Spring IN ACTION

SIXTH EDITION

Craig Walls

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Spring in Action
Sixth Edition
Version 4

For more information on this and other Manning titles go to
manning.com

welcome

Thank you for purchasing the MEAP for *Spring in Action, Sixth Edition*. We've come a long way over the course of the last five editions and I'm thrilled to have a chance to bring the latest and greatest Spring has to offer in this new edition. This book should be a valuable resource regardless of whether you're completely new to Spring or are reading this book to brush up on the newest features.

I've attempted to make this edition follow a hands-on, narrative style; leading you through a journey of building an application, starting with initializing the project and going all the way through to how to ready the application for deployment.

We're releasing the first five chapters to start. These chapters aren't all that different from the previous edition, although there are additional updates coming soon that haven't been worked into these chapters yet. In chapter 1, you'll learn how to kick start your Spring project leveraging the Spring Initializr and Spring Boot, and in chapter 2, we'll build on that foundation by using Spring MVC to develop additional browser-based functionality. Chapter 3 looks at persisting data with JDBC and Spring Data JPA. In chapter 4, we'll look at securing applications with Spring Security. Finally, chapter 5 gets into configuring Spring using configuration properties.

Looking further ahead, we'll expand on this first group of chapters additional persistence options, including Spring Data JDBC and non-relational databases such as MongoDB and Cassandra. In part 2 of the book will see us integrating our application with other applications. In part 3, we'll dig into Spring 6's support for reactive programming and revisit some previously developed components to make them more reactive. Finally, in part 4 we'll talk more about deployment.

We hope to have frequent updates to the book, every few weeks, whether that is new chapters or updates to existing chapters. As you are reading, I invite you to visit the liveBook's Discussion Forum to ask questions and leave comments. Your feedback is truly appreciated and I find it valuable in guiding me as I write it.

—Craig Walls

brief contents

1 Getting started with Spring

This chapter covers

- Spring and Spring Boot essentials
- Initializing a Spring project
- An overview of the Spring landscape

Although the Greek philosopher Heraclitus wasn't well known as a software developer, he seemed to have a good handle on the subject. He has been quoted as saying, "The only constant is change." That statement captures a foundational truth of software development.

The way we develop applications today is different than it was a year ago, 5 years ago, 10 years ago, and certainly 20 years ago, before an initial form of the Spring Framework was introduced in Rod Johnson's book, *Expert One-on-One J2EE Design and Development* (Wrox, 2002, http://mng.bz/oVjy).

Back then, the most common types of applications developed were browser-based web applications, backed by relational databases. While that type of development is still relevant, and Spring is well equipped for those kinds of applications, we're now also interested in developing applications composed of microservices destined for the cloud that persist data in a variety of databases. And a new interest in reactive programming aims to provide greater scalability and improved performance with non-blocking operations.

As software development evolved, the Spring Framework also changed to address modern development concerns, including microservices and reactive programming. Spring also set out to simplify its own development model by introducing Spring Boot.

Whether you're developing a simple database-backed web application or constructing a modern

application built around microservices, Spring is the framework that will help you achieve your goals. This chapter is your first step in a journey through modern application development with Spring.

1.1 What is Spring?

I know you're probably itching to start writing a Spring application, and I assure you that before this chapter ends, you'll have developed a simple one. But first, let me set the stage with a few basic Spring concepts that will help you understand what makes Spring tick.

Any non-trivial application is composed of many components, each responsible for its own piece of the overall application functionality, coordinating with the other application elements to get the job done. When the application is run, those components somehow need to be created and introduced to each other.

At its core, Spring offers a *container*, often referred to as the *Spring application context*, that creates and manages application components. These components, or *beans*, are wired together inside the Spring application context to make a complete application, much like bricks, mortar, timber, nails, plumbing, and wiring are bound together to make a house.

The act of wiring beans together is based on a pattern known as *dependency injection* (DI). Rather than have components create and maintain the lifecycle of other beans that they depend on, a dependency-injected application relies on a separate entity (the container) to create and maintain all components and inject those into the beans that need them. This is done typically through constructor arguments or property accessor methods.

For example, suppose that among an application's many components, there are two that you'll address: an inventory service (for fetching inventory levels) and a product service (for providing basic product information). The product service depends on the inventory service to be able to provide a complete set of information about products. Figure 1.1 illustrates the relationships between these beans and the Spring application context.

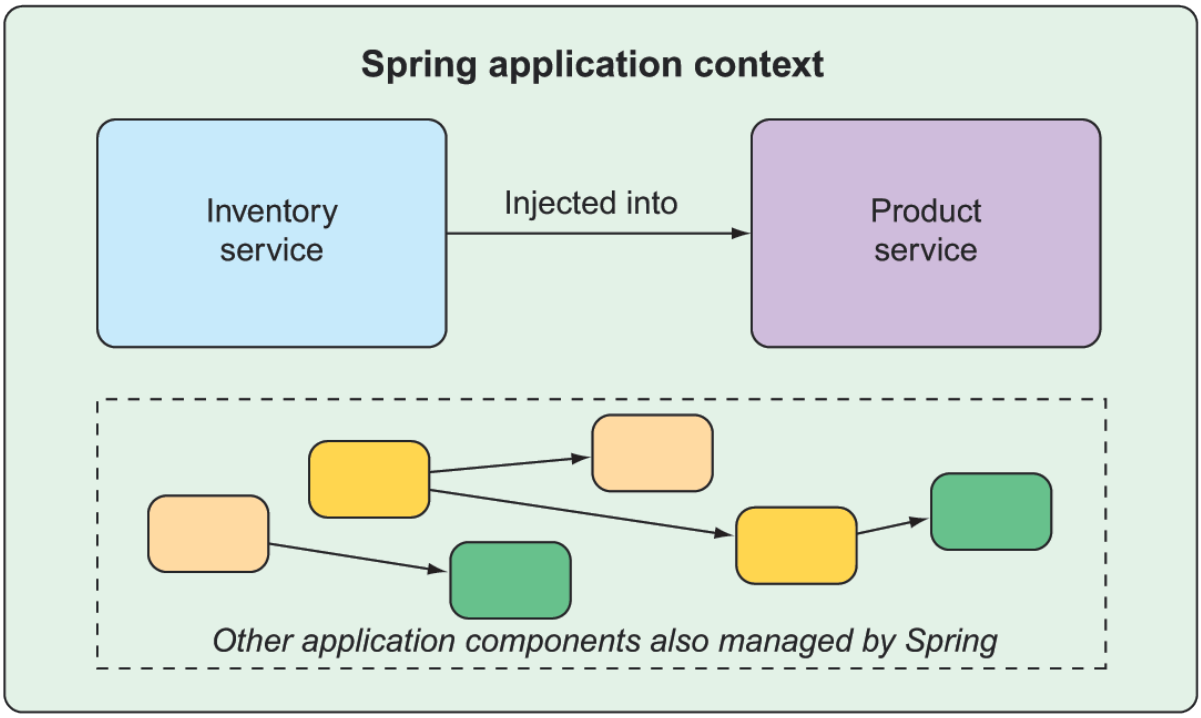

Figure 1.1 Application components are managed and injected into each other by the Spring application context

On top of its core container, Spring and a full portfolio of related libraries offer a web framework, a variety of data persistence options, a security framework, integration with other systems, runtime monitoring, microservice support, a reactive programming model, and many other features necessary for modern application development.

Historically, the way you would guide Spring's application context to wire beans together was with one or more XML files that described the components and their relationship to other components.

For example, the following XML declares two beans, an `InventoryService` bean and a `ProductService` bean, and wires the `InventoryService` bean into `ProductService` via a constructor argument:

```
<bean id="inventoryService"
      class="com.example.InventoryService" />

<bean id="productService"
      class="com.example.ProductService" />
  <constructor-arg ref="inventoryService" />
</bean>
```

In recent versions of Spring, however, a Java-based configuration is more common. The following Java-based configuration class is equivalent to the XML configuration:

```
@Configuration
public class ServiceConfiguration {
  @Bean
  public InventoryService inventoryService() {
```

```
    return new InventoryService();
  }

  @Bean
  public ProductService productService() {
    return new ProductService(inventoryService());
  }
}
```

The `@Configuration` annotation indicates to Spring that this is a configuration class that will provide beans to the Spring application context. The configuration's class methods are annotated with `@Bean`, indicating that the objects they return should be added as beans in the application context (where, by default, their respective bean IDs will be the same as the names of the methods that define them).

Java-based configuration offers several benefits over XML-based configuration, including greater type safety and improved refactorability. Even so, explicit configuration with either Java or XML is only necessary if Spring is unable to automatically configure the components.

Automatic configuration has its roots in the Spring techniques known as *autowiring* and *component scanning*. With component scanning, Spring can automatically discover components from an application's classpath and create them as beans in the Spring application context. With autowiring, Spring automatically injects the components with the other beans that they depend on.

More recently, with the introduction of Spring Boot, automatic configuration has gone well beyond component scanning and autowiring. Spring Boot is an extension of the Spring Framework that offers several productivity enhancements. The most well-known of these enhancements is *autoconfiguration*, where Spring Boot can make reasonable guesses of what components need to be configured and wired together, based on entries in the classpath, environment variables, and other factors.

I'd like to show you some example code that demonstrates autoconfiguration. But I can't. You see, autoconfiguration is much like the wind. You can see the effects of it, but there's no code that I can show you and say "Look! Here's an example of autoconfiguration!" Stuff happens, components are enabled, and functionality is provided without writing code. It's this lack of code that's essential to autoconfiguration and what makes it so wonderful.

Spring Boot autoconfiguration has dramatically reduced the amount of explicit configuration (whether with XML or Java) required to build an application. In fact, by the time you finish the example in this chapter, you'll have a working Spring application that has only a single line of Spring configuration code!

Spring Boot enhances Spring development so much that it's hard to imagine developing Spring applications without it. For that reason, this book treats Spring and Spring Boot as if they were one and the same. We'll use Spring Boot as much as possible, and explicit configuration only

when necessary. And, because Spring XML configuration is the old-school way of working with Spring, we'll focus primarily on Spring's Java-based configuration.

But enough of this chitchat, yakety-yak, and flimflam. This book's title includes the phrase *in action*, so let's get moving, and you can start writing your first application with Spring.

1.2 Initializing a Spring application

Through the course of this book, you'll create Taco Cloud, an online application for ordering the most wonderful food created by man—tacos. Of course, you'll use Spring, Spring Boot, and a variety of related libraries and frameworks to achieve this goal.

You'll find several options for initializing a Spring application. Although I could walk you through the steps of manually creating a project directory structure and defining a build specification, that's wasted time—time better spent writing application code. Therefore, you're going to lean on the Spring Initializr to bootstrap your application.

The Spring Initializr is both a browser-based web application and a REST API, which can produce a skeleton Spring project structure that you can flesh out with whatever functionality you want. Several ways to use Spring Initializr follow:

- From the web application at http://start.spring.io
- From the command line using the `curl` command
- From the command line using the Spring Boot command-line interface
- When creating a new project with Spring Tool Suite
- When creating a new project with IntelliJ IDEA
- When creating a new project with NetBeans

Rather than spend several pages of this chapter talking about each one of these options, I've collected those details in the appendix. In this chapter, and throughout this book, I'll show you how to create a new project using my favorite option: Spring Initializr support in the Spring Tool Suite.

As its name suggests, Spring Tool Suite is a fantastic Spring development environment. But it also offers a handy Spring Boot Dashboard feature that makes it easy to start, restart, and stop Spring Boot applications from the IDE.

If you're not a Spring Tool Suite user, that's fine; we can still be friends. Hop over to the appendix and substitute the Initializr option that suits you best for the instructions in the following sections. But know that throughout this book, I may occasionally reference features specific to Spring Tool Suite, such as the Spring Boot Dashboard. If you're not using Spring Tool Suite, you'll need to adapt those instructions to fit your IDE.

1.2.1 Initializing a Spring project with Spring Tool Suite

To get started with a new Spring project in Spring Tool Suite, go to the File menu and select New, and then Spring Starter Project. Figure 1.2 shows the menu structure to look for.

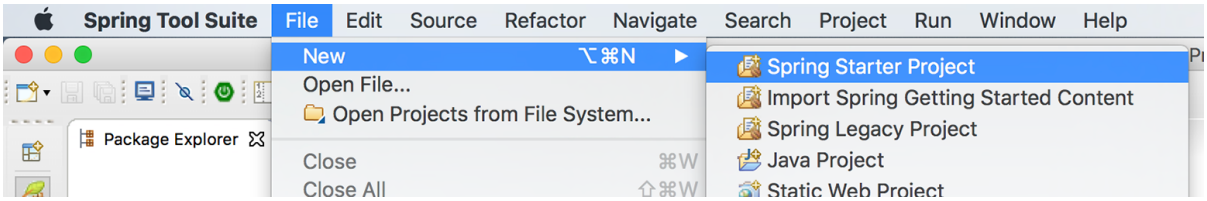

Figure 1.2 Starting a new project with the Initializr in Spring Tool Suite

Once you select Spring Starter Project, a new project wizard dialog (figure 1.3) appears. The first page in the wizard asks you for some general project information, such as the project name, description, and other essential information. If you're familiar with the contents of a Maven pom.xml file, you'll recognize most of the fields as items that end up in a Maven build specification. For the Taco Cloud application, fill in the dialog as shown in figure 1.3, and then click Next.

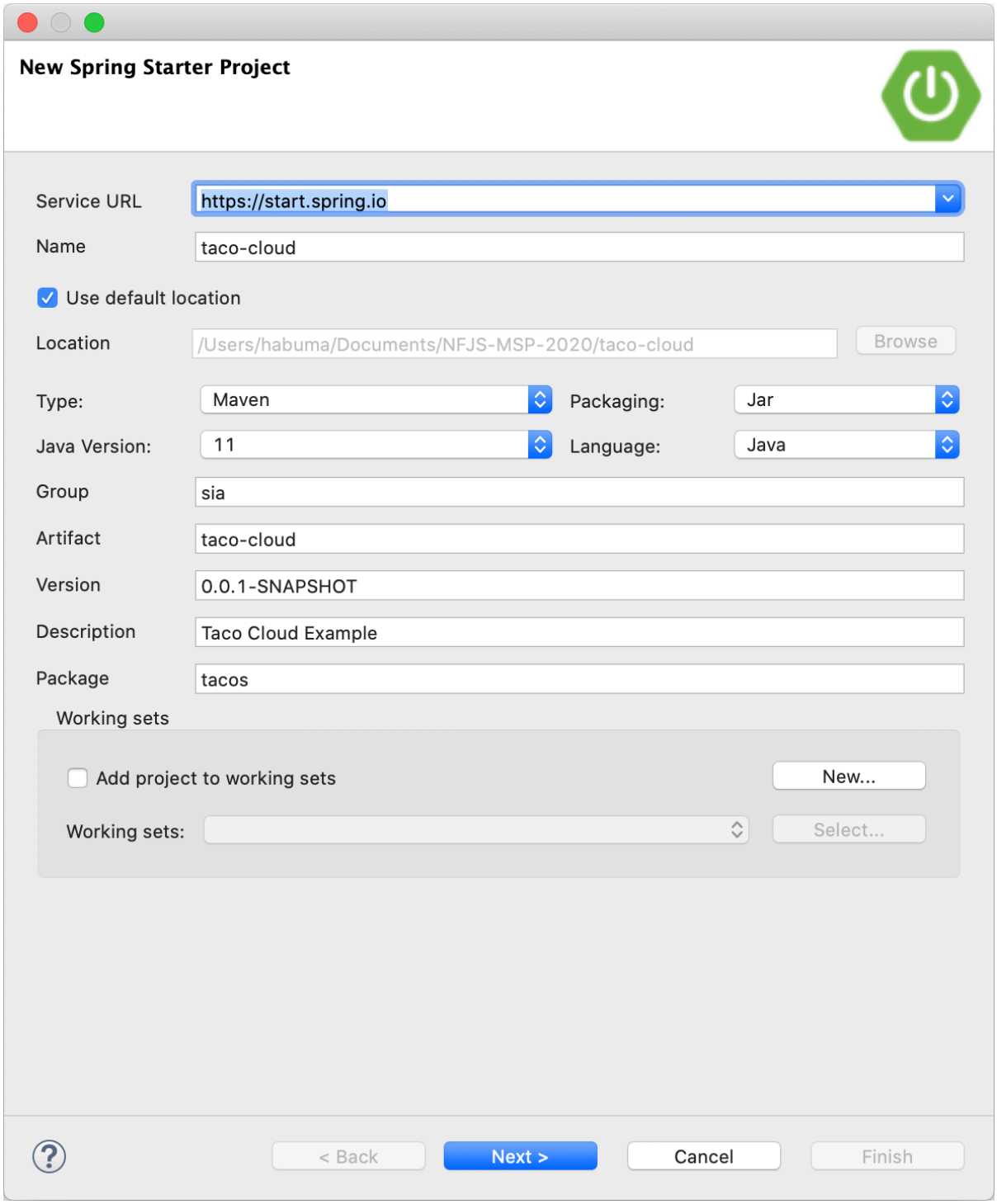

Figure 1.3 Specifying general project information for the Taco Cloud application

The next page in the wizard lets you select dependencies to add to your project (see figure 1.4). Notice that near the top of the dialog, you can select which version of Spring Boot you want to base your project on. This defaults to the most current version available. It's generally a good idea to leave it as is unless you need to target a different version.

As for the dependencies themselves, you can either expand the various sections and seek out the desired dependencies manually, or search for them in the search box at the top of the Available

list. For the Taco Cloud application, you'll start with the dependencies shown in figure 1.4.

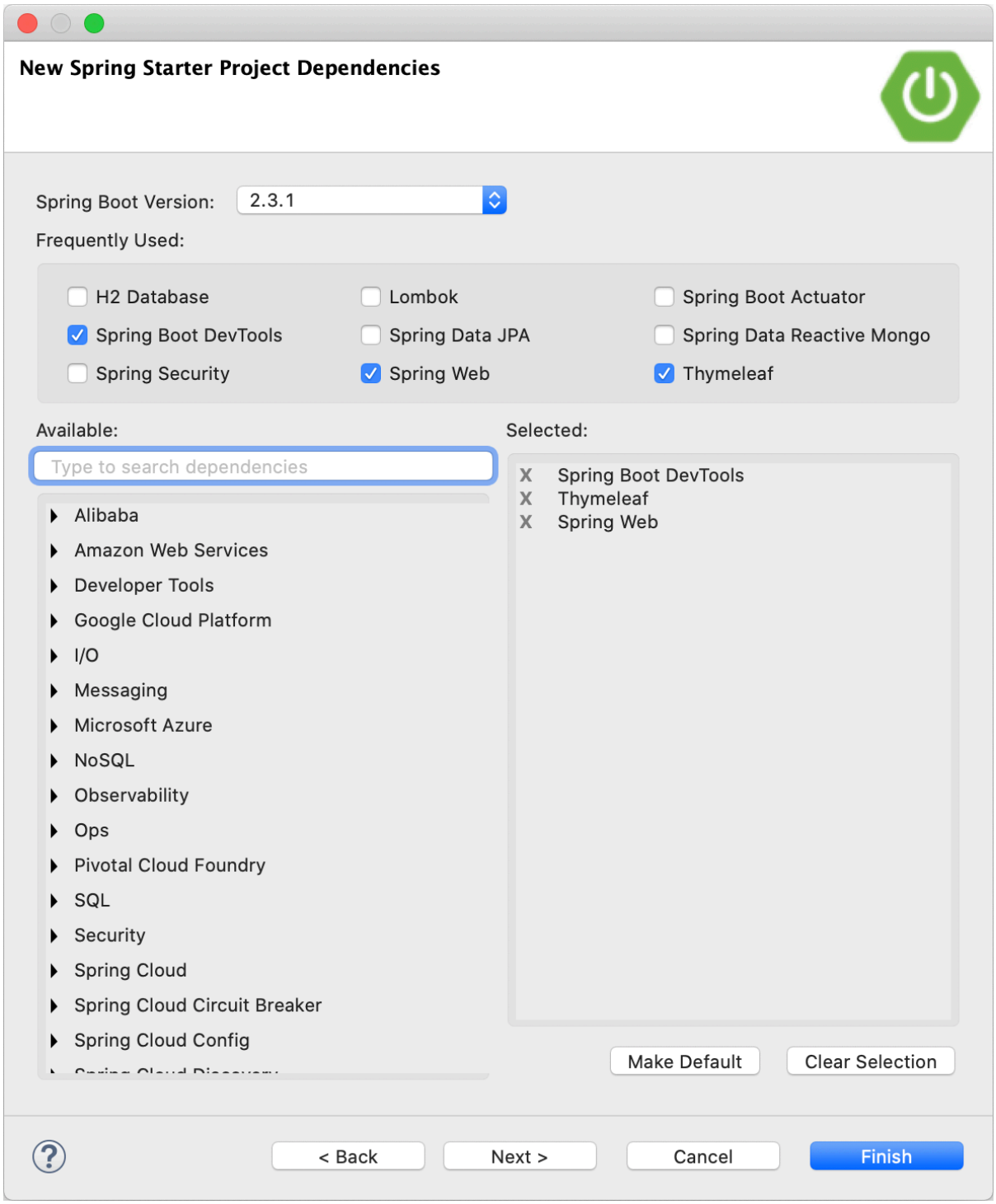

Figure 1.4 Choosing starter dependencies

At this point, you can click Finish to generate the project and add it to your workspace. But if you're feeling slightly adventurous, click Next one more time to see the final page of the new starter project wizard, as shown in figure 1.5.

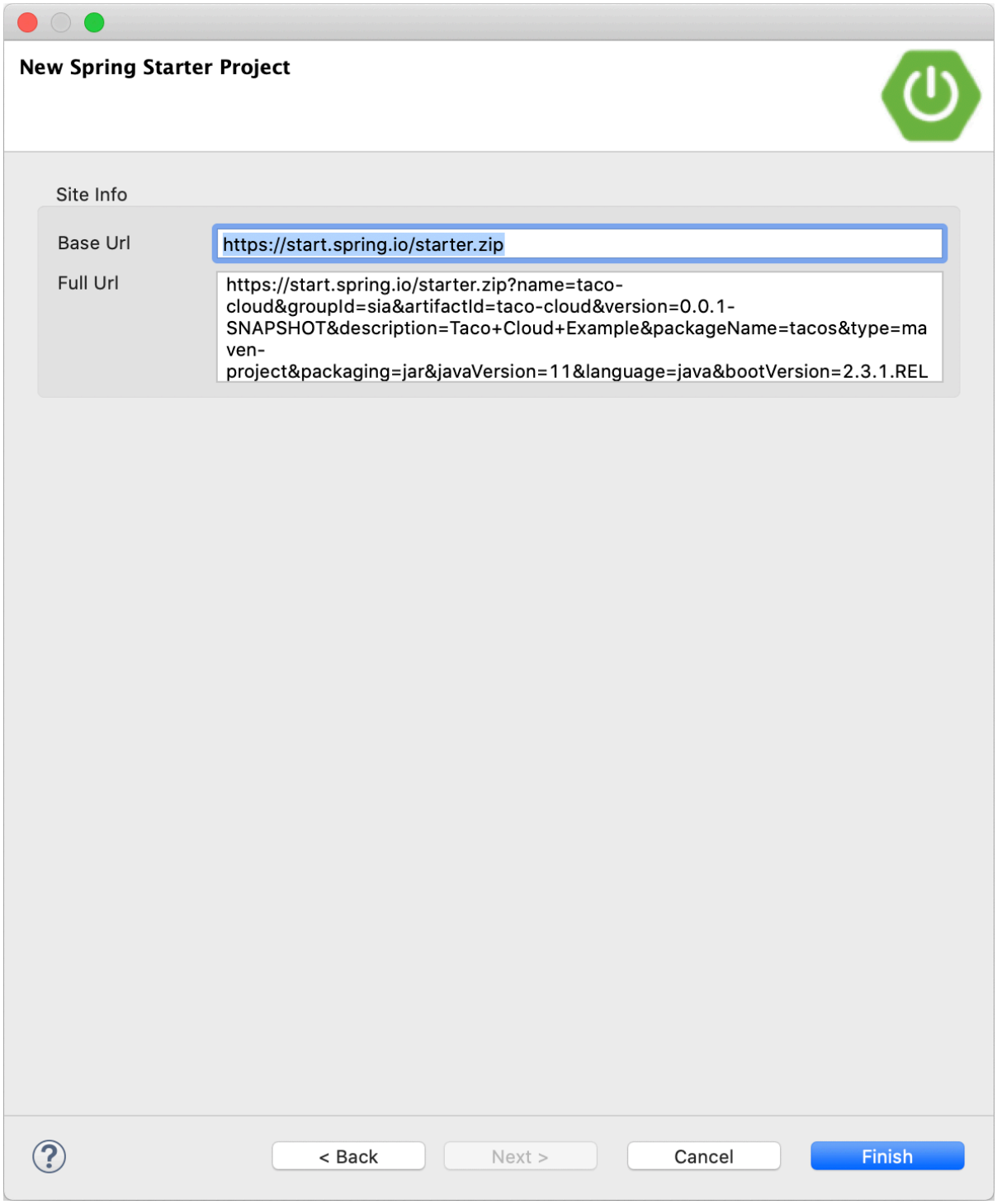

Figure 1.5 Optionally specifying an alternate Initializr address

By default, the new project wizard makes a call to the Spring Initializr at http://start.spring.io to generate the project. Generally, there's no need to override this default, which is why you could have clicked Finish on the second page of the wizard. But if for some reason you're hosting your own clone of Initializr (perhaps a local copy on your own machine or a customized clone running inside your company firewall), then you'll want to change the Base Url field to point to your Initializr instance before clicking Finish.

After you click Finish, the project is downloaded from the Initializr and loaded into your workspace. Wait a few moments for it to load and build, and then you'll be ready to start developing application functionality. But first, let's take a look at what the Initializr gave you.

1.2.2 Examining the Spring project structure

After the project loads in the IDE, expand it to see what it contains. Figure 1.6 shows the expanded Taco Cloud project in Spring Tool Suite.

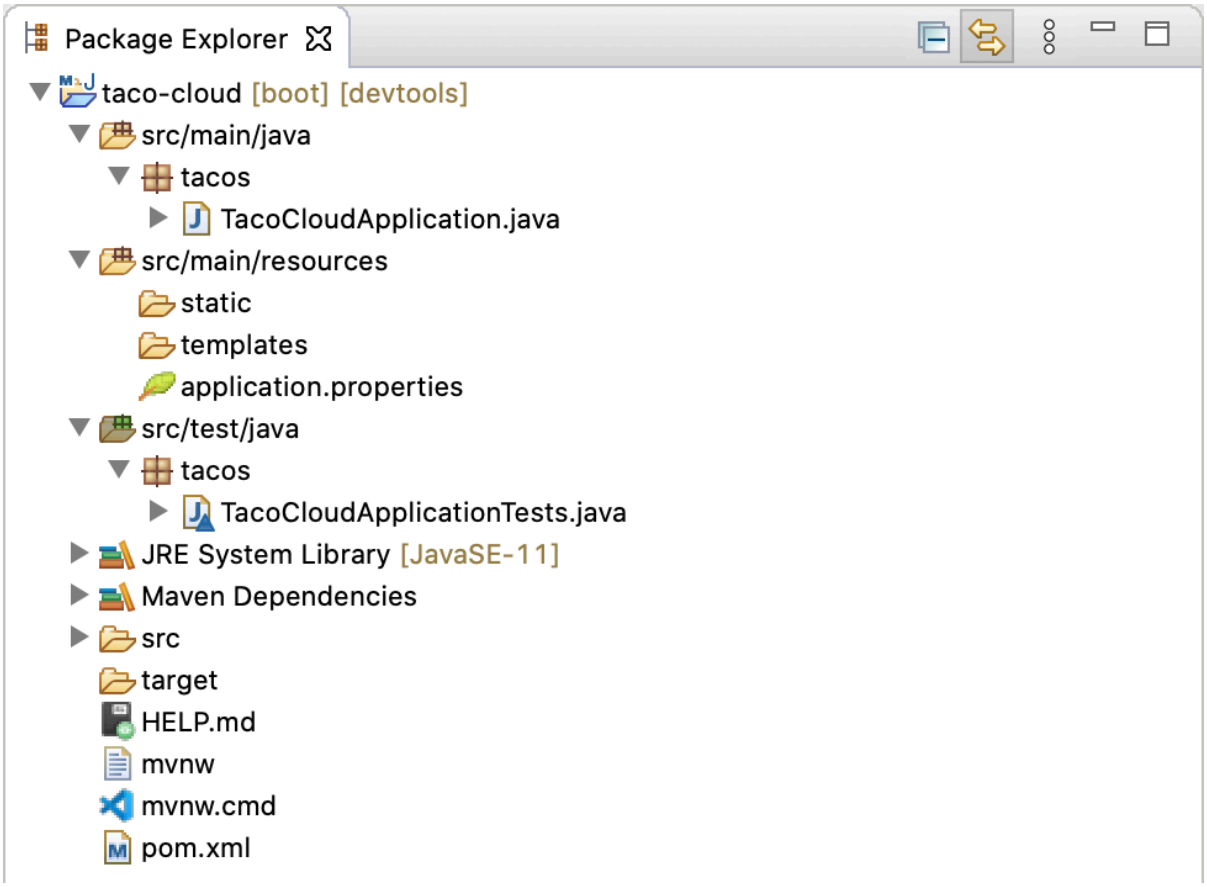

Figure 1.6 The initial Spring project structure as shown in Spring Tool Suite

You may recognize this as a typical Maven or Gradle project structure, where application source code is placed under src/main/java, test code is placed under src/test/java, and non-Java resources are placed under src/main/resources. Within that project structure, you'll want to take note of these items:

- `mvnw` and `mvnw.cmd` — These are Maven wrapper scripts. You can use these scripts to build your project even if you don't have Maven installed on your machine.
- pom.xml — This is the Maven build specification. We'll look deeper into this in a moment.
- `TacoCloudApplication.java` —This is the Spring Boot main class that bootstraps the project. We'll take a closer look at this class in a moment.
- application.properties — This file is initially empty, but offers a place where you can

specify configuration properties. We'll tinker with this file a little in this chapter, but I'll postpone a detailed explanation of configuration properties to chapter 5.

- static — This folder is where you can place any static content (images, stylesheets, JavaScript, and so forth) that you want to serve to the browser. It's initially empty.
- templates — This folder is where you'll place template files that will be used to render content to the browser. It's initially empty, but you'll add a Thymeleaf template soon.
- `TacoCloudApplicationTests.java` — This is a simple test class that ensures that the Spring application context loads successfully. You'll add more tests to the mix as you develop the application.

As the Taco Cloud application grows, you'll fill in this barebones project structure with Java code, images, stylesheets, tests, and other collateral that will make your project more complete. But in the meantime, let's dig a little deeper into a few of the items that Spring Initializr provided.

EXPLORING THE BUILD SPECIFICATION

When you filled out the Initializr form, you specified that your project should be built with Maven. Therefore, the Spring Initializr gave you a pom.xml file already populated with the choices you made. The following listing shows the entire pom.xml file provided by the Initializr.

Listing 1.1 The initial Maven build specification

```
<?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
        https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.4.0</version>     (1)
    <relativePath />
  </parent>
  <groupId>sia</groupId>
  <artifactId>taco-cloud</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>taco-cloud</name>
  <description>Taco Cloud Example</description>

  <properties>
    <java.version>11</java.version>
  </properties>

  <dependencies>
    <dependency>                                                    (2)
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-thymeleaf</artifactId>
    </dependency>

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

    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-devtools</artifactId>
      <scope>runtime</scope>
      <optional>true</optional>
    </dependency>

    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
      <exclusions>
        <exclusion>
          <groupId>org.junit.vintage</groupId>
          <artifactId>junit-vintage-engine</artifactId>
        </exclusion>
      </exclusions>
    </dependency>

  </dependencies>

  <build>
    <plugins>
      <plugin>                                                      (3)
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </build>

  <repositories>
    <repository>
```

```
      <id>spring-milestones</id>
      <name>Spring Milestones</name>
      <url>https://repo.spring.io/milestone</url>
    </repository>
  </repositories>
  <pluginRepositories>
    <pluginRepository>
      <id>spring-milestones</id>
      <name>Spring Milestones</name>
      <url>https://repo.spring.io/milestone</url>
    </pluginRepository>
  </pluginRepositories>

</project>
```

❶ Spring Boot version

❷ Starter dependencies

❸ Spring Boot plugin

The first thing to take note of is the `<parent>` element and, more specifically, its `<version>` child. This specifies that your project has `spring-boot-starter-parent` as its parent POM. Among other things, this parent POM provides dependency management for several libraries commonly used in Spring projects. For those libraries covered by the parent POM, you won't have to specify a version, as it's inherited from the parent. The version, `2.4.0`, indicates that you're using Spring Boot 2.4.0 and, thus, will inherit dependency management as defined by that version of Spring Boot. Among other things, Spring Boot's dependency management for version 2.4.0 specifies that the underlying version of the core Spring Framework will be 5.3.0.

While we're on the subject of dependencies, note that there are four dependencies declared under the `<dependencies>` element. The first three should look somewhat familiar to you. They correspond directly to the `Spring Web`, `Thymeleaf`, and `Spring Boot DevTools` dependencies that you selected before clicking the Finish button in the Spring Tool Suite new project wizard. The other dependency is one that provides a lot of helpful testing capabilities. You didn't have to check a box for it to be included because the Spring Initializr assumes (hopefully, correctly) that you'll be writing tests.

You may also notice that all dependencies except for the DevTools dependency have the word *starter* in their artifact ID. Spring Boot starter dependencies are special in that they typically don't have any library code themselves, but instead transitively pull in other libraries. These starter dependencies offer three primary benefits:

- Your build file will be significantly smaller and easier to manage because you won't need to declare a dependency on every library you might need.
- You're able to think of your dependencies in terms of what capabilities they provide, rather than in terms of library names. If you're developing a web application, you'll add the web starter dependency rather than a laundry list of individual libraries that enable you to write a web application.
- You're freed from the burden of worrying about library versions. You can trust that for a

given version of Spring Boot, the versions of the libraries brought in transitively will be compatible. You only need to worry about which version of Spring Boot you're using.

Finally, the build specification ends with the Spring Boot plugin. This plugin performs a few important functions:

- It provides a Maven goal that enables you to run the application using Maven. You'll try out this goal in section 1.3.4.
- It ensures that all dependency libraries are included within the executable JAR file and available on the runtime classpath.
- It produces a manifest file in the JAR file that denotes the bootstrap class (`TacoCloudApplication`, in your case) as the main class for the executable JAR.

Speaking of the bootstrap class, let's open it up and take a closer look.

BOOTSTRAPPING THE APPLICATION

Because you'll be running the application from an executable JAR, it's important to have a main class that will be executed when that JAR file is run. You'll also need at least a minimal amount of Spring configuration to bootstrap the application. That's what you'll find in the `TacoCloudApplication` class, shown in the following listing.

Listing 1.2 The Taco Cloud bootstrap class

```
package tacos;

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

@SpringBootApplication      ❶
public class TacoCloudApplication {

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

}
```

❶ Spring Boot application

❷ Runs the application

Although there's little code in `TacoCloudApplication`, what's there packs quite a punch. One of the most powerful lines of code is also one of the shortest. The `@SpringBootApplication` annotation clearly signifies that this is a Spring Boot application. But there's more to `@SpringBootApplication` than meets the eye.

`@SpringBootApplication` is a composite annotation that combines three other annotations:

- `@SpringBootConfiguration` — Designates this class as a configuration class. Although there's not much configuration in the class yet, you can add Java-based Spring

Framework configuration to this class if you need to. This annotation is, in fact, a specialized form of the `@Configuration` annotation.

- `@EnableAutoConfiguration` — Enables Spring Boot automatic configuration. We'll talk more about autoconfiguration later. For now, know that this annotation tells Spring Boot to automatically configure any components that it thinks you'll need.
- `@ComponentScan` — Enables component scanning. This lets you declare other classes with annotations like `@Component`, `@Controller`, `@Service`, and others, to have Spring automatically discover them and register them as components in the Spring application context.

The other important piece of `TacoCloudApplication` is the `main()` method. This is the method that will be run when the JAR file is executed. For the most part, this method is boilerplate code; every Spring Boot application you write will have a method similar or identical to this one (class name differences notwithstanding).

The `main()` method calls a static `run()` method on the SpringApplication class, which performs the actual bootstrapping of the application, creating the Spring application context. The two parameters passed to the `run()` method are a configuration class and the command-line arguments. Although it's not necessary that the configuration class passed to `run()` be the same as the bootstrap class, this is the most convenient and typical choice.

Chances are you won't need to change anything in the bootstrap class. For simple applications, you might find it convenient to configure one or two other components in the bootstrap class, but for most applications, you're better off creating a separate configuration class for anything that isn't autoconfigured. You'll define several configuration classes throughout the course of this book, so stay tuned for details.

TESTING THE APPLICATION

Testing is an important part of software development. Recognizing this, the Spring Initializr gives you a test class to get started. The following listing shows the baseline test class.

Listing 1.3 A baseline application test

```
package tacos;

import org.junit.jupiter.api.Test;
import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest                    ❶
public class TacoCloudApplicationTests {

  @Test                            ❷
  public void contextLoads() {
  }

}
```

❶ A Spring Boot test

❷ The test method

There's not much to be seen in `TacoCloudApplicationTests`: the one test method in the class is empty. Even so, this test class does perform an essential check to ensure that the Spring application context can be loaded successfully. If you make any changes that prevent the Spring application context from being created, this test fails, and you can react by fixing the problem.

The `@SpringBootTest` annotation tells JUnit to bootstrap the test with Spring Boot capabilities. Just like `@SpringBootApplication`, `@SpringBootTest` is a composite annotation, which is itself annotated with `@ExtendWith(SpringExtension.class)`, to add Spring testing capabilities to JUnit 5. For now, though, it's enough to think of this as the test class equivalent of calling `SpringApplication.run()` in a `main()` method. Over the course of this book, you'll see `@SpringBootTest` several times, and we'll uncover some of its power.

Finally, there's the test method itself. Although `@SpringBootTest` is tasked with loading the Spring application context for the test, it won't have anything to do if there aren't any test methods. Even without any assertions or code of any kind, this empty test method will prompt the two annotations to do their job and load the Spring application context. If there are any problems in doing so, the test fails.

At this point, we've concluded our review of the code provided by the Spring Initializr. You've seen some of the boilerplate foundation that you can use to develop a Spring application, but you still haven't written a single line of code. Now it's time to fire up your IDE, dust off your keyboard, and add some custom code to the Taco Cloud application.

1.3 Writing a Spring application

Because you're just getting started, we'll start off with a relatively small change to the Taco Cloud application, but one that will demonstrate a lot of Spring's goodness. It seems appropriate that as you're just starting, the first feature you'll add to the Taco Cloud application is a homepage. As you add the homepage, you'll create two code artifacts:

- A controller class that handles requests for the homepage
- A view template that defines what the homepage looks like

And because testing is important, you'll also write a simple test class to test the homepage. But first things first … let's write that controller.

1.3.1 Handling web requests

Spring comes with a powerful web framework known as Spring MVC. At the center of Spring MVC is the concept of a *controller*, a class that handles requests and responds with information of some sort. In the case of a browser-facing application, a controller responds by optionally populating model data and passing the request on to a view to produce HTML that's returned to the browser.

You're going to learn a lot about Spring MVC in chapter 2. But for now, you'll write a simple controller class that handles requests for the root path (for example, `/`) and forwards those requests to the homepage view without populating any model data. The following listing shows the simple controller class.

Listing 1.4 The homepage controller

```
package tacos;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;

@Controller                ❶
public class HomeController {

  @GetMapping("/")         ❷
  public String home() {
    return "home";         ❸
  }

}
```

❶ The controller

❷ Handles requests for the root path /

❸ Returns the view name

As you can see, this class is annotated with `@Controller`. On its own, `@Controller` doesn't do much. Its primary purpose is to identify this class as a component for component scanning. Because `HomeController` is annotated with `@Controller`, Spring's component scanning automatically discovers it and creates an instance of `HomeController` as a bean in the Spring application context.

In fact, a handful of other annotations (including `@Component`, `@Service`, and `@Repository`) serve a purpose similar to `@Controller`. You could have just as effectively annotated `HomeController` with any of those other annotations, and it would have still worked the same. The choice of `@Controller` is, however, more descriptive of this component's role in the application.

The `home()` method is as simple as controller methods come. It's annotated with `@GetMapping` to indicate that if an HTTP GET request is received for the root path /, then this method should

handle that request. It does so by doing nothing more than returning a `String` value of `home`.

This value is interpreted as the logical name of a view. How that view is implemented depends on a few factors, but because Thymeleaf is in your classpath, you can define that template with Thymeleaf.

> **NOTE** **Why Thymeleaf?**
>
> You may be wondering why to chose Thymeleaf for a template engine. Why not JSP? Why not FreeMarker? Why not one of several other options?
>
> Put simply, I had to choose something, and I like Thymeleaf and generally prefer it over those other options. And even though JSP may seem like an obvious choice, there are some challenges to overcome when using JSP with Spring Boot. I didn't want to go down that rabbit hole in chapter 1. Hang tight. We'll look at other template options, including JSP, in chapter 2.

The template name is derived from the logical view name by prefixing it with /templates/ and postfixing it with .html. The resulting path for the template is /templates/home.html. Therefore, you'll need to place the template in your project at /src/main/resources/templates/home.html. Let's create that template now.

1.3.2 Defining the view

In the interest of keeping your homepage simple, it should do nothing more than welcome users to the site. The next listing shows the basic Thymeleaf template that defines the Taco Cloud homepage.

Listing 1.5 The Taco Cloud homepage template

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">
  <head>
    <title>Taco Cloud</title>
  </head>

  <body>
    <h1>Welcome to...</h1>
    <img th:src="@{/images/TacoCloud.png}"/>
  </body>
</html>
```

There's not much to discuss with regard to this template. The only notable line of code is the one with the `<img>` tag to display the Taco Cloud logo. It uses a Thymeleaf `th:src` attribute and an `@{…}` expression to reference the image with a context-relative path. Aside from that, it's not much more than a Hello World page.

Let's talk about that image a bit more. I'll leave it up to you to define a Taco Cloud logo that you

like. But you'll need to make sure you place it at the right place within the project.

The image is referenced with the context-relative path /images/TacoCloud.png. As you'll recall from our review of the project structure, static content such as images is kept in the /src/main/resources/static folder. That means that the Taco Cloud logo image must also reside within the project at /src/main/resources/static/images/ TacoCloud.png.

Now that you've got a controller to handle requests for the homepage and a view template to render the homepage, you're almost ready to fire up the application and see it in action. But first, let's see how you can write a test against the controller.

1.3.3 Testing the controller

Testing web applications can be tricky when making assertions against the content of an HTML page. Fortunately, Spring comes with some powerful test support that makes testing a web application easy.

For the purposes of the homepage, you'll write a test that's comparable in complexity to the homepage itself. Your test will perform an HTTP GET request for the root path / and expect a successful result where the view name is home and the resulting content contains the phrase "Welcome to…". The following should do the trick.

Listing 1.6 A test for the homepage controller

```
package tacos;

import static org.hamcrest.Matchers.containsString;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.content;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.view;

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.test.web.servlet.MockMvc;

@WebMvcTest(HomeController.class)     ❶
public class HomeControllerTest {

  @Autowired
  private MockMvc mockMvc;     ❷

  @Test
  public void testHomePage() throws Exception {
    mockMvc.perform(get("/"))      ❸
      .andExpect(status().isOk())    ❹
      .andExpect(view().name("home"))    ❺
      .andExpect(content().string(            ❻
          containsString("Welcome to...")));
  }

}
```

❶ Web test for HomeController

❷ Injects MockMvc

❸ Performs GET /

❹ Expects HTTP 200

❺ Expects home view

❻ Expects Welcome to…

The first thing you might notice about this test is that it differs slightly from the `TacoCloudApplicationTests` class with regard to the annotations applied to it. Instead of `@SpringBootTest` markup, HomeControllerTest is annotated with `@WebMvcTest`. This is a special test annotation provided by Spring Boot that arranges for the test to run in the context of a Spring MVC application. More specifically, in this case, it arranges for HomeController to be registered in Spring MVC so that you can throw requests against it.

`@WebMvcTest` also sets up Spring support for testing Spring MVC. Although it could be made to start a server, mocking the mechanics of Spring MVC is sufficient for your purposes. The test class is injected with a `MockMvc` object for the test to drive the mockup.

The `testHomePage()` method defines the test you want to perform against the homepage. It starts with the `MockMvc` object to perform an HTTP GET request for / (the root path). From that request, it sets the following expectations:

- The response should have an HTTP 200 (OK) status.
- The view should have a logical name of home.
- The rendered view should contain the text “Welcome to….”

If, after the `MockMvc` object performs the request, any of those expectations aren’t met, then the test fails. But your controller and view template are written to satisfy those expectations, so the test should pass with flying colors—or at least with some shade of green indicating a passing test.

The controller has been written, the view template created, and you have a passing test. It seems that you’ve implemented the homepage successfully. But even though the test passes, there’s something slightly more satisfying with seeing the results in a browser. After all, that’s how Taco Cloud customers are going to see it. Let’s build the application and run it.

1.3.4 Building and running the application

Just as there are several ways to initialize a Spring application, there are several ways to run one. If you like, you can flip over to the appendix to read about some of the more common ways to run a Spring Boot application.

Because you chose to use Spring Tool Suite to initialize and work on the project, you have a

handy feature called the Spring Boot Dashboard available to help you run your application inside the IDE. The Spring Boot Dashboard appears as a tab, typically near the bottom left of the IDE window. Figure 1.7 shows an annotated screenshot of the Spring Boot Dashboard.

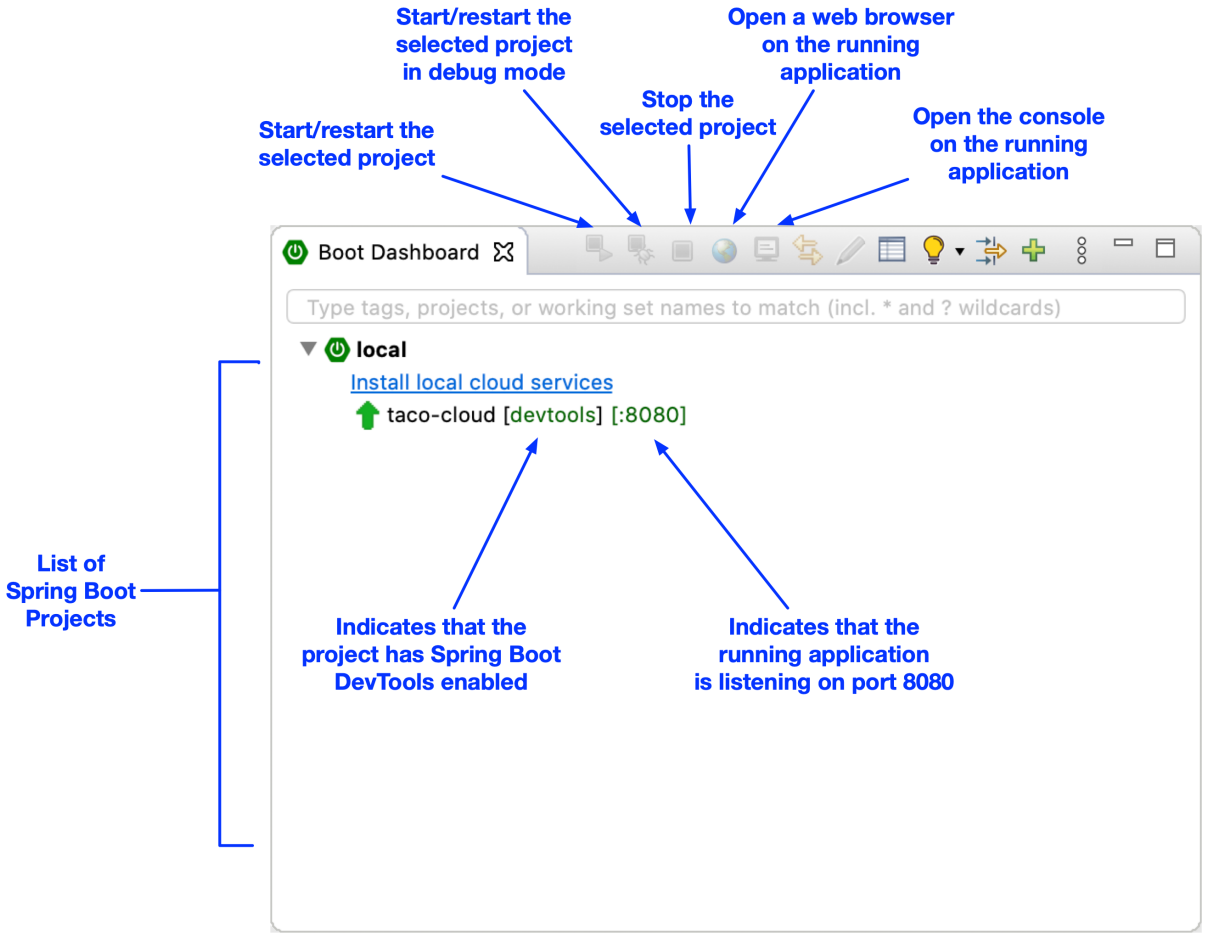

Figure 1.7 Highlights of the Spring Boot Dashboard

I don't want to spend much time going over everything the Spring Boot Dashboard does, although figure 1.7 covers some of the most useful details. The important thing to know right now is how to use it to run the Taco Cloud application. Make sure taco-cloud application is highlighted in the list of projects (it's the only application shown in figure 1.7), and then click the start button (the left-most button with both a green triangle and a red square). The application should start right up.

As the application starts, you'll see some Spring ASCII art fly by in the console, followed by some log entries describing the steps as the application starts. Before the logging stops, you'll see a log entry saying Tomcat started on port(s): 8080 (http), which means that you're ready to point your web browser at the homepage to see the fruits of your labor.

Wait a minute. Tomcat started? When did you deploy the application to Tomcat?

Spring Boot applications tend to bring everything they need with them and don't need to be

deployed to some application server. You never deployed your application to Tomcat … Tomcat is a part of your application! (I'll describe the details of how Tomcat became part of your application in section 1.3.6.)

Now that the application has started, point your web browser to http://localhost:8080 (or click the globe button in the Spring Boot Dashboard) and you should see something like figure 1.8. Your results may be different if you designed your own logo image. But it shouldn't vary much from what you see in figure 1.8.

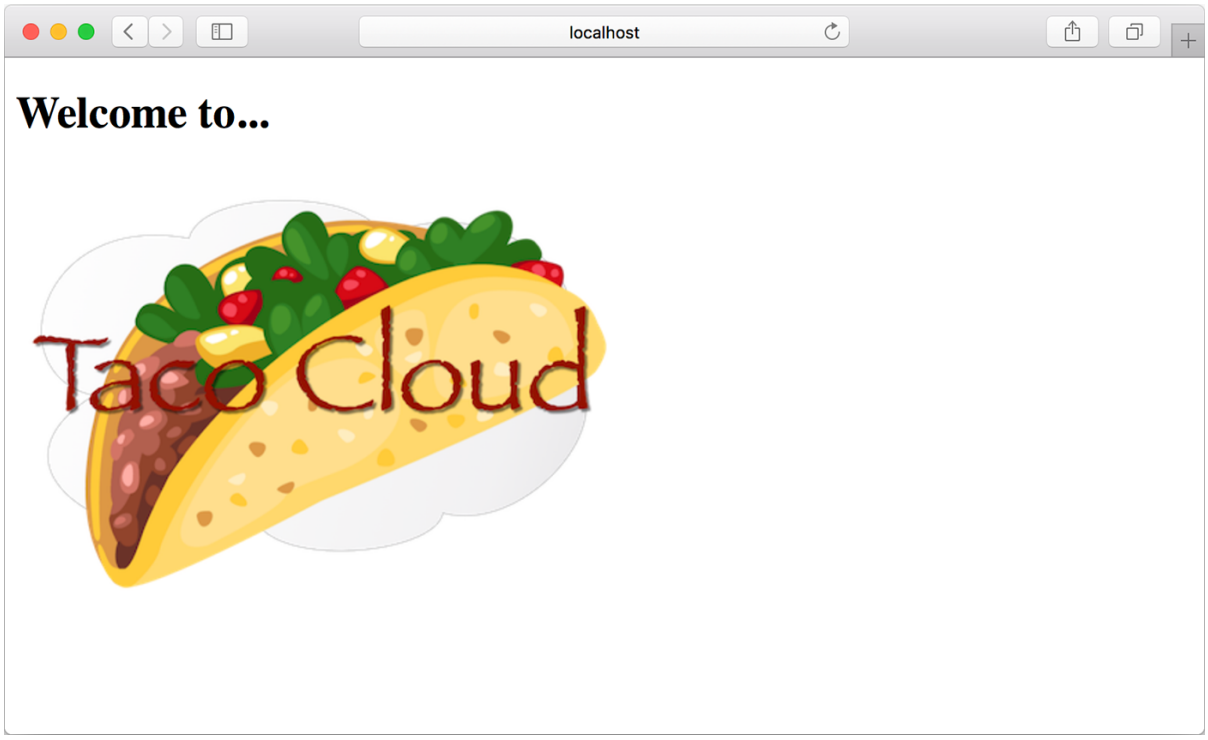

Figure 1.8 The Taco Cloud homepage

It may not be much to look at. But this isn't exactly a book on graphic design. The humble appearance of the homepage is more than sufficient for now. And it provides you a solid start on getting to know Spring.

One thing I've glossed over up until now is DevTools. You selected it as a dependency when initializing your project. It appears as a dependency in the produced pom.xml file. And the Spring Boot Dashboard even shows that the project has DevTools enabled. But what is DevTools, and what does it do for you? Let's take a quick survey of a couple of DevTools' most useful features.

1.3.5 Getting to know Spring Boot DevTools

As its name suggests, DevTools provides Spring developers with some handy development-time tools. Among those are

- Automatic application restart when code changes
- Automatic browser refresh when browser-destined resources (such as templates, JavaScript, stylesheets, and so on) change
- Automatic disable of template caches
- Built in H2 Console if the H2 database is in use

It's important to understand that DevTools isn't an IDE plugin, nor does it require that you use a specific IDE. It works equally well in Spring Tool Suite, IntelliJ IDEA, and NetBeans. Furthermore, because it's only intended for development purposes, it's smart enough to disable itself when deploying in a production setting. (We'll discuss how it does this when you get around to deploying your application in chapter 19.) For now, let's focus on the most useful features of Spring Boot DevTools, starting with automatic application restart.

AUTOMATIC APPLICATION RESTART

With DevTools as part of your project, you'll be able to make changes to Java code and properties files in the project and see those changes applied after a brief moment. DevTools monitors for changes, and when it sees something has changed, it automatically restarts the application.

More precisely, when DevTools is in play, the application is loaded into two separate class loaders in the Java virtual machine (JVM). One class loader is loaded with your Java code, property files, and pretty much anything that's in the src/main/ path of the project. These are items that are likely to change frequently. The other class loader is loaded with dependency libraries, which aren't likely to change as often.

When a change is detected, DevTools reloads only the class loader containing your project code and restarts the Spring application context, but leaves the other class loader and the JVM intact. Although subtle, this strategy affords a small reduction in the time it takes to start the application.

The downside of this strategy is that changes to dependencies won't be available in automatic restarts. That's because the class loader containing dependency libraries isn't automatically reloaded. This means that any time you add, change, or remove a dependency in your build specification, you'll need to do a hard restart of the application for those changes to take effect.

AUTOMATIC BROWSER REFRESH AND TEMPLATE CACHE DISABLE

By default, template options such as Thymeleaf and FreeMarker are configured to cache the results of template parsing so that templates don't need to be reparsed with every request they serve. This is great in production, as it buys a bit of performance benefit.

Cached templates, however, are not so great at development time. Cached templates make it impossible to make changes to the templates while the application is running and see the results

after refreshing the browser. Even if you've made changes, the cached template will still be in use until you restart the application.

DevTools addresses this issue by automatically disabling all template caching. Make as many changes as you want to your templates and know that you're only a browser refresh away from seeing the results.

But if you're like me, you don't even want to be burdened with the effort of clicking the browser's refresh button. It'd be much nicer if you could make the changes and witness the results in the browser immediately. Fortunately, DevTools has something special for those of us who are too lazy to click a refresh button.

When DevTools is in play, it automatically enables a LiveReload (http://livereload.com/) server along with your application. By itself, the LiveReload server isn't very useful. But when coupled with a corresponding LiveReload browser plugin, it causes your browser to automatically refresh when changes are made to templates, images, stylesheets, JavaScript, and so on—in fact, almost anything that ends up being served to your browser.

LiveReload has browser plugins for Google Chrome, Safari, and Firefox browsers. (Sorry, Internet Explorer and Edge fans.) Visit http://livereload.com/extensions/ to find information on how to install LiveReload for your browser.

BUILT IN H2 CONSOLE

Although your project doesn't yet use a database, that will change in chapter 3. If you choose to use the H2 database for development, DevTools will also automatically enable an H2 Console that you can access from your web browser. You only need to point your web browser to http://localhost:8080/h2-console to gain insight into the data your application is working with.

At this point, you've written a complete, albeit simple, Spring application. You'll expand on it throughout the course of the book. But now is a good time to step back and review what you've accomplished and how Spring played a part.

1.3.6 Let's review

Think back on how you got to this point. In short, these are the steps you've taken to build your Spring-based Taco Cloud application:

- You created an initial project structure using Spring Initializr.
- You wrote a controller class to handle the homepage request.
- You defined a view template to render the homepage.
- You wrote a simple test class to prove out your work.

Seems pretty straightforward, doesn't it? With the exception of the first step to bootstrap the

project, each action you've taken has been keenly focused on achieving the goal of producing a homepage.

In fact, almost every line of code you've written is aimed toward that goal. Not counting Java `import` statements, I count only two lines of code in your controller class and no lines in the view template that are Spring-specific. And although the bulk of the test class utilizes Spring testing support, it seems a little less invasive in the context of a test.

That's an important benefit of developing with Spring. You can focus on the code that meets the requirements of an application rather than on satisfying the demands of a framework. Although you'll no doubt need to write some framework-specific code from time to time, it'll usually be only a small fraction of your codebase. As I said before, Spring (with Spring Boot) can be considered the *frameworkless framework*.

How does this even work? What is Spring doing behind the scenes to make sure your application needs are met? To understand what Spring is doing, let's start by looking at the build specification.

In the pom.xml file, you declared a dependency on the `Web` and `Thymeleaf` starters. These two dependencies transitively brought in a handful of other dependencies, including

- Spring's MVC framework
- Embedded Tomcat
- Thymeleaf and the Thymeleaf layout dialect

It also brought Spring Boot's autoconfiguration library along for the ride. When the application starts, Spring Boot autoconfiguration detects those libraries and automatically

- Configures the beans in the Spring application context to enable Spring MVC
- Configures the embedded Tomcat server in the Spring application context
- Configures a Thymeleaf view resolver for rendering Spring MVC views with Thymeleaf templates

In short, autoconfiguration does all the grunt work, leaving you to focus on writing code that implements your application functionality. That's a pretty sweet arrangement, if you ask me!

Your Spring journey has just begun. The Taco Cloud application only touched on a small portion of what Spring has to offer. Before you take your next step, let's survey the Spring landscape and see what landmarks you'll encounter on your journey.

1.4 Surveying the Spring landscape

To get an idea of the Spring landscape, look no further than the enormous list of checkboxes on the full version of the Spring Initializr web form. It lists over 100 dependency choices, so I won't try to list them all here or to provide a screenshot. But I encourage you to take a look. In the meantime, I'll mention a few of the highlights.

1.4.1 The core Spring Framework

As you might expect, the core Spring Framework is the foundation of everything else in the Spring universe. It provides the core container and dependency injection framework. But it also provides a few other essential features.

Among these is Spring MVC, Spring's web framework. You've already seen how to use Spring MVC to write a controller class to handle web requests. What you've not yet seen, however, is that Spring MVC can also be used to create REST APIs that produce non-HTML output. We're going to dig more into Spring MVC in chapter 2 and then take another look at how to use it to create REST APIs in chapter 6.

The core Spring Framework also offers some elemental data persistence support, specifically template-based JDBC support. You'll see how to use `JdbcTemplate` in chapter 3.

Spring includes support for reactive-style programming, including a new reactive web framework called Spring WebFlux that borrows heavily from Spring MVC. You'll look at Spring's reactive programming model in part 3 and Spring WebFlux specifically in chapter 10.

1.4.2 Spring Boot

We've already seen many of the benefits of Spring Boot, including starter dependencies and autoconfiguration. Be certain that we'll use as much of Spring Boot as possible throughout this book and avoid any form of explicit configuration, unless it's absolutely necessary. But in addition to starter dependencies and autoconfiguration, Spring Boot also offers a handful of other useful features:

- The Actuator provides runtime insight into the inner workings of an application, including metrics, thread dump information, application health, and environment properties available to the application.
- Flexible specification of environment properties.
- Additional testing support on top of the testing assistance found in the core framework.

What's more, Spring Boot offers an alternative programming model based on Groovy scripts that's called the Spring Boot CLI (command-line interface). With the Spring Boot CLI, you can

write entire applications as a collection of Groovy scripts and run them from the command line. We won't spend much time with the Spring Boot CLI, but we'll touch on it on occasion when it fits our needs.

Spring Boot has become such an integral part of Spring development; I can't imagine developing a Spring application without it. Consequently, this book takes a Spring Boot–centric view, and you might catch me using the word *Spring* when I'm referring to something that Spring Boot is doing.

1.4.3 Spring Data

Although the core Spring Framework comes with basic data persistence support, Spring Data provides something quite amazing: the ability to define your application's data repositories as simple Java interfaces, using a naming convention when defining methods to drive how data is stored and retrieved.

What's more, Spring Data is capable of working with several different kinds of databases, including relational (via JDBC or JPA), document (Mongo), graph (Neo4j), and others. You'll use Spring Data to help create repositories for the Taco Cloud application in chapter 3.

1.4.4 Spring Security

Application security has always been an important topic, and it seems to become more important every day. Fortunately, Spring has a robust security framework in Spring Security.

Spring Security addresses a broad range of application security needs, including authentication, authorization, and API security. Although the scope of Spring Security is too large to be properly covered in this book, we'll touch on some of the most common use cases in chapters 4 and 12.

1.4.5 Spring Integration and Spring Batch

At some point, most applications will need to integrate with other applications or even with other components of the same application. Several patterns of application integration have emerged to address these needs. Spring Integration and Spring Batch provide the implementation of these patterns for Spring-based applications.

Spring Integration addresses real-time integration where data is processed as it's made available. In contrast, Spring Batch addresses batched integration where data is allowed to collect for a time until some trigger (perhaps a time trigger) signals that it's time for the batch of data to be processed. You'll explore both Spring Batch and Spring Integration in chapter 9.

1.4.6 Spring Cloud

As I'm writing this, the application development world is entering a new era where we'll no longer develop our applications as single deployment unit monoliths and will instead compose applications from several individual deployment units known as *microservices*.

Microservices are a hot topic, addressing several practical development and runtime concerns. In doing so, however, they bring to fore their own challenges. Those challenges are met head-on by Spring Cloud, a collection of projects for developing cloud-native applications with Spring.

Spring Cloud covers a lot of ground, and it'd be impossible to cover it all in this book. We'll look at some of the most common components of Spring Cloud in chapters 13, 14, and 15. For a more complete discussion of Spring Cloud, I suggest taking a look at *Spring Microservices in Action* by John Carnell and Illary Huaylupo Sánchez (Manning, 2019, www.manning.com/books/spring-microservices-in-action-second-edition).

1.5 Summary

- Spring aims to make developer challenges easy, like creating web applications, working with databases, securing applications, and microservices.
- Spring Boot builds on top of Spring to make Spring even easier with simplified dependency management, automatic configuration, and runtime insights.
- Spring applications can be initialized using the Spring Initializr, which is web-based and supported natively in most Java development environments.
- The components, commonly referred to as beans, in a Spring application context can be declared explicitly with Java or XML, discovered by component scanning, or automatically configured with Spring Boot autoconfiguration.

2 Developing web applications

This chapter covers

- Presenting model data in the browser
- Processing and validating form input
- Choosing a view template library

First impressions are important. Curb appeal can sell a house long before the home buyer enters the door. A car's cherry paint job will turn more heads than what's under the hood. And literature is replete with stories of love at first sight. What's inside is very important, but what's outside—what's seen first—is important.

The applications you'll build with Spring will do all kinds of things, including crunching data, reading information from a database, and interacting with other applications. But the first impression your application users will get comes from the user interface. And in many applications, that UI is a web application presented in a browser.

In chapter 1, you created your first Spring MVC controller to display your application homepage. But Spring MVC can do far more than simply display static content. In this chapter, you'll develop the first major bit of functionality in your Taco Cloud application—the ability to design custom tacos. In doing so, you'll dig deeper into Spring MVC, and you'll see how to display model data and process form input.

2.1 Displaying information

Fundamentally, Taco Cloud is a place where you can order tacos online. But more than that, Taco Cloud wants to enable its customers to express their creative side and to design custom tacos from a rich palette of ingredients.

Therefore, the Taco Cloud web application needs a page that displays the selection of ingredients for taco artists to choose from. The ingredient choices may change at any time, so they shouldn't be hardcoded into an HTML page. Rather, the list of available ingredients should be fetched from a database and handed over to the page to be displayed to the customer.

In a Spring web application, it's a controller's job to fetch and process data. And it's a view's job to render that data into HTML that will be displayed in the browser. You're going to create the following components in support of the taco creation page:

- A domain class that defines the properties of a taco ingredient
- A Spring MVC controller class that fetches ingredient information and passes it along to the view
- A view template that renders a list of ingredients in the user's browser

The relationship between these components is illustrated in figure 2.1.

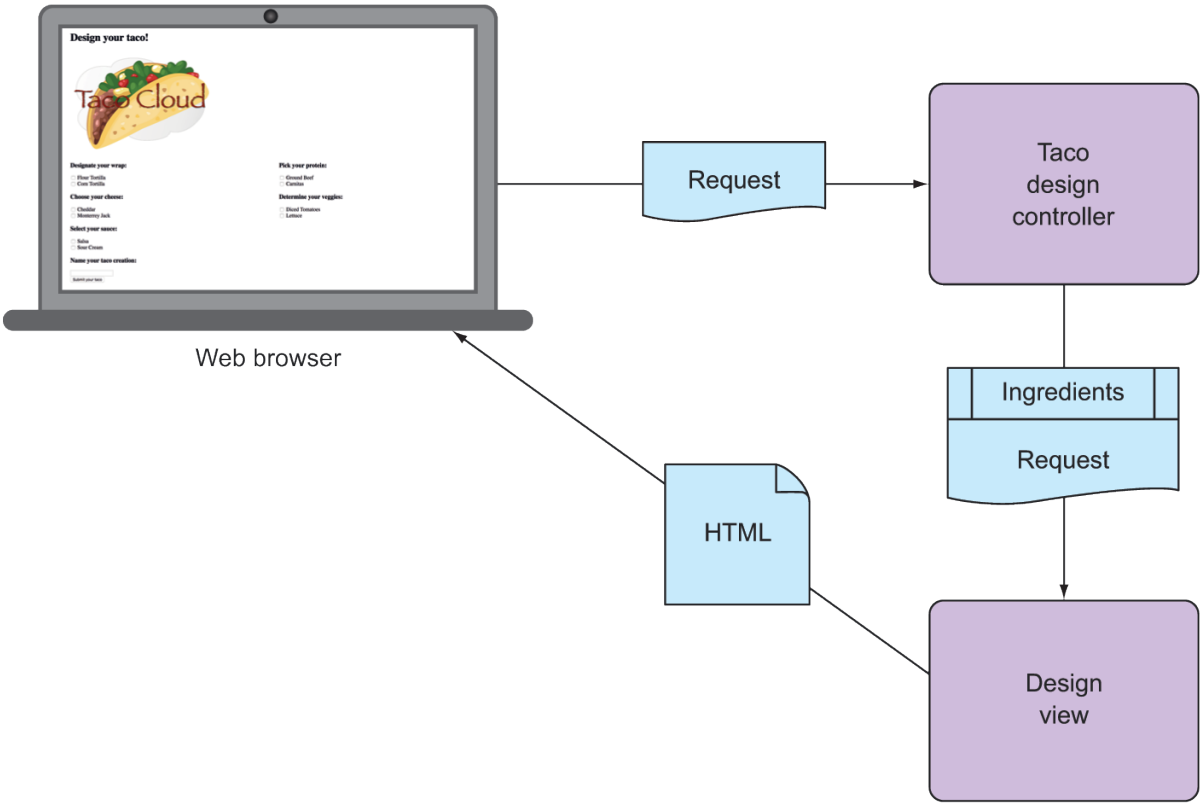

Figure 2.1 A typical Spring MVC request flow

Because this chapter focuses on Spring's web framework, we'll defer any of the database stuff to chapter 3. For now, the controller will be solely responsible for providing the ingredients to the view. In chapter 3, you'll rework the controller to collaborate with a repository that fetches ingredients data from a database.

Before you write the controller and view, let's hammer out the domain type that represents an

ingredient. This will establish a foundation on which you can develop your web components.

2.1.1 Establishing the domain

An application's domain is the subject area that it addresses—the ideas and concepts that influence the understanding of the application.[1] In the Taco Cloud application, the domain includes such objects as taco designs, the ingredients that those designs are composed of, customers, and taco orders placed by the customers. To get started, we'll focus on taco ingredients.

In your domain, taco ingredients are fairly simple objects. Each has a name as well as a type so that it can be visually categorized (proteins, cheeses, sauces, and so on). Each also has an ID by which it can easily and unambiguously be referenced. The following `Ingredient` class defines the domain object you need.

Listing 2.1 Defining taco ingredients

```
package tacos;

import lombok.Data;

@Data
public class Ingredient {

  private final String id;
  private final String name;
  private final Type type;

  public enum Type {
    WRAP, PROTEIN, VEGGIES, CHEESE, SAUCE
  }

}
```

As you can see, this is a run-of-the-mill Java domain class, defining the three properties needed to describe an ingredient. Perhaps the most unusual thing about the `Ingredient` class as defined in listing 2.1 is that it seems to be missing the usual set of getter and setter methods, not to mention useful methods like `equals()`, `hashCode()`, `toString()`, and others.

You don't see them in the listing partly to save space, but also because you're using an amazing library called Lombok to automatically generate those methods at compile-time so that they will be available at runtime. In fact, the `@Data` annotation at the class level is provided by Lombok and tells Lombok to generate all of those missing methods as well as a constructor that accepts all `final` properties as arguments. By using Lombok, you can keep the code for `Ingredient` slim and trim.

Lombok isn't a Spring library, but it's so incredibly useful that I find it hard to develop without it. And it's a lifesaver when I need to keep code examples in a book short and sweet.

To use Lombok, you'll need to add it as a dependency in your project. If you're using Spring Tool Suite, it's an easy matter of right-clicking on the pom.xml file and selecting Edit Starters from the Spring context menu option. The same selection of dependencies you were given in chapter 1 (in figure 1.4) will appear, giving you a chance to add or change your selected dependencies. Find the Lombok choice, make sure it's checked, and click OK; Spring Tool Suite will automatically add it to your build specification.

Alternatively, you can manually add it with the following entry in pom.xml:

```
<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
</dependency>
```

This dependency will provide you with Lombok annotations (such as `@Data`) at development time and with automatic method generation at compile-time. But you'll also need to add Lombok as an extension in your IDE, or your IDE will complain with errors about missing methods and `final` properties that aren't being set. Visit https://projectlombok.org/ to find out how to install Lombok in your IDE of choice.

NOTE **Why are there so many errors in my code?**

It bears repeating that when using Lombok, you must install the Lombok plugin into your IDE. Without it, your IDE won't be aware that Lombok is providing getters, setters, and other methods and will complain that they are missing.

Lombok is supported in a number of popular IDEs, including Eclipse, Spring Tool Suite, IntelliJ IDEA, and Visual Studio Code. Visit https://projectlombok.org/ for more information on how to install the Lombok plugin into your IDE.

I think you'll find Lombok to be very useful, but know that it's optional. You don't need it to develop Spring applications, so if you'd rather not use it, feel free to write those missing methods by hand. Go ahead … I'll wait.

Ingredients are only the essential building blocks of a taco. In order to captures how those ingredients are brought together, we'll define the `Taco` domain class:

Listing 2.2 A domain object defining a taco design

```
package tacos;
import java.util.List;
import lombok.Data;

@Data
public class Taco {

  private String name;

  private List<Ingredient> ingredients;

}
```

As you can see, `Taco` is a straightforward Java domain object with a couple of properties. Like `Ingredient`, the `Taco` class is annotated with `@Data` to have Lombok automatically generate essential JavaBean methods for you at compile time.

Now that we have defined `Ingredient` and `Taco`, we need one more domain class that defines customers specify the tacos that they want to order, along with payment and delivery information. That's the job of the `TacoOrder` class:

Listing 2.3 A domain object for taco orders

```
package tacos;
import java.util.List;
import java.util.ArrayList;
import lombok.Data;

@Data
public class TacoOrder {

  private String deliveryName;
  private String deliveryStreet;
  private String deliveryCity;
  private String deliveryState;
  private String deliveryZip;
  private String ccNumber;
  private String ccExpiration;
  private String ccCVV;

  private List<Taco> tacos = new ArrayList<>();

  public void addTaco(Taco taco) {
    this.tacos.add(taco);
  }
}
```

Aside from having more properties than either `Ingredient` or `Taco`, there's nothing particularly new to discuss about `TacoOrder`. It's a simple domain class with nine properties: 5 for delivery information, 3 for payment information, and 1 that is the list of `Taco` objects that make up the order. There's also an `addTaco()` method that's added for the convenience of adding tacos to the order.

Now that the domain types are defined, we're ready to put them to work. Let's add a few

controllers to handle web requests in the application.

2.1.2 Creating a controller class

Controllers are the major players in Spring's MVC framework. Their primary job is to handle HTTP requests and either hand a request off to a view to render HTML (browser-displayed) or write data directly to the body of a response (RESTful). In this chapter, we're focusing on the kinds of controllers that use views to produce content for web browsers. When we get to chapter 6, we'll look at writing controllers that handle requests in a REST API.

For the Taco Cloud application, you need a simple controller that will do the following:

- Handle HTTP `GET` requests where the request path is /design
- Build a list of ingredients
- Hand the request and the ingredient data off to a view template to be rendered as HTML and sent to the requesting web browser

The following `DesignTacoController` class addresses those requirements.

Listing 2.4 The beginnings of a Spring controller class

```
package tacos.web;

import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.SessionAttributes;

import lombok.extern.slf4j.Slf4j;
import tacos.Ingredient;
import tacos.Ingredient.Type;
import tacos.Taco;

@Slf4j
@Controller
@RequestMapping("/design")
@SessionAttributes("tacoOrder")
public class DesignTacoController {

@ModelAttribute
public void addIngredientsToModel(Model model) {
    List<Ingredient> ingredients = Arrays.asList(
      new Ingredient("FLTO", "Flour Tortilla", Type.WRAP),
      new Ingredient("COTO", "Corn Tortilla", Type.WRAP),
      new Ingredient("GRBF", "Ground Beef", Type.PROTEIN),
      new Ingredient("CARN", "Carnitas", Type.PROTEIN),
      new Ingredient("TMTO", "Diced Tomatoes", Type.VEGGIES),
      new Ingredient("LETC", "Lettuce", Type.VEGGIES),
      new Ingredient("CHED", "Cheddar", Type.CHEESE),
      new Ingredient("JACK", "Monterrey Jack", Type.CHEESE),
      new Ingredient("SLSA", "Salsa", Type.SAUCE),
      new Ingredient("SRCR", "Sour Cream", Type.SAUCE)
    );

    Type[] types = Ingredient.Type.values();
    for (Type type : types) {
      model.addAttribute(type.toString().toLowerCase(),
          filterByType(ingredients, type));
    }
}
  @GetMapping
  public String showDesignForm(Model model) {
    model.addAttribute("taco", new Taco());
    return "design";
  }

  private Iterable<Ingredient> filterByType(
      List<Ingredient> ingredients, Type type) {
    return ingredients
              .stream()
              .filter(x -> x.getType().equals(type))
              .collect(Collectors.toList());
  }

}
```

The first thing to note about `DesignTacoController` is the set of annotations applied at the class level. The first, `@Slf4j`, is a Lombok-provided annotation that, at compilation time, will

automatically generate an SLF4J (Simple Logging Facade for Java, https://www.slf4j.org/) `Logger` static property in the class. This modest annotation has the same effect as if you were to explicitly add the following lines within the class:

```
private static final org.slf4j.Logger log =
    org.slf4j.LoggerFactory.getLogger(DesignTacoController.class);
```

You'll make use of this `Logger` a little later.

The next annotation applied to `DesignTacoController` is `@Controller`. This annotation serves to identify this class as a controller and to mark it as a candidate for component scanning, so that Spring will discover it and automatically create an instance of `DesignTacoController` as a bean in the Spring application context.

`DesignTacoController` is also annotated with `@RequestMapping`. The `@RequestMapping` annotation, when applied at the class level, specifies the kind of requests that this controller handles. In this case, it specifies that `DesignTacoController` will handle requests whose path begins with /design.

HANDLING A GET REQUEST

The class-level `@RequestMapping` specification is refined with the `@GetMapping` annotation that adorns the `showDesignForm()` method. `@GetMapping`, paired with the class-level `@RequestMapping`, specifies that when an HTTP GET request is received for /design, `showDesignForm()` will be called to handle the request.

`@GetMapping` is just one member of a family of request-mapping annotations. Table 2.1 lists all of the request-mapping annotations available in Spring MVC.

Table 2.1 Spring MVC request-mapping annotations

Annotation	Description
`@RequestMapping`	General-purpose request handling
`@GetMapping`	Handles HTTP GET requests
`@PostMapping`	Handles HTTP POST requests
`@PutMapping`	Handles HTTP PUT requests
`@DeleteMapping`	Handles HTTP DELETE requests
`@PatchMapping`	Handles HTTP PATCH requests

When `showDesignForm()` handles a GET request for `/design`, it doesn't really do much. The main thing it does is return a `String` value of "design", which is the logical name of the view that will be used to render the model to the browser. But before it does that, it also populates the given `Model` with an empty `Taco` object under a key whose name is "design". This will enable the form to have a blank slate on which to create a taco masterpiece.

It would seem that a GET request to `/design` doesn't do much. But on the contrary, there's a bit more involved than what is found in the `showDesignForm()` method. You'll also notice that there is a method named `addIngredientsToModel()` that is annotated with `@ModelAttribute`. This method will also be invoked when a request is handled and will construct a list of `Ingredient` objects to be put into the model. The list is hardcoded for now. When we get to chapter 3, you'll pull the list of available taco ingredients from a database.

Once the list of ingredients is ready, the next few lines of `addIngredientsToModel()` filters the list by ingredient type (using a helper method named `filterByType()`). A list of ingredient types is then added as an attribute to the `Model` object that will be passed into `showDesignForm()`. `Model` is an object that ferries data between a controller and whatever view is charged with rendering that data. Ultimately, data that's placed in `Model` attributes is copied into the servlet request attributes, where the view can find them and use them to render a page in the user's browser.

Your `DesignTacoController` is really starting to take shape. If you were to run the application now and point your browser at the /design path, the `DesignTacoController` 's `showDesignForm()` and `addIngredientsToModel()` would be engaged, placing ingredients and an empty `Taco` into the model before passing the request on to the view. But because you haven't defined the view yet, the request would take a horrible turn, resulting in an HTTP 500 (Internal Server Error) error. To fix that, let's switch our attention to the view where the data will be decorated with HTML to be presented in the user's web browser.

2.1.3 Designing the view

After the controller is finished with its work, it's time for the view to get going. Spring offers several great options for defining views, including JavaServer Pages (JSP), Thymeleaf, FreeMarker, Mustache, and Groovy-based templates. For now, we'll use Thymeleaf, the choice we made in chapter 1 when starting the project. We'll consider a few of the other options in section 2.5.

We have already added Thymeleaf as a dependency in chapter 1. At runtime, Spring Boot autoconfiguration will see that Thymeleaf is in the classpath and will automatically create the beans that support Thymeleaf views for Spring MVC.

View libraries such as Thymeleaf are designed to be decoupled from any particular web framework. As such, they're unaware of Spring's model abstraction and are unable to work with the data that the controller places in `Model`. But they can work with servlet request attributes. Therefore, before Spring hands the request over to a view, it copies the model data into request attributes that Thymeleaf and other view-templating options have ready access to.

Thymeleaf templates are just HTML with some additional element attributes that guide a

template in rendering request data. For example, if there were a request attribute whose key is `"message"`, and you wanted it to be rendered into an HTML `<p>` tag by Thymeleaf, you'd write the following in your Thymeleaf template:

```
<p th:text="${message}">placeholder message</p>
```

When the template is rendered into HTML, the body of the `<p>` element will be replaced with the value of the servlet request attribute whose key is `"message"`. The `th:text` attribute is a Thymeleaf-namespaced attribute that performs the replacement. The `${}` operator tells it to use the value of a request attribute (`"message"`, in this case).

Thymeleaf also offers another attribute, `th:each`, that iterates over a collection of elements, rendering the HTML once for each item in the collection. This will come in handy as you design your view to list taco ingredients from the model. For example, to render just the list of `"wrap"` ingredients, you can use the following snippet of HTML:

```
<h3>Designate your wrap:</h3>
<div th:each="ingredient : ${wrap}">
  <input th:field="*{ingredients}" type="checkbox"
         th:value="${ingredient.id}"/>
  <span th:text="${ingredient.name}">INGREDIENT</span><br/>
</div>
```

Here, you use the `th:each` attribute on the `<div>` tag to repeat rendering of the `<div>` once for each item in the collection found in the `wrap` request attribute. On each iteration, the ingredient item is bound to a Thymeleaf variable named `ingredient`.

Inside the `<div>` element, there's a check box `<input>` element and a `<span>` element to provide a label for the check box. The check box uses Thymeleaf's `th:value` to set the rendered `<input>` element's `value` attribute to the value found in the ingredient's `id` property. The `th:field` attribute ultimately sets the `<input>` element's `name` attribute and is used to remember whether the checkbox is checked or not. When validation is added later, this will ensure that the checkbox maintains its state should the form need to be redisplayed after a validation error. The `<span>` element uses `th:text` to replace the `"INGREDIENT"` placeholder text with the value of the ingredient's `name` property.

When rendered with actual model data, one iteration of that `<div>` loop might look like this:

```
<div>
  <input name="ingredients" type="checkbox" value="FLTO" />
  <span>Flour Tortilla</span><br/>
</div>
```

Ultimately, the preceding Thymeleaf snippet is just part of a larger HTML form through which your taco artist users will submit their tasty creations. The complete Thymeleaf template, including all ingredient types and the form, is shown in the following listing.

Listing 2.5 The complete design-a-taco page

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">
  <head>
    <title>Taco Cloud</title>
    <link rel="stylesheet" th:href="@{/styles.css}" />
  </head>

  <body>
    <h1>Design your taco!</h1>
    <img th:src="@{/images/TacoCloud.png}"/>

    <form method="POST" th:object="${taco}">
    <div class="grid">
      <div class="ingredient-group" id="wraps">
      <h3>Designate your wrap:</h3>
      <div th:each="ingredient : ${wrap}">
        <input th:field="*{ingredients}" type="checkbox"
               th:value="${ingredient.id}"/>
        <span th:text="${ingredient.name}">INGREDIENT</span><br/>
      </div>
      </div>

      <div class="ingredient-group" id="proteins">
      <h3>Pick your protein:</h3>
      <div th:each="ingredient : ${protein}">
        <input th:field="*{ingredients}" type="checkbox"
               th:value="${ingredient.id}"/>
        <span th:text="${ingredient.name}">INGREDIENT</span><br/>
      </div>
      </div>

      <div class="ingredient-group" id="cheeses">
      <h3>Choose your cheese:</h3>
      <div th:each="ingredient : ${cheese}">
        <input th:field="*{ingredients}" type="checkbox"
               th:value="${ingredient.id}"/>
        <span th:text="${ingredient.name}">INGREDIENT</span><br/>
      </div>
      </div>

      <div class="ingredient-group" id="veggies">
      <h3>Determine your veggies:</h3>
      <div th:each="ingredient : ${veggies}">
        <input th:field="*{ingredients}" type="checkbox"
               th:value="${ingredient.id}"/>
        <span th:text="${ingredient.name}">INGREDIENT</span><br/>
      </div>
      </div>

      <div class="ingredient-group" id="sauces">
      <h3>Select your sauce:</h3>
      <div th:each="ingredient : ${sauce}">
        <input th:field="*{ingredients}" type="checkbox"
               th:value="${ingredient.id}"/>
        <span th:text="${ingredient.name}">INGREDIENT</span><br/>
      </div>
      </div>
      </div>

      <div>


      <h3>Name your taco creation:</h3>
      <input type="text" th:field="*{name}"/>
```

```
      <br/>

      <button>Submit Your Taco</button>
      </div>
    </form>
  </body>
</html>
```

As you can see, you repeat the `<div>` snippet for each of the types of ingredients. And you include a Submit button and field where the user can name their creation.

It's also worth noting that the complete template includes the Taco Cloud logo image and a `<link>` reference to a stylesheet.[2] In both cases, Thymeleaf's `@{}` operator is used to produce a context-relative path to the static artifacts that they're referencing. As you learned in chapter 1, static content in a Spring Boot application is served from the /static directory at the root of the classpath.

Now that your controller and view are complete, you can fire up the application to see the fruits of your labor. There are many ways to run a Spring Boot application. In chapter 1, I showed you how to run the application by clicking the start button in the Spring Boot Dashboard. But no matter how you fire up the Taco Cloud application, once it starts, point your browser to http://localhost:8080/design. You should see a page that looks something like figure 2.2.

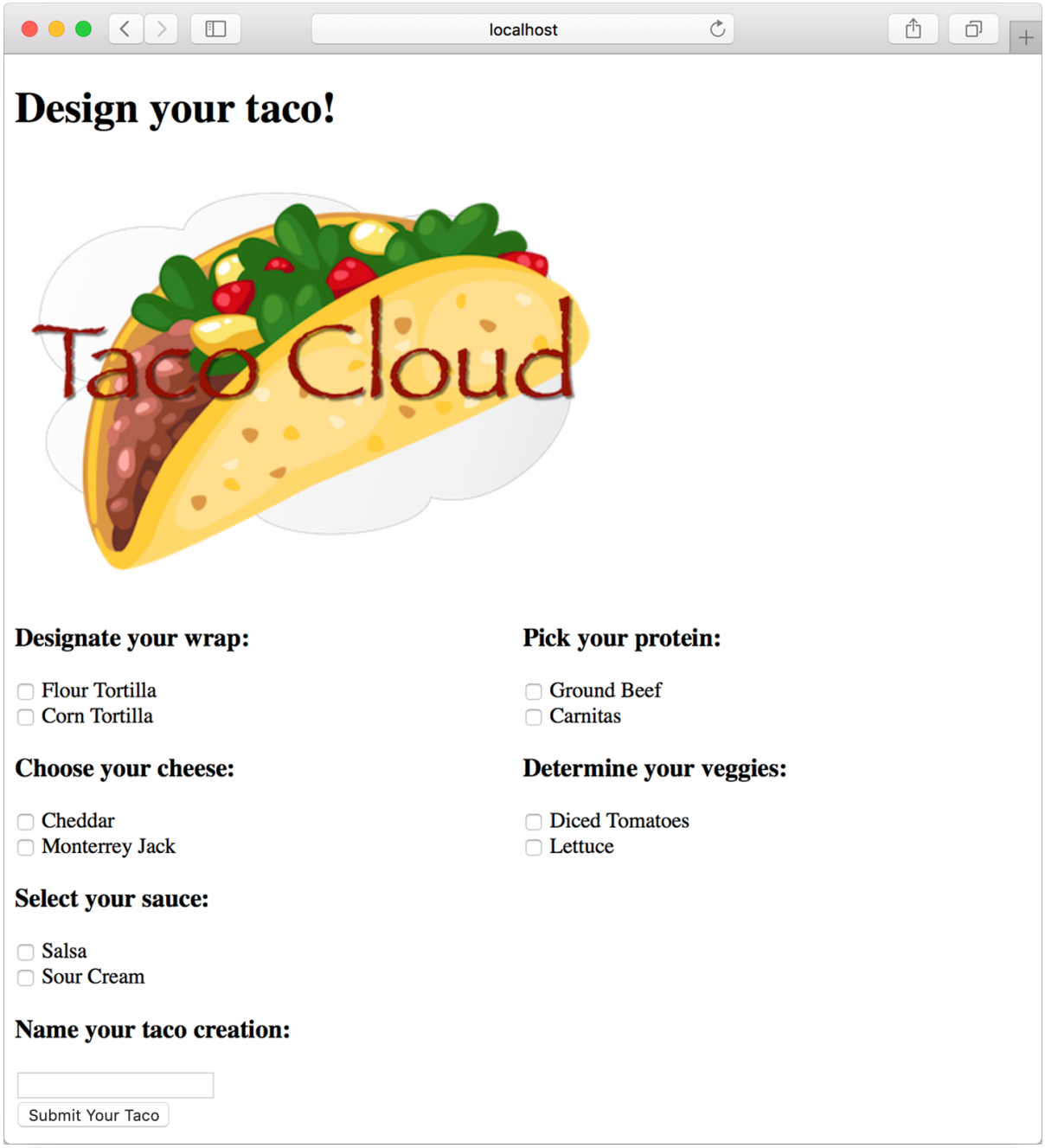

Figure 2.2 The rendered taco design page

It's looking good! A taco artist visiting your site is presented with a form containing a palette of taco ingredients from which they can create their masterpiece. But what happens when they click the Submit Your Taco button?

Your `DesignTacoController` isn't yet ready to accept taco creations. If the design form is submitted, the user will be presented with an error. (Specifically, it will be an HTTP 405 error: Request Method "POST" Not Supported.) Let's fix that by writing some more controller code that handles form submission.

2.2 Processing form submission

If you take another look at the `<form>` tag in your view, you can see that its `method` attribute is set to `POST`. Moreover, the `<form>` doesn't declare an `action` attribute. This means that when the form is submitted, the browser will gather up all the data in the form and send it to the server in an HTTP POST request to the same path for which a GET request displayed the form—the /design path.

Therefore, you need a controller handler method on the receiving end of that POST request. You need to write a new handler method in `DesignTacoController` that handles a POST request for /design.

In listing 2.4, you used the `@GetMapping` annotation to specify that the `showDesignForm()` method should handle HTTP GET requests for /design. Just like `@GetMapping` handles GET requests, you can use `@PostMapping` to handle POST requests. For handling taco design submissions, add the `processTaco()` method in the following listing to `DesignTacoController`.

Listing 2.6 Handling POST requests with `@PostMapping`

```
@PostMapping
public String processTaco(Taco taco) {
  // Save the taco...
  // We'll do this in chapter 3
  log.info("Processing taco: " + taco);

  return "redirect:/orders/current";
}
```

As applied to the `processTaco()` method, `@PostMapping` coordinates with the class-level `@RequestMapping` to indicate that `processTaco()` should handle POST requests for /design. This is precisely what you need to process a taco artist's submitted creations.

When the form is submitted, the fields in the form are bound to properties of a `Taco` object (whose class is shown in the next listing) that's passed as a parameter into `processTaco()`. From there, the `processTaco()` method can do whatever it wants with the `Taco` object.

If you look back at the form in listing 2.5, you'll see several `checkbox` elements, all with the name `ingredients`, and a text input element named `name`. Those fields in the form correspond directly to the `ingredients` and `name` properties of the `Taco` class.

The `name` field on the form only needs to capture a simple textual value. Thus the `name` property of `Taco` is of type `String`. The ingredients check boxes also have textual values, but because zero or many of them may be selected, the `ingredients` property that they're bound to is a `List<Ingredient>` that will capture each of the chosen ingredients.

But wait. If the ingredients check boxes have textual (e.g., `String`) values, but the `Taco` object represents a list of ingredients as `List<Ingredient>`, then isn't there a mismatch? How can a textual list like `["flto", "grbf", "letc"]` be bound to a list of `Ingredient` objects that are richer objects containing not only an ID, but a descriptive name and ingredient type?

That's where a converter comes in handy. A converter is any class that implements Spring's `Converter` interface and implements its `convert()` method to take one value and convert it to another. To convert a `String` to an `Ingredient`, we'll use the `IngredientByIdConverter` shown here:

Listing 2.7 Converting Strings to Ingredients

```
package tacos.web;

import java.util.HashMap;
import java.util.Map;

import org.springframework.core.convert.converter.Converter;
import org.springframework.stereotype.Component;

import tacos.Ingredient;
import tacos.Ingredient.Type;

@Component
public class IngredientByIdConverter implements Converter<String, Ingredient> {

  private Map<String, Ingredient> ingredientMap = new HashMap<>();

  public IngredientByIdConverter() {
    ingredientMap.put("FLTO",
        new Ingredient("FLTO", "Flour Tortilla", Type.WRAP));
    ingredientMap.put("COTO",
        new Ingredient("COTO", "Corn Tortilla", Type.WRAP));
    ingredientMap.put("GRBF",
        new Ingredient("GRBF", "Ground Beef", Type.PROTEIN));
    ingredientMap.put("CARN",
        new Ingredient("CARN", "Carnitas", Type.PROTEIN));
    ingredientMap.put("TMTO",
        new Ingredient("TMTO", "Diced Tomatoes", Type.VEGGIES));
    ingredientMap.put("LETC",
        new Ingredient("LETC", "Lettuce", Type.VEGGIES));
    ingredientMap.put("CHED",
        new Ingredient("CHED", "Cheddar", Type.CHEESE));
    ingredientMap.put("JACK",
        new Ingredient("JACK", "Monterrey Jack", Type.CHEESE));
    ingredientMap.put("SLSA",
        new Ingredient("SLSA", "Salsa", Type.SAUCE));
    ingredientMap.put("SRCR",
        new Ingredient("SRCR", "Sour Cream", Type.SAUCE));
  }

  @Override
  public Ingredient convert(String id) {
    return ingredientMap.get(id);
  }

}
```

Because we don't yet have a database from which to pull `Ingredient` objects from, the constructor of `IngredientByIdConverter` creates a `Map` keyed on a `String` which is the

ingredient ID and whose values are `Ingredient` objects. (In chapter 3, we'll adapt this converter to pull the ingredient data from a database instead of being hardcoded like this.) The `convert()` method then simply takes a `String` which is the ingredient ID and uses it to lookup the `Ingredient` from the map.

Notice that the `IngredientByIdConverter` is annotated with `@Component` to make it discoverable as a bean in the Spring application context. Spring Boot autoconfiguration will discover this, and any other `Converter` beans, and will automatically register them with Spring MVC to be used when conversion of request parameters to bound properties is needed.

For now, the `processTaco()` method does nothing with the `Taco` object. In fact, it doesn't do much of anything at all. That's OK. In chapter 3, you'll add some persistence logic that will save the submitted `Taco` to a database.

Just as with the `showDesignForm()` method, `processTaco()` finishes by returning a `String` value. And just like `showDesignForm()`, the value returned indicates a view that will be shown to the user. But what's different is that the value returned from `processTaco()` is prefixed with `"redirect:"`, indicating that this is a redirect view. More specifically, it indicates that after `processTaco()` completes, the user's browser should be redirected to the relative path /orders/current.

The idea is that after creating a taco, the user will be redirected to an order form from which they can place an order to have their taco creations delivered. But you don't yet have a controller that will handle a request for /orders/current.

Given what you now know about `@Controller`, `@RequestMapping`, and `@Get-Mapping`, you can easily create such a controller. It might look something like the following listing.

Listing 2.8 A controller to present a taco order form

```
package tacos.web;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;

import lombok.extern.slf4j.Slf4j;
import tacos.TacoOrder;

@Slf4j
@Controller
@RequestMapping("/orders")
public class OrderController {

  @GetMapping("/current")
  public String orderForm(Model model) {
    model.addAttribute("tacoOrder", new TacoOrder());
    return "orderForm";
  }

}
```

Once again, you use Lombok's `@Slf4j` annotation to create a free SLF4J `Logger` object at compile-time. You'll use this `Logger` in a moment to log the details of the order that's submitted.

The class-level `@RequestMapping` specifies that any request-handling methods in this controller will handle requests whose path begins with /orders. When combined with the method-level `@GetMapping`, it specifies that the `orderForm()` method will handle HTTP GET requests for /orders/current.

As for the `orderForm()` method itself, it's extremely basic, only returning a logical view name of `orderForm`. Once you have a way to persist taco creations to a database in chapter 3, you'll revisit this method and modify it to populate the model with a list of `Taco` objects to be placed in the order.

The `orderForm` view is provided by a Thymeleaf template named orderForm.html, which is shown next.

Listing 2.9 A taco order form view

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">
  <head>
    <title>Taco Cloud</title>
    <link rel="stylesheet" th:href="@{/styles.css}" />
  </head>

  <body>

    <form method="POST" th:action="@{/orders}" th:object="${tacoOrder}">
      <h1>Order your taco creations!</h1>

      <img th:src="@{/images/TacoCloud.png}"/>
      <a th:href="@{/design}" id="another">Design another taco</a><br/>

      <h3>Deliver my taco masterpieces to...</h3>
      <label for="deliveryName">Name: </label>
      <input type="text" th:field="*{deliveryName}"/>
      <br/>

      <label for="deliveryStreet">Street address: </label>
      <input type="text" th:field="*{deliveryStreet}"/>
      <br/>

      <label for="deliveryCity">City: </label>
      <input type="text" th:field="*{deliveryCity}"/>
      <br/>

      <label for="deliveryState">State: </label>
      <input type="text" th:field="*{deliveryState}"/>
      <br/>

      <label for="deliveryZip">Zip code: </label>
      <input type="text" th:field="*{deliveryZip}"/>
      <br/>

      <h3>Here's how I'll pay...</h3>
      <label for="ccNumber">Credit Card #: </label>
      <input type="text" th:field="*{ccNumber}"/>
      <br/>

      <label for="ccExpiration">Expiration: </label>
      <input type="text" th:field="*{ccExpiration}"/>
      <br/>

      <label for="ccCVV">CVV: </label>
      <input type="text" th:field="*{ccCVV}"/>
      <br/>

      <input type="submit" value="Submit Order"/>
    </form>
  </body>
</html>
```

For the most part, the `orderForm.html` view is typical HTML/Thymeleaf content, with very little of note. But notice that the `<form>` tag here is different from the `<form>` tag used in listing 2.5 in that it also specifies a form action. Without an action specified, the form would submit an HTTP POST request back to the same URL that presented the form. But here, you specify that the form should be POSTed to /orders (using Thymeleaf's `@{…}` operator for a context-relative path).

Therefore, you're going to need to add another method to your `OrderController` class that handles POST requests for /orders. You won't have a way to persist orders until the next chapter, so you'll keep it simple here—something like what you see in the next listing.

Listing 2.10 Handling a taco order submission

```
@PostMapping
public String processOrder(Order order) {
  log.info("Order submitted: " + order);
  return "redirect:/";
}
```

When the `processOrder()` method is called to handle a submitted order, it's given an `TacoOrder` object whose properties are bound to the submitted form fields. `TacoOrder`, much like `Taco`, is a fairly straightforward class that carries order information.

Now that you've developed an `OrderController` and the order form view, you're ready to try it out. Open your browser to http://localhost:8080/design, select some ingredients for your taco, and click the Submit Your Taco button. You should see a form similar to what's shown in figure 2.3.

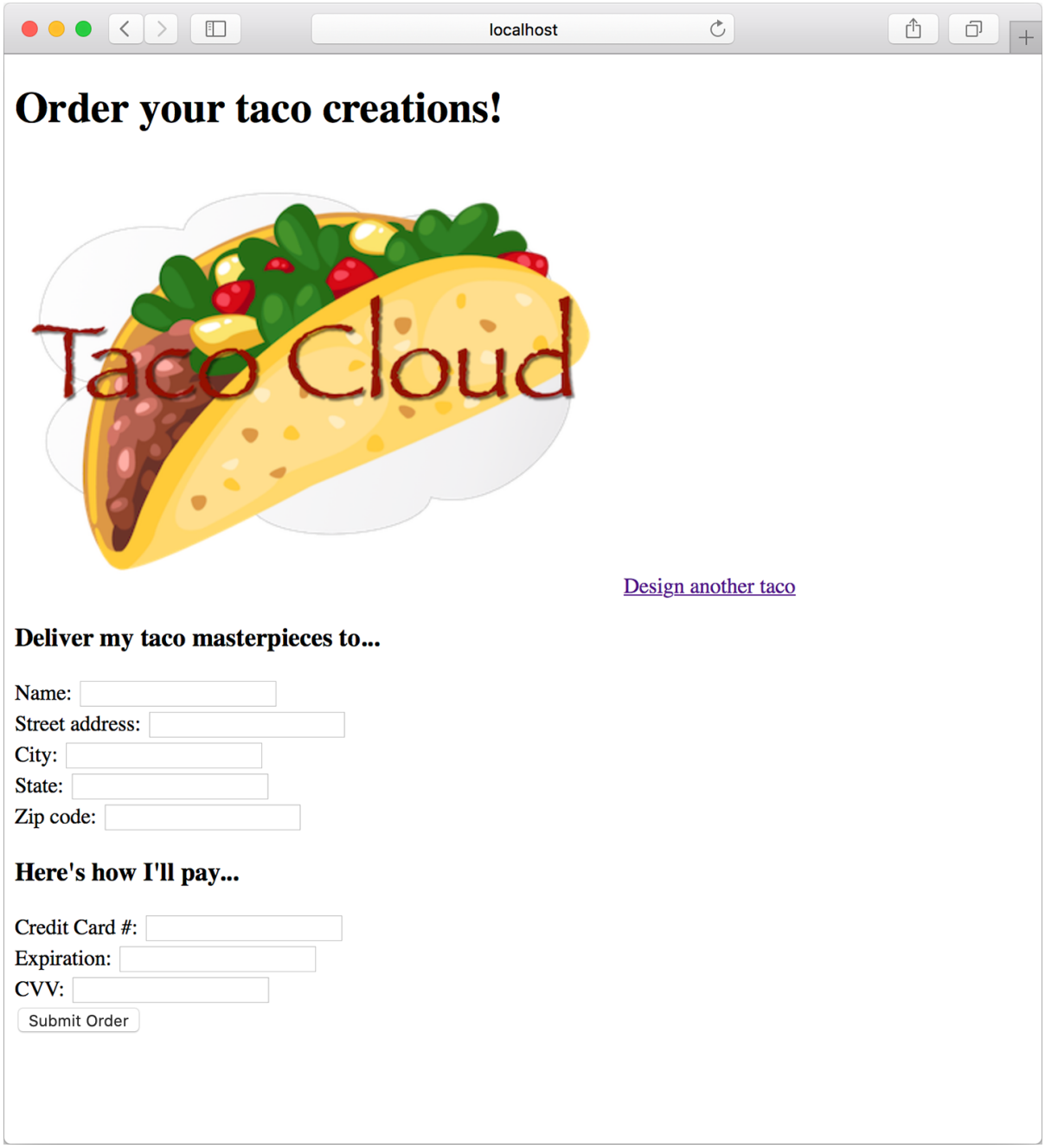

Figure 2.3 The taco order form

Fill in some fields in the form, and press the Submit Order button. As you do, keep an eye on the application logs to see your order information. When I tried it, the log entry looked something like this (reformatted to fit the width of this page):

```
Order submitted: TacoOrder(deliveryName=Craig Walls, deliveryStreet=1234 7th
Street, deliveryCity=Somewhere, deliveryState=Who knows?, deliveryZip=zipzap,
ccNumber=Who can guess?, ccExpiration=Some day, ccCVV=See-vee-vee, tacos=[])
```

It appears that the `processOrder()` method did its job, handling the form submission by logging details about the order. The `tacos` property is empty, but that's only because we're not doing anything in `DesignTacoController` to persist tacos or associate them with the order

(yet—we'll do that in chapter 3). But if you look carefully at the log entry from my test order, you can see that it let a little bit of bad information get in. Most of the fields in the form contained data that couldn't possibly be correct. Let's add some validation to ensure that the data provided at least resembles the kind of information required.

2.3 Validating form input

When designing a new taco creation, what if the user selects no ingredients or fails to specify a name for their creation? When submitting the order, what if they fail to fill in the required address fields? Or what if they enter a value into the credit card field that isn't even a valid credit card number?

As things stand now, nothing will stop the user from creating a taco without any ingredients or with an empty delivery address, or even submitting the lyrics to their favorite song as the credit card number. That's because you haven't yet specified how those fields should be validated.

One way to perform form validation is to litter the `processTaco()` and `processOrder()` methods with a bunch of `if`/`then` blocks, checking each and every field to ensure that it meets the appropriate validation rules. But that would be cumbersome and difficult to read and debug.

Fortunately, Spring supports Java's Bean Validation API (also known as JSR-303; https://jcp.org/en/jsr/detail?id=303). This makes it easy to declare validation rules as opposed to explicitly writing declaration logic in your application code.

To apply validation in Spring MVC, you need to

- Add the Spring Validation starter to the build.
- Declare validation rules on the class that is to be validated: specifically, the Taco class.
- Specify that validation should be performed in the controller methods that require validation: specifically, the `DesignTacoController`'s `processTaco()` method and `OrderController`'s `processOrder()` method.
- Modify the form views to display validation errors.

The Validation API offers several annotations that can be placed on properties of domain objects to declare validation rules. Hibernate's implementation of the Validation API adds even more validation annotations. Both can be added to a project by adding the Spring Validation starter to the build. The "Validation" checkbox in the Spring Boot Starters wizard will get the job done, but if you prefer manually editing your build, the following entry in the Maven pom.xml file will do the trick:

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

Or if you're using Gradle, then this is the dependency you'll need:

```
implementation 'org.springframework.boot:spring-boot-starter-validation'
```

> **NOTE** **Is the validation starter required?**
>
> In earlier versions of Spring Boot, the Spring Validation starter was automatically included with the web starter. Starting with Spring Boot 2.3.0, you'll need to explicitly add it to your build if you intend to apply validation.

With the validation starter in place, let's see how you can apply a few annotations to validate a submitted `Taco` or `TacoOrder`.

2.3.1 Declaring validation rules

For the `Taco` class, you want to ensure that the `name` property isn't empty or `null` and that the list of selected ingredients has at least one item. The following listing shows an updated `Taco` class that uses `@NotNull` and `@Size` to declare those validation rules.

Listing 2.11 Adding validation to the `Taco` domain class

```
package tacos;
import java.util.List;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;
import lombok.Data;

@Data
public class Taco {

  @NotNull
  @Size(min=5, message="Name must be at least 5 characters long")
  private String name;

  @NotNull
  @Size(min=1, message="You must choose at least 1 ingredient")
  private List<Ingredient> ingredients;

}
```

You'll notice that in addition to requiring that the `name` property isn't `null`, you declare that it should have a value that's at least 5 characters in length.

When it comes to declaring validation on submitted taco orders, you must apply annotations to the `TacoOrder` class. For the address properties, you only want to be sure that the user doesn't leave any of the fields blank. For that, you'll use the `@NotBlank` annotation.

Validation of the payment fields, however, is a bit more exotic. You need to not only ensure that the `ccNumber` property isn't empty, but that it contains a value that could be a valid credit card number. The `ccExpiration` property must conform to a format of MM/YY (two-digit month and year). And the `ccCVV` property needs to be a three-digit number. To achieve this kind of

validation, you need to use a few other Java Bean Validation API annotations and borrow a validation annotation from the Hibernate Validator collection of annotations. The following listing shows the changes needed to validate the `TacoOrder` class.

Listing 2.12 Validating order fields

```
package tacos;
import javax.validation.constraints.Digits;
import javax.validation.constraints.NotBlank;
import javax.validation.constraints.Pattern;
import org.hibernate.validator.constraints.CreditCardNumber;
import java.util.List;
import java.util.ArrayList;
import lombok.Data;

@Data
public class TacoOrder {

  @NotBlank(message="Delivery name is required")
  private String deliveryName;

  @NotBlank(message="Street is required")
  private String deliveryStreet;

  @NotBlank(message="City is required")
  private String deliveryCity;

  @NotBlank(message="State is required")
  private String deliveryState;

  @NotBlank(message="Zip code is required")
  private String deliveryZip;

  @CreditCardNumber(message="Not a valid credit card number")
  private String ccNumber;

  @Pattern(regexp="^(0[1-9]|1[0-2])([\\/])([1-9][0-9])$",
           message="Must be formatted MM/YY")
  private String ccExpiration;

  @Digits(integer=3, fraction=0, message="Invalid CVV")
  private String ccCVV;

  private List<Taco> tacos = new ArrayList<>();

  public void addTaco(Taco taco) {
    this.tacos.add(taco);
  }
}
```

As you can see, the `ccNumber` property is annotated with `@CreditCardNumber`. This annotation declares that the property's value must be a valid credit card number that passes the Luhn algorithm check (https://en.wikipedia.org/wiki/Luhn_algorithm). This prevents user mistakes and deliberately bad data but doesn't guarantee that the credit card number is actually assigned to an account or that the account can be used for charging.

Unfortunately, there's no ready-made annotation for validating the MM/YY format of the `ccExpiration` property. I've applied the `@Pattern` annotation, providing it with a regular expression that ensures that the property value adheres to the desired format. If you're wondering

how to decipher the regular expression, I encourage you to check out the many online regular expression guides, including http://www.regular-expressions.info/. Regular expression syntax is a dark art and certainly outside the scope of this book.

Finally, the `ccCVV` property is annotated with `@Digits` to ensure that the value contains exactly three numeric digits.

All of the validation annotations include a `message` attribute that defines the message you'll display to the user if the information they enter doesn't meet the requirements of the declared validation rules.

2.3.2 Performing validation at form binding

Now that you've declared how a `Taco` and `TacoOrder` should be validated, we need to revisit each of the controllers, specifying that validation should be performed when the forms are POSTed to their respective handler methods.

To validate a submitted `Taco`, you need to add the Java Bean Validation API's `@Valid` annotation to the `Taco` argument of `DesignTacoController`'s `processTaco()` method.

Listing 2.13 Validating a POSTed `Taco`

```
@PostMapping
public String processTaco(@Valid @ModelAttribute("taco") Taco taco, Errors errors) {
  if (errors.hasErrors()) {
    return "design";
  }

  // Save the taco...
  // We'll do this in chapter 3
  log.info("Processing taco: " + taco);

  return "redirect:/orders/current";
}
```

The `@Valid` annotation tells Spring MVC to perform validation on the submitted `Taco` object after it's bound to the submitted form data and before the `processTaco()` method is called. If there are any validation errors, the details of those errors will be captured in an `Errors` object that's passed into `processTaco()`. The first few lines of `processTaco()` consult the `Errors` object, asking its `hasErrors()` method if there are any validation errors. If there are, the method concludes without processing the `Taco` and returns the `"design"` view name so that the form is redisplayed.

To perform validation on submitted `TacoOrder` objects, similar changes are also required in the `processOrder()` method of `OrderController`.

Listing 2.14 Validating a POSTed `TacoOrder`

```
@PostMapping
public String processOrder(@Valid TacoOrder order, Errors errors) {
  if (errors.hasErrors()) {
    return "orderForm";
  }

  log.info("Order submitted: " + order);
  return "redirect:/";
}
```

In both cases, the method will be allowed to process the submitted data if there are no validation errors. If there are validation errors, the request will be forwarded to the form view to give the user a chance to correct their mistakes.

But how will the user know what mistakes require correction? Unless you call out the errors on the form, the user will be left guessing about how to successfully submit the form.

2.3.3 Displaying validation errors

Thymeleaf offers convenient access to the `Errors` object via the `fields` property and with its `th:errors` attribute. For example, to display validation errors on the credit card number field, you can add a `<span>` element that uses these error references to the order form template, as follows.

Listing 2.15 Displaying validation errors

```
<label for="ccNumber">Credit Card #: </label>
<input type="text" th:field="*{ccNumber}"/>
<span class="validationError"
      th:if="${#fields.hasErrors('ccNumber')}"
      th:errors="*{ccNumber}">CC Num Error</span>
```

Aside from a `class` attribute that can be used to style the error so that it catches the user's attention, the `<span>` element uses a `th:if` attribute to decide whether or not to display the `<span>`. The `fields` property's `hasErrors()` method checks if there are any errors in the `ccNumber` field. If so, the `<span>` will be rendered.

The `th:errors` attribute references the `ccNumber` field and, assuming there are errors for that field, it will replace the placeholder content of the `<span>` element with the validation message.

If you were to sprinkle similar `<span>` tags around the order form for the other fields, you might see a form that looks like figure 2.4 when you submit invalid information. The errors indicate that the name, city, and ZIP code fields have been left blank, and that all of the payment fields fail to meet the validation criteria.

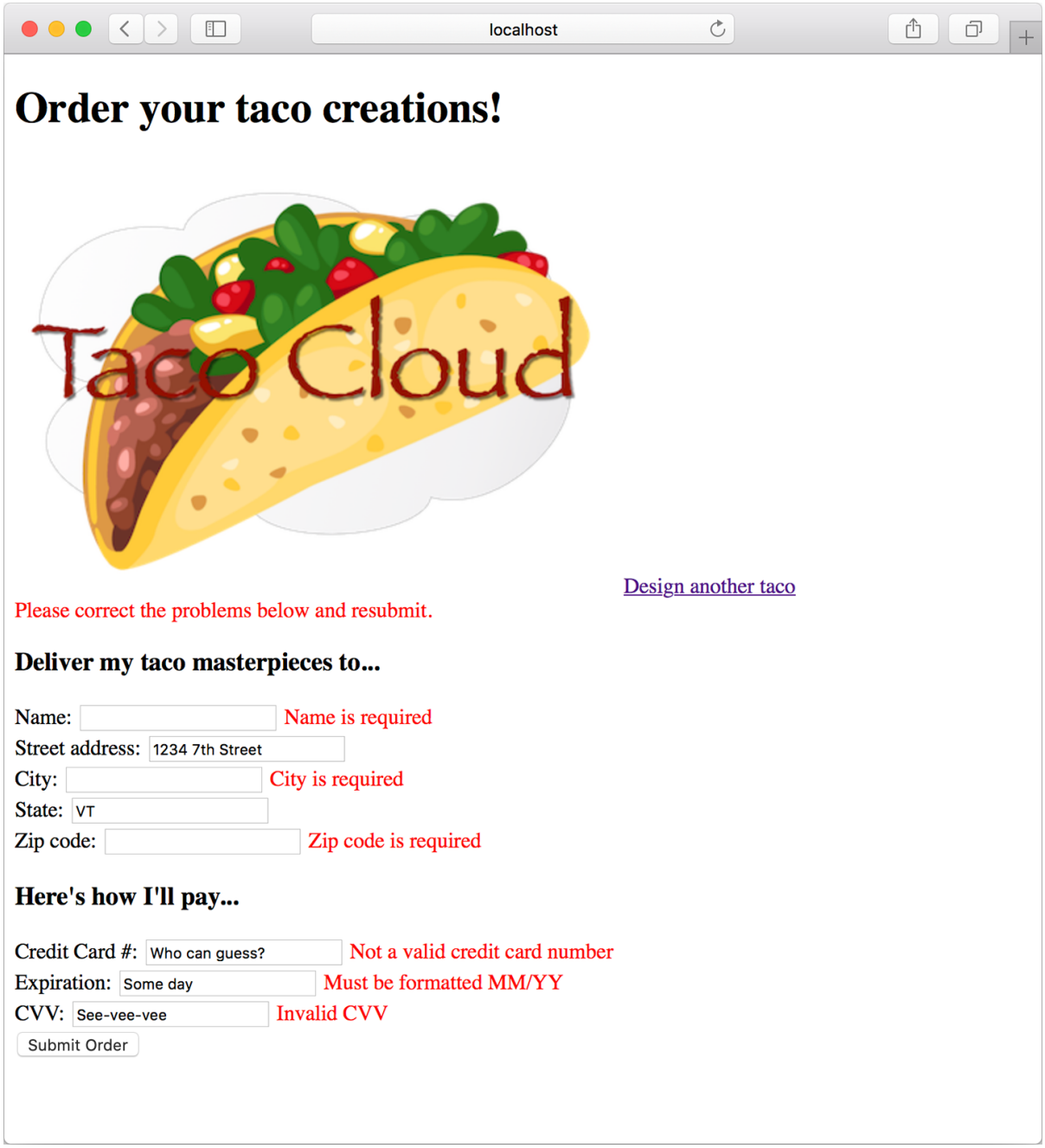

Figure 2.4 Validation errors displayed on the order form

Now your Taco Cloud controllers not only display and capture input, but they also validate that the information meets some basic validation rules. Let's step back and reconsider the `HomeController` from chapter 1, looking at an alternative implementation.

2.4 Working with view controllers

Thus far, you've written three controllers for the Taco Cloud application. Although each controller serves a distinct purpose in the functionality of the application, they all pretty much follow the same programming model:

- They're all annotated with `@Controller` to indicate that they're controller classes that should be automatically discovered by Spring component scanning and instantiated as beans in the Spring application context.
- All but `HomeController` are annotated with `@RequestMapping` at the class level to define a baseline request pattern that the controller will handle.
- They all have one or more methods that are annotated with `@GetMapping` or `@PostMapping` to provide specifics on which methods should handle which kinds of requests.

Most of the controllers you'll write will follow that pattern. But when a controller is simple enough that it doesn't populate a model or process input—as is the case with your `HomeController`—there's another way that you can define the controller. Have a look at the next listing to see how you can declare a view controller—a controller that does nothing but forward the request to a view.

Listing 2.16 Declaring a view controller

```
package tacos.web;

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 WebConfig implements WebMvcConfigurer {

  @Override
  public void addViewControllers(ViewControllerRegistry registry) {
    registry.addViewController("/").setViewName("home");
  }

}
```

The most significant thing to notice about `WebConfig` is that it implements the `WebMvcConfigurer` interface. `WebMvcConfigurer` defines several methods for configuring Spring MVC. Even though it's an interface, it provides default implementations of all the methods, so you only need to override the methods you need. In this case, you override `addViewControllers()`.

The `addViewControllers()` method is given a `ViewControllerRegistry` that you can use to register one or more view controllers. Here, you call `addViewController()` on the registry, passing in "/", which is the path for which your view controller will handle GET requests. That method returns a `ViewControllerRegistration` object, on which you immediately call `setViewName()` to specify `home` as the view that a request for "/" should be forwarded to.

And just like that, you've been able to replace `HomeController` with a few lines in a configuration class. You can now delete `HomeController`, and the application should still behave as it did before. The only other change required is to revisit `HomeControllerTest` from chapter 1, removing the reference to `HomeController` from the `@WebMvcTest` annotation, so that the test class will compile without errors.

Here, you've created a new `WebConfig` configuration class to house the view controller declaration. But any configuration class can implement `WebMvcConfigurer` and override the `addViewController` method. For instance, you could have added the same view controller declaration to the bootstrap `TacoCloudApplication` class like this:

```
@SpringBootApplication
public class TacoCloudApplication implements WebMvcConfigurer {

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

  @Override
  public void addViewControllers(ViewControllerRegistry registry) {
    registry.addViewController("/").setViewName("home");
  }

}
```

By extending an existing configuration class, you can avoid creating a new configuration class, keeping your project artifact count down. But I tend to prefer creating a new configuration class for each kind of configuration (web, data, security, and so on), keeping the application bootstrap configuration clean and simple.

Speaking of view controllers, and more generically the views that controllers forward requests to, so far you've been using Thymeleaf for all of your views. I like Thymeleaf a lot, but maybe you prefer a different template model for your application views. Let's have a look at Spring's many supported view options.

2.5 Choosing a view template library

For the most part, your choice of a view template library is a matter of personal taste. Spring is very flexible and supports many common templating options. With only a few small exceptions, the template library you choose will itself have no idea that it's even working with Spring.[3]

Table 2.2 catalogs the template options supported by Spring Boot autoconfiguration.

Table 2.2 Supported template options

Template	Spring Boot starter dependency
FreeMarker	`spring-boot-starter-freemarker`
Groovy Templates	`spring-boot-starter-groovy-templates`
JavaServer Pages (JSP)	None (provided by Tomcat or Jetty)
Mustache	`spring-boot-starter-mustache`
Thymeleaf	`spring-boot-starter-thymeleaf`

Generally speaking, you select the view template library you want, add it as a dependency in your build, and start writing templates in the /templates directory (under the src/main/resources

directory in a Maven- or Gradle-built project). Spring Boot will detect your chosen template library and automatically configure the components required for it to serve views for your Spring MVC controllers.

You've already done this with Thymeleaf for the Taco Cloud application. In chapter 1, you selected the Thymeleaf check box when initializing the project. This resulted in Spring Boot's Thymeleaf starter being included in the pom.xml file. When the application starts up, Spring Boot autoconfiguration detects the presence of Thymeleaf and automatically configures the Thymeleaf beans for you. All you had to do was start writing templates in /templates.

If you'd rather use a different template library, you simply select it at project initialization or edit your existing project build to include the newly chosen template library.

For example, let's say you wanted to use Mustache instead of Thymeleaf. No problem. Just visit the project pom.xml file and replace this,

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

with this:

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

Of course, you'd need to make sure that you write all the templates with Mustache syntax instead of Thymeleaf tags. The specifics of working with Mustache (or any of the template language choices) is well outside of the scope of this book, but to give you an idea of what to expect, here's a snippet from a Mustache template that will render one of the ingredient groups in the taco design form:

```
<h3>Designate your wrap:</h3>
{{#wrap}}
<div>
  <input name="ingredients" type="checkbox" value="{{id}}" />
  <span>{{name}}</span><br/>
</div>
{{/wrap}}
```

This is the Mustache equivalent of the Thymeleaf snippet in section 2.1.3. The `{{#wrap}}` block (which concludes with `{{/wrap}}`) iterates through a collection in the request attribute whose key is `wrap` and renders the embedded HTML for each item. The `{{id}}` and `{{name}}` tags reference the `id` and `name` properties of the item (which should be an `Ingredient`).

You'll notice in table 2.2 that JSP doesn't require any special dependency in the build. That's because the servlet container itself (Tomcat by default) implements the JSP specification, thus

requiring no further dependencies.

But there's a gotcha if you choose to use JSP. As it turns out, Java servlet containers—including embedded Tomcat and Jetty containers—usually look for JSPs somewhere under /WEB-INF. But if you're building your application as an executable JAR file, there's no way to satisfy that requirement. Therefore, JSP is only an option if you're building your application as a WAR file and deploying it in a traditional servlet container. If you're building an executable JAR file, you must choose Thymeleaf, FreeMarker, or one of the other options in table 2.2.

2.5.1 Caching templates

By default, templates are only parsed once, when they're first used, and the results of that parse are cached for subsequent use. This is a great feature for production, as it prevents redundant template parsing on each request and thus improves performance.

That feature is not so awesome at development time, however. Let's say you fire up your application and hit the taco design page and decide to make a few changes to it. When you refresh your web browser, you'll still be shown the original version. The only way you can see your changes is to restart the application, which is quite inconvenient.

Fortunately, there's a way to disable caching. All you need to do is set a template-appropriate caching property to `false`. Table 2.3 lists the caching properties for each of the supported template libraries.

Table 2.3 Properties to enable/disable template caching

Template	Cache enable property
FreeMarker	`spring.freemarker.cache`
Groovy Templates	`spring.groovy.template.cache`
Mustache	`spring.mustache.cache`
Thymeleaf	`spring.thymeleaf.cache`

By default, all of these properties are set to `true` to enable caching. You can disable caching for your chosen template engine by setting its cache property to `false`. For example, to disable Thymeleaf caching, add the following line in application.properties:

```
spring.thymeleaf.cache=false
```

The only catch is that you'll want to be sure to remove this line (or set it to `true`) before you deploy your application to production. One option is to set the property in a profile. (We'll talk about profiles in chapter 5.)

A much simpler option is to use Spring Boot's DevTools, as we opted to do in chapter 1. Among the many helpful bits of development-time help offered by DevTools, it will disable caching for all template libraries but will disable itself (and thus reenable template caching) when your

application is deployed.

2.6 Summary

- Spring offers a powerful web framework called Spring MVC that can be used to develop the web frontend for a Spring application.
- Spring MVC is annotation-based, enabling the declaration of request-handling methods with annotations such as `@RequestMapping`, `@GetMapping`, and `@PostMapping`.
- Most request-handling methods conclude by returning the logical name of a view, such as a Thymeleaf template, to which the request (along with any model data) is forwarded.
- Spring MVC supports validation through the Java Bean Validation API and implementations of the Validation API such as Hibernate Validator.
- View controllers can be used to handle HTTP GET requests for which no model data or processing is required.
- In addition to Thymeleaf, Spring supports a variety of view options, including FreeMarker, Groovy Templates, and Mustache.

3 Working with data

This chapter covers

- Using Spring's `JdbcTemplate`
- Creating Spring Data JDBC repositories
- Declaring JPA repositories with Spring Data

Most applications offer more than just a pretty face. Although the user interface may provide interaction with an application, it's the data it presents and stores that separates applications from static websites.

In the Taco Cloud application, you need to be able to maintain information about ingredients, tacos, and orders. Without a database to store this information, the application wouldn't be able to progress much further than what you developed in chapter 2.

In this chapter, you're going to add data persistence to the Taco Cloud application. You'll start by using Spring support for JDBC (Java Database Connectivity) to eliminate boilerplate code. Then you'll rework the data repositories to work with the JPA (Java Persistence API), eliminating even more code.

3.1 Reading and writing data with JDBC

For decades, relational databases and SQL have enjoyed their position as the leading choice for data persistence. Even though many alternative database types have emerged in recent years, the relational database is still a top choice for a general-purpose data store and will not likely be usurped from its position any time soon.

When it comes to working with relational data, Java developers have several options. The two

most common choices are JDBC and the JPA. Spring supports both of these with abstractions, making working with either JDBC or JPA easier than it would be without Spring. In this section, we'll focus on how Spring supports JDBC, and then we'll look at Spring support for JPA in section 3.2.

Spring JDBC support is rooted in the `JdbcTemplate` class. `JdbcTemplate` provides a means by which developers can perform SQL operations against a relational database without all the ceremony and boilerplate typically required when working with JDBC.

To gain an appreciation of what `JdbcTemplate` does, let's start by looking at an example of how to perform a simple query in Java without `JdbcTemplate`.

Listing 3.1 Querying a database without `JdbcTemplate`

```
@Override
public Optional<Ingredient> findById(String id) {
  Connection connection = null;
  PreparedStatement statement = null;
  ResultSet resultSet = null;
  try {
    connection = dataSource.getConnection();
    statement = connection.prepareStatement(
        "select id, name, type from Ingredient");
    statement.setString(1, id);
    resultSet = statement.executeQuery();
    Ingredient ingredient = null;
    if(resultSet.next()) {
      ingredient = new Ingredient(
          resultSet.getString("id"),
          resultSet.getString("name"),
          Ingredient.Type.valueOf(resultSet.getString("type")));
    }
    return Optional.of(ingredient);
  } catch (SQLException e) {
    // ??? What should be done here ???
  } finally {
    if (resultSet != null) {
      try {
        resultSet.close();
      } catch (SQLException e) {}
    }
    if (statement != null) {
      try {
        statement.close();
      } catch (SQLException e) {}
    }
    if (connection != null) {
      try {
        connection.close();
      } catch (SQLException e) {}
    }
  }
  return null;
}
```

I assure you that somewhere in listing 3.1 there are a couple of lines that query the database for ingredients. But I'll bet you had a hard time spotting that query needle in the JDBC haystack. It's surrounded by code that creates a connection, creates a statement, and cleans up by closing the

connection, statement, and result set.

To make matters worse, any number of things could go wrong when creating the connection or the statement, or when performing the query. This requires that you catch a `SQLException`, which may or may not be helpful in figuring out what went wrong or how to address the problem.

`SQLException` is a checked exception, which requires handling in a catch block. But the most common problems, such as failure to create a connection to the database or a mistyped query, can't possibly be addressed in a catch block and are likely to be rethrown for handling upstream. In contrast, consider the following method that uses Spring's `JdbcTemplate`.

Listing 3.2 Querying a database with `JdbcTemplate`

```
private JdbcTemplate jdbcTemplate;

public Optional<Ingredient> findById(String id) {
  List<Ingredient> results = jdbcTemplate.query(
      "select id, name, type from Ingredient where id=?",
      this::mapRowToIngredient,
      id);
  return results.size() == 0 ?
          Optional.empty() :
          Optional.of(results.get(0));
}
private Ingredient mapRowToIngredient(ResultSet row, int rowNum)
    throws SQLException{
  return new Ingredient(
      row.getString("id"),
      row.getString("name"),
      Ingredient.Type.valueOf(row.getString("type")));
}
```

The code in listing 3.2 is clearly much simpler than the raw JDBC example in listing 3.1; there aren't any statements or connections being created. And, after the method is finished, there isn't any cleanup of those objects. Finally, there isn't any handling of exceptions that can't properly be handled in a catch block. What's left is code that's focused solely on performing a query (the call to `JdbcTemplate`'s `query()` method) and mapping the results to an `Ingredient` object (handled by the `mapRowToIngredient()` method).

The code in listing 3.2 is a snippet of what you need to do to use `JdbcTemplate` to persist and read data in the Taco Cloud application. Let's take the next steps necessary to outfit the application with JDBC persistence. We'll start by making a few tweaks to the domain objects.

3.1.1 Adapting the domain for persistence

When persisting objects to a database, it's generally a good idea to have one field that uniquely identifies the object. Your `Ingredient` class already has an `id` field, but you need to add `id` fields to both `Taco` and `TacoOrder`.

Moreover, it might be useful to know when a `Taco` is created and when a `TacoOrder` is placed. You'll also need to add a field to each object to capture the date and time that the objects are saved. The following listing shows the new `id` and `createdAt` fields needed in the `Taco` class.

Listing 3.3 Adding ID and timestamp fields to the `Taco` class

```
@Data
public class Taco {

  private Long id;

  private Date createdAt = new Date();

   ...

}
```

Because you use Lombok to automatically generate accessor methods at runtime, there's no need to do anything more than declare the `id` and `createdAt` properties. They'll have appropriate getter and setter methods as needed at runtime. Similar changes are required in the `TacoOrder` class, as shown here:

```
@Data
public class TacoOrder implements Serializable {

  private static final long serialVersionUID = 1L;

  private Long id;

  private Date placedAt;
   ...

}
```

Again, Lombok automatically generates the accessor methods, so these are the only changes required in `TacoOrder`. (If for some reason you choose not to use Lombok, you'll need to write these methods yourself.)

Your domain classes are now ready for persistence. Let's see how to use `JdbcTemplate` to read and write them to a database.

3.1.2 Working with JdbcTemplate

Before you can start using `JdbcTemplate`, you need to add it to your project classpath. This can easily be accomplished by adding Spring Boot's JDBC starter dependency to the build:

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

You're also going to need a database where your data will be stored. For development purposes, an embedded database will be just fine. I favor the H2 embedded database, so I've added the

following dependency to the build:

```
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
  <scope>runtime</scope>
</dependency>
```

By default, the database name is randomly generated. But that makes it hard to determine the database URL if, for some reason, you need to connect to the database using the H2 Console (which Spring Boot DevTools enables at http://localhost:8080/h2-console). So, it's a good idea to pin down the database name by setting a couple of properties in `application.yml`:

```
spring:
  datasource:
    generate-unique-name: false
    name: tacocloud
```

By setting the `spring.datasource.generate-unique-name` property to `false`, we're telling Spring to not generate a unique random value for the database name. Instead, it should use the value set to the `spring.datasource.name` property. In this case, the database name will be "tacocloud". Consequently, the database URL will be "jdbc:h2:mem:tacocloud".

Later, you'll see how to configure the application to use an external database. But for now, let's move on to writing a repository that fetches and saves `Ingredient` data.

DEFINING JDBC REPOSITORIES

Your `Ingredient` repository needs to perform these operations:

- Query for all ingredients into a collection of `Ingredient` objects
- Query for a single `Ingredient` by its `id`
- Save an `Ingredient` object

The following `IngredientRepository` interface defines those three operations as method declarations:

```
package tacos.data;

import java.util.Optional;

import tacos.Ingredient;

public interface IngredientRepository {

  Iterable<Ingredient> findAll();

  Optional<Ingredient> findById(String id);

  Ingredient save(Ingredient ingredient);

}
```

Although the interface captures the essence of what you need an ingredient repository to do, you'll still need to write an implementation of `IngredientRepository` that uses `JdbcTemplate` to query the database. The code shown next is the first step in writing that implementation.

Listing 3.4 Beginning an ingredient repository with `JdbcTemplate`

```
package tacos.data;
import java.sql.ResultSet;
import java.sql.SQLException;

import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Repository;

import tacos.Ingredient;

@Repository
public class JdbcIngredientRepository implements IngredientRepository {

  private JdbcTemplate jdbcTemplate;

  public JdbcIngredientRepository(JdbcTemplate jdbcTemplate) {
    this.jdbcTemplate = jdbcTemplate;
  }

  ...

}
```

As you can see, `JdbcIngredientRepository` is annotated with `@Repository`. This annotation is one of a handful of stereotype annotations that Spring defines, including `@Controller` and `@Component`. By annotating `JdbcIngredientRepository` with `@Repository`, you declare that it should be automatically discovered by Spring component scanning and instantiated as a bean in the Spring application context.

When Spring creates the `JdbcIngredientRepository` bean, it injects it with `JdbcTemplate`. That's because when there's only one constructor, Spring implicitly applies autowiring of dependencies through that constructor's parameters. If there were more than one constructor, or if you just want autowiring to be explicitly stated, then you can annotated the constructor with `@Autowired` :

```
@Autowired
public JdbcIngredientRepository(JdbcTemplate jdbcTemplate) {
  this.jdbcTemplate = jdbcTemplate;
}
```

The constructor assigns `JdbcTemplate` to an instance variable that will be used in other methods to query and insert into the database. Speaking of those other methods, let's take a look at the implementations of `findAll()` and `findById()`.

Listing 3.5 Querying the database with `JdbcTemplate`

```
@Override
public Iterable<Ingredient> findAll() {
  return jdbcTemplate.query(
      "select id, name, type from Ingredient",
      this::mapRowToIngredient);
}

@Override
public Optional<Ingredient> findById(String id) {
  List<Ingredient> results = jdbcTemplate.query(
      "select id, name, type from Ingredient where id=?",
      this::mapRowToIngredient,
      id);
  return results.size() == 0 ?
          Optional.empty() :
          Optional.of(results.get(0));
}
private Ingredient mapRowToIngredient(ResultSet row, int rowNum)
    throws SQLException{
  return new Ingredient(
      row.getString("id"),
      row.getString("name"),
      Ingredient.Type.valueOf(row.getString("type")));
}
```

Both `findAll()` and `findById()` use `JdbcTemplate` in a similar way. The `findAll()` method, expecting to return a collection of objects, uses `JdbcTemplate` 's `query()` method. The `query()` method accepts the SQL for the query as well as an implementation of Spring's `RowMapper` for the purpose of mapping each row in the result set to an object. `query()` also accepts as its final argument(s) a list of any parameters required in the query. But, in this case, there aren't any required parameters.

In contrast, the `findById()` method will need to include a `where` clause in its query to compare the value of the `id` column with the value of the `id` parameter passed into the method. Therefore, the call to `query()` includes, as its final parameter, the `id` parameter. When the query is performed, the `?` will be replaced with this value.

As shown in listing 3.5, the `RowMapper` parameter for both `findAll()` and `findById()` is given as a method reference to the `mapRowToIngredient()` method. Java's method references and lambdas are convenient when working with `JdbcTemplate` as an alternative to an explicit `RowMapper` implementation. But if for some reason you want or need an explicit `RowMapper`, then the following implementation of `findById()` shows how to do that:

```
@Override
public Ingredient findById(String id) {
  return jdbcTemplate.queryForObject(
      "select id, name, type from Ingredient where id=?",
      new RowMapper<Ingredient>() {
        public Ingredient mapRow(ResultSet rs, int rowNum)
            throws SQLException {
          return new Ingredient(
              rs.getString("id"),
              rs.getString("name"),
```

```
                Ingredient.Type.valueOf(rs.getString("type")));
          };
      }, id);
}
```

Reading data from a database is only part of the story. At some point, data must be written to the database so that it can be read. So let's see about implementing the `save()` method.

INSERTING A ROW

`JdbcTemplate` 's `update()` method can be used for any query that writes or updates data in the database. And, as shown in the following listing, it can be used to insert data into the database.

Listing 3.6 Inserting data with `JdbcTemplate`

```
@Override
public Ingredient save(Ingredient ingredient) {
  jdbcTemplate.update(
      "insert into Ingredient (id, name, type) values (?, ?, ?)",
      ingredient.getId(),
      ingredient.getName(),
      ingredient.getType().toString());
  return ingredient;
}
```

Because it isn't necessary to map `ResultSet` data to an object, the `update()` method is much simpler than `query()`. It only requires a `String` containing the SQL to perform as well as values to assign to any query parameters. In this case, the query has three parameters, which correspond to the final three parameters of the `save()` method, providing the ingredient's ID, name, and type.

With `JdbcIngredientRepository` complete, you can now inject it into `DesignTacoController` and use it to provide a list of `Ingredient` objects instead of using hardcoded values (as you did in chapter 2). The changes to `DesignTacoController` are shown next.

Listing 3.7 Injecting and using a repository in the controller

```
@Controller
@RequestMapping("/design")
@SessionAttributes("tacoOrder")
public class DesignTacoController {

  private final IngredientRepository ingredientRepo;

  @Autowired
  public DesignTacoController(
        IngredientRepository ingredientRepo) {
    this.ingredientRepo = ingredientRepo;
  }

  @ModelAttribute
  public void addIngredientsToModel(Model model) {
    Iterable<Ingredient> ingredients = ingredientRepo.findAll();
    Type[] types = Ingredient.Type.values();
    for (Type type : types) {
      model.addAttribute(type.toString().toLowerCase(),
          filterByType(ingredients, type));
    }
  }


  ...
}
```

The `addIngredientsToModel()` method uses the injected `IngredientRepository`'s `findAll()` method to fetch all ingredients from the database. It then filters them into distinct ingredient types before adding them to the model.

Now that we have an `IngredientRepository` to fetch `Ingredient` objects from, we can also simplify the `IngredientByIdConverter` that we created in chapter 2, replacing its hard-coded `Map` of `Ingredient` objects with a simple call to the `IngredientRepository.findById()` method:

Listing 3.8 Simplifying IngredientByIdConverter

```
package tacos.web;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.core.convert.converter.Converter;
import org.springframework.stereotype.Component;

import tacos.Ingredient;
import tacos.data.IngredientRepository;

@Component
public class IngredientByIdConverter implements Converter<String, Ingredient> {

  private IngredientRepository ingredientRepo;

  @Autowired
  public IngredientByIdConverter(IngredientRepository ingredientRepo) {
    this.ingredientRepo = ingredientRepo;
  }

  @Override
  public Ingredient convert(String id) {
    return ingredientRepo.findById(id).orElse(null);
  }

}
```

You're almost ready to fire up the application and try these changes out. But before you can start reading data from the `Ingredient` table referenced in the queries, you should probably create that table and populate it with some ingredient data.

3.1.3 Defining a schema and preloading data

Aside from the `Ingredient` table, you'll also going to need some tables that hold order and design information. Figure 3.1 illustrates the tables you'll need, as well as the relationships between those tables.

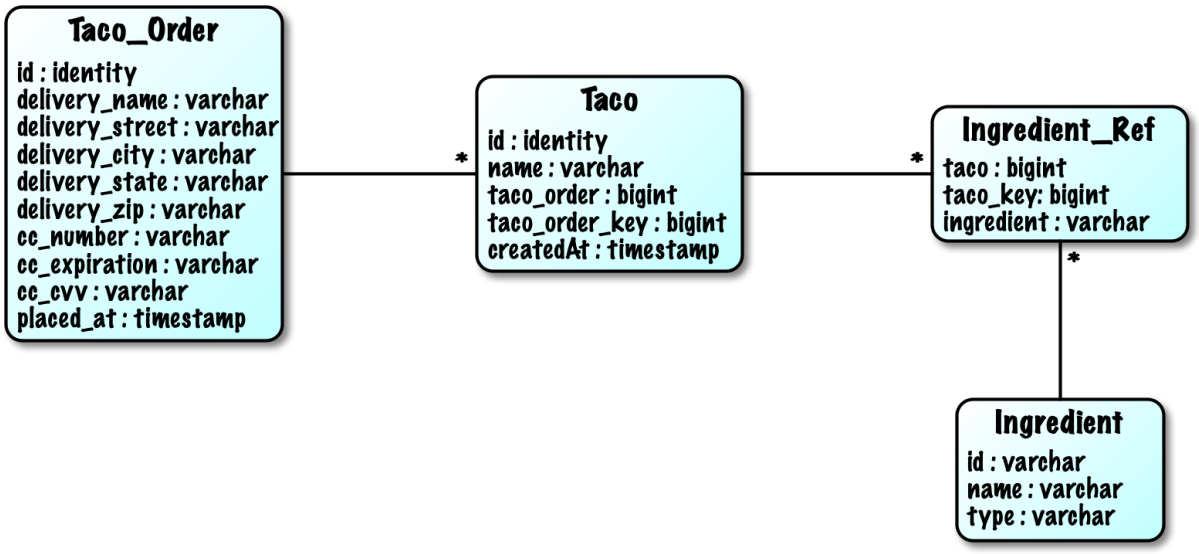

Figure 3.1 The tables for the Taco Cloud schema

The tables in figure 3.1 serve the following purposes:

- `Taco_Order` — Holds essential order details
- `Taco` — Holds essential information about a taco design
- `Ingredient_Ref` — Contains one or more rows for each row in `Taco`, mapping the taco to the ingredients for that taco
- `Ingredient` — Holds ingredient information

In our application, a `Taco` can't exist outside of the context of a `Taco_Order`. Thus, `Taco_Order` and `Taco` are considered members of an aggregate where `Taco_Order` is the aggregate root. `Ingredient` objects, on the other hand, are sole members of their own aggregate and are referenced by `Taco` by way of `Ingredient_Ref`.

NOTE **Aggregates and aggregate roots are core concepts of Domain Driven Design, a design approach that promotes the idea that the structure and language of software code should match the business domain. Although we're applying a little Domain Driven Design (DDD) in the Taco Cloud domain objects, there's much more to DDD than aggregates and aggregate roots. For more on this subject, read the seminal work on the subject, "Domain-Driven Design: Tackling Complexity in the Heart of Software", by Eric Evans.**

The next listing shows the SQL that creates the tables.

Listing 3.9 Defining the Taco Cloud schema

```
create table if not exists Taco_Order (
  id identity,
  delivery_Name varchar(50) not null,
  delivery_Street varchar(50) not null,
  delivery_City varchar(50) not null,
  delivery_State varchar(2) not null,
  delivery_Zip varchar(10) not null,
  cc_number varchar(16) not null,
  cc_expiration varchar(5) not null,
  cc_cvv varchar(3) not null,
  placed_at timestamp not null
);

create table if not exists Taco (
  id identity,
  name varchar(50) not null,
  taco_order bigint not null,
  taco_order_key bigint not null,
  created_at timestamp not null
);

create table if not exists Ingredient_Ref (
  ingredient varchar(4) not null,
  taco bigint not null,
  taco_key bigint not null
);


create table if not exists Ingredient (
  id varchar(4) not null,
  name varchar(25) not null,
  type varchar(10) not null
);


alter table Taco
    add foreign key (taco_order) references Taco_Order(id);
alter table Ingredient_Ref
    add foreign key (ingredient) references Ingredient(id);
```

The big question is where to put this schema definition. As it turns out, Spring Boot answers that question.

If there's a file named schema.sql in the root of the application's classpath, then the SQL in that file will be executed against the database when the application starts. Therefore, you should place the contents of listing 3.8 in your project as a file named schema.sql in the src/main/resources folder.

You also need to preload the database with some ingredient data. Fortunately, Spring Boot will also execute a file named data.sql from the root of the classpath when the application starts. Therefore, you can load the database with ingredient data using the insert statements in the next listing, placed in src/main/resources/data.sql.

Listing 3.10 Preloading the database with data.sql

```
delete from Ingredient_Ref;
delete from Taco;
delete from Taco_Order;

delete from Ingredient;
insert into Ingredient (id, name, type)
                values ('FLTO', 'Flour Tortilla', 'WRAP');
insert into Ingredient (id, name, type)
                values ('COTO', 'Corn Tortilla', 'WRAP');
insert into Ingredient (id, name, type)
                values ('GRBF', 'Ground Beef', 'PROTEIN');
insert into Ingredient (id, name, type)
                values ('CARN', 'Carnitas', 'PROTEIN');
insert into Ingredient (id, name, type)
                values ('TMTO', 'Diced Tomatoes', 'VEGGIES');
insert into Ingredient (id, name, type)
                values ('LETC', 'Lettuce', 'VEGGIES');
insert into Ingredient (id, name, type)
                values ('CHED', 'Cheddar', 'CHEESE');
insert into Ingredient (id, name, type)
                values ('JACK', 'Monterrey Jack', 'CHEESE');
insert into Ingredient (id, name, type)
                values ('SLSA', 'Salsa', 'SAUCE');
insert into Ingredient (id, name, type)
                values ('SRCR', 'Sour Cream', 'SAUCE');
```

Even though you've only developed a repository for ingredient data, you can fire up the Taco Cloud application at this point and visit the design page to see `JdbcIngredientRepository` in action. Go ahead … give it a try. When you get back, you'll write the repositories for persisting `Taco` and `TacoOrder` data.

3.1.4 Inserting data

You've already had a glimpse into how to use `JdbcTemplate` to write data to the database. The `save()` method in `JdbcIngredientRepository` used the `update()` method of `JdbcTemplate` to save `Ingredient` objects to the database.

Although that was a good first example, it was perhaps a bit too simple. As you'll soon see, saving data can be more involved than what `JdbcIngredientRepository` needed.

In our design, `TacoOrder` and `Taco` are part of an aggregate in which `TacoOrder` is the aggregate root. In other words, `Taco` objects don't exist outside of the context of a `TacoOrder`. So, for now, we only need to define a repository to persist `TacoOrder` objects and, in turn, `Taco` objects along with them. Such a repository can be defined in a `OrderRepository` interface like this:

```
package tacos.data;

import java.util.Optional;

import tacos.TacoOrder;

public interface OrderRepository {
```

```
  TacoOrder save(TacoOrder order);

}
```

Seems simple enough, right? Not so quick. When you save a `TacoOrder`, you also must save the `Taco` objects that go with it. And when you save the `Taco` object, you'll also need to save an object that represents the link between the `Taco` and each `Ingredient` that makes up the taco. The `IngredientRef` class defines that linking between `Taco` and `Ingredient`:

```
package tacos;

import lombok.Data;

@Data
public class IngredientRef {

  private final String ingredient;

}
```

Suffice it to say that the `save()` method will be a bit more interesting than the corresponding method you created earlier for saving a humble `Ingredient` object.

Another thing that the `save()` method will need to do is determine what ID is assigned to the order once it has been saved. Per the schema, the `id` property on the `Taco_Order` table is an `identity`, meaning that the database will determine the value automatically. But if the database determines the value for you, then you will need to know what that value is so that it can be returned in the `TacoOrder` object returned from the `save()` method. Fortunately, Spring offers a helpful `GeneratedKeyHolder` type that can help with that. But it involves working with a prepared statement, as shown in the following implementation of the `save()` method:

```
package tacos.data;

import java.sql.Types;
import java.util.Arrays;
import java.util.Date;
import java.util.List;
import java.util.Optional;

import org.springframework.asm.Type;
import org.springframework.jdbc.core.JdbcOperations;
import org.springframework.jdbc.core.PreparedStatementCreator;
import org.springframework.jdbc.core.PreparedStatementCreatorFactory;
import org.springframework.jdbc.support.GeneratedKeyHolder;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;

import tacos.IngredientRef;
import tacos.Taco;
import tacos.TacoOrder;

@Repository
public class JdbcOrderRepository implements OrderRepository {

  private JdbcOperations jdbcOperations;

  public JdbcOrderRepository(JdbcOperations jdbcOperations) {
```

```
    this.jdbcOperations = jdbcOperations;
  }

  @Override
  @Transactional
  public TacoOrder save(TacoOrder order) {
    PreparedStatementCreatorFactory pscf =
      new PreparedStatementCreatorFactory(
        "insert into Taco_Order "
        + "(delivery_name, delivery_street, delivery_city, "
        + "delivery_state, delivery_zip, cc_number, "
        + "cc_expiration, cc_cvv, placed_at) "
        + "values (?,?,?,?,?,?,?,?,?)",
        Types.VARCHAR, Types.VARCHAR, Types.VARCHAR,
        Types.VARCHAR, Types.VARCHAR, Types.VARCHAR,
        Types.VARCHAR, Types.VARCHAR, Types.TIMESTAMP
    );
    pscf.setReturnGeneratedKeys(true);

    order.setPlacedAt(new Date());
    PreparedStatementCreator psc =
        pscf.newPreparedStatementCreator(
            Arrays.asList(
                order.getDeliveryName(),
                order.getDeliveryStreet(),
                order.getDeliveryCity(),
                order.getDeliveryState(),
                order.getDeliveryZip(),
                order.getCcNumber(),
                order.getCcExpiration(),
                order.getCcCVV(),
                order.getPlacedAt()));

    GeneratedKeyHolder keyHolder = new GeneratedKeyHolder();
    jdbcOperations.update(psc, keyHolder);
    long orderId = keyHolder.getKey().longValue();
    order.setId(orderId);

    List<Taco> tacos = order.getTacos();
    int i=0;
    for (Taco taco : tacos) {
      saveTaco(orderId, i++, taco);
    }

    return order;
  }
}
```

There appears to be a lot going on in the `save()` method. But if you break it down, there are only a handful of significant steps. First, you create a `PreparedStatementCreatorFactory` that describes the `insert` query along with the types of the query's input fields. Because you'll later need to fetch the saved order's ID, you also will need to call `setReturnGeneratedKeys(true)`.

After defining the `PreparedStatementCreatorFactory`, you use it to create a `PreparedStatementCreator`, passing in the values from the `TacoOrder` object that will be persisted. The last field given to the `PreparedStatementCreator` is the date that the order is created, which you'll also need to set on the `TacoOrder` object itself so that the returned `TacoOrder` will have that information available.

Now that you have a `PreparedStatementCreator` in hand, you're ready to actually save the

order data by calling the `update()` method on `JdbcTemplate`, passing in the `PreparedStatementCreator` and a `GeneratedKeyHolder`. After the order data has been saved, the `GeneratedKeyHolder` will contain the value of the `id` field as assigned by the database and should be copied into the `TacoOrder` object's `id` property.

At this point, the order has been saved, but you need to also save the `Taco` objects associated with the order. You can do that by calling `saveTaco()` for each `Taco` in the order.

The `saveTaco()` method is quite similar to the `save()` method, as you can see here:

```
private long saveTaco(Long orderId, int orderKey, Taco taco) {
  taco.setCreatedAt(new Date());
  PreparedStatementCreatorFactory pscf =
          new PreparedStatementCreatorFactory(
      "insert into Taco "
      + "(name, created_at, taco_order, taco_order_key) "
      + "values (?, ?, ?, ?)",
      Types.VARCHAR, Types.TIMESTAMP, Type.LONG, Type.LONG
  );
  pscf.setReturnGeneratedKeys(true);

  PreparedStatementCreator psc =
      pscf.newPreparedStatementCreator(
          Arrays.asList(
              taco.getName(),
              taco.getCreatedAt(),
              orderId,
              orderKey));

  GeneratedKeyHolder keyHolder = new GeneratedKeyHolder();
  jdbcOperations.update(psc, keyHolder);
  long tacoId = keyHolder.getKey().longValue();
  taco.setId(tacoId);

  saveIngredientRef(tacoId, taco.getIngredients());

  return tacoId;
}
```

Step by step, `saveTaco()` mirrors the structure of `save()`, albeit for `Taco` data instead of `TacoOrder` data. In the end, it makes a call to `saveIngredientRef()` to create a row in the `Ingredient_Def` table to link the `Taco` row to an `Ingredient` row. The `saveIngredientRef()` method looks like this:

```
private void saveIngredientRef(
    long tacoId, List<IngredientRef> ingredientRefs) {
  int key = 0;
  for (IngredientRef ingredientRef : ingredientRefs) {
    jdbcOperations.update(
        "insert into Ingredient_Ref (ingredient, taco, taco_key) "
        + "values (?, ?, ?)",
        ingredientRef.getIngredient(), tacoId, key++);
  }
}
```

Thankfully, the `saveIngredientRef()` method is much simpler. It cycles through a list of `IngredientRef` objects, saving each into the `Ingredient_Ref` table. It also has a local `key`

variable which is used as an index to ensure that the ordering of the ingredients stays intact.

All that's left to do with `OrderRepository` is to inject it into `OrderController` and use it when saving an order. The following listing shows the changes necessary for injecting the repository.

Listing 3.11 Injecting and using `OrderRepository`

```
package tacos.web;
import javax.validation.Valid;

import org.springframework.stereotype.Controller;
import org.springframework.validation.Errors;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.SessionAttributes;
import org.springframework.web.bind.support.SessionStatus;

import tacos.TacoOrder;
import tacos.data.OrderRepository;

@Controller
@RequestMapping("/orders")
@SessionAttributes("tacoOrder")
public class OrderController {

  private OrderRepository orderRepo;

  public OrderController(OrderRepository orderRepo) {
    this.orderRepo = orderRepo;
  }

 ...
  @PostMapping
  public String processOrder(@Valid TacoOrder order, Errors errors, SessionStatus sessionStatus) {
    if (errors.hasErrors()) {
      return "orderForm";
    }

    orderRepo.save(order);
    sessionStatus.setComplete();

    return "redirect:/";
  }

}
```

As you can see, the constructor takes an `OrderRepository` as a parameter and assigns it to an instance variable that it will use in the `processOrder()` method. Speaking of the `processOrder()` method, it has been changed to call the `save()` method on the `OrderRepository` instead of logging the `TacoOrder` object.

Spring's `JdbcTemplate` makes working with relational databases significantly simpler than with plain vanilla JDBC. But even with `JdbcTemplate` some persistence tasks are still challenging, especially when persisting nested domain objects in an aggregate. If only there were a way to work with JDBC that was even simpler.

Let's have a look at Spring Data JDBC, which makes working with JDBC insanely easy—even

when persisting aggregates.

3.2 Working with Spring Data JDBC

The Spring Data project is a rather large umbrella project comprised of several subprojects, most of which are focused on data persistence with a variety of different database types. A few of the most popular Spring Data projects include these:

- *Spring Data JDBC*— JDBC-based persistence against a relational database
- *Spring Data JPA*— JPA persistence against a relational database
- *Spring Data MongoDB*— Persistence to a Mongo document database
- *Spring Data Neo4j* — Persistence to a Neo4j graph database
- *Spring Data Redis*— Persistence to a Redis key-value store
- *Spring Data Cassandra* — Persistence to a Cassandra column store database

One of the most interesting and useful features provided by Spring Data for all of these projects is the ability to automatically create repositories, based on a repository specification interface. Consequently, persistence with Spring Data projects have little or no persistence logic and involve only writing one or more repository interfaces.

Let's see how to apply Spring Data JDBC to our project to simplify data persistence with JDBC. First, you'll need to add Spring Data JDBC to the project build.

3.2.1 Adding Spring Data JDBC to the build

Spring Data JDBC is available as a starter dependency for Spring Boot apps. When added to the project's pom.xml file, the starter dependency looks like this:

Listing 3.12 Adding the Spring Data JDBC dependency to the build.

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

You will no longer need the JDBC starter that gave us `JdbcTemplate`, so you can remove the starter that looks like this:

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

You'll still need a database, however, so don't remove the H2 dependency.

3.2.2 Defining repository interfaces

Fortunately, we've already created `IngredientRepository` and `OrderRepository`, so much of the work in defining our repositories is already done. But we'll need to make a subtle change to them in order to use them with Spring Data JDBC.

Spring Data will automatically generate implementations for our repository interfaces at runtime. But it will only do that for interfaces that extends one of the repository interfaces provided by Spring Data. At very least, our repository interfaces will need to extend `Repository` so that Spring Data knows to create the implementation automatically. For example, here's how you might write `IngredientRepository` such that it extends `Repository`:

```
package tacos.data;
import java.util.Optional;
import org.springframework.data.repository.Repository;
import tacos.Ingredient;

public interface IngredientRepository
         extends Repository<Ingredient, String> {

  Iterable<Ingredient> findAll();

  Optional<Ingredient> findById(String id);

  Ingredient save(Ingredient ingredient);

}
```

As you can see, the `Repository` interface is parameterized. The first parameter is the type of the object to be persisted by this repository—in this case, `Ingredient` The second parameter is the type of the persisted object's ID field. For `Ingredient` that's `String`.

While `IngredientRepository` will work as shown here by extending `Repository`, Spring Data also offers `CrudRepository` as a base interface for common operations, including the three methods we've defined in `IngredientRepository`. So, instead of extending `Repository`, it's often easier to extend `CrudRepository`, as shown here:

Listing 3.13 Defining a repository interface for persisting ingredients.

```
package tacos.data;

import org.springframework.data.repository.CrudRepository;

import tacos.Ingredient;

public interface IngredientRepository
         extends CrudRepository<Ingredient, String> {

}
```

Similarly, our `OrderRepository` can extend `CrudRepository` like this:

Listing 3.14 Defining a repository interface for persisting taco orders.

```
package tacos.data;

import org.springframework.data.repository.CrudRepository;

import tacos.TacoOrder;

public interface OrderRepository
         extends CrudRepository<TacoOrder, Long> {

}
```

In both cases, because `CrudRepository` already defines the methods you need, there's no need to explicitly define them in the `IngredientRepository` and `OrderRepository` interfaces.

And now you have your two repositories. You might be thinking that you need to write the implementations for both repositories, including the dozen methods defined in `CrudRepository`. But that's the good news about Spring Data—there's no need to write an implementation! When the application starts, Spring Data will automatically generate an implementation on the fly. This means the repositories are ready to use from the get-go. Just inject them into the controllers and you're done.

What's more, because Spring Data will automatically create implementations of these interfaces at runtime, you no longer need the explicit implementations in `JdbcIngredientRepository` and `JdbcOrderRepository`. You can delete those two classes and never look back!

3.2.3 Annotating the domain for persistence

The only other thing we'll need to do is annotate our domain classes so that Spring Data JDBC will know how to persist them. Generally speaking, this means annotating the identity properties with `@Id`--so that Spring Data will know which field represents the object's identity—and optionally annotating the class with `@Table`.

For example, the `TacoOrder` class might be annotated with `@Table` and `@Id` like this:

Listing 3.15 Preparing the Taco class for persistence.

```
package tacos;
import java.io.Serializable;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;

import javax.validation.constraints.Digits;
import javax.validation.constraints.NotBlank;
import javax.validation.constraints.Pattern;

import org.hibernate.validator.constraints.CreditCardNumber;
import org.springframework.data.annotation.Id;
import org.springframework.data.relational.core.mapping.Table;

import lombok.Data;

@Data
@Table
public class TacoOrder implements Serializable {

  private static final long serialVersionUID = 1L;

  @Id
  private Long id;

  ...

}
```

The `@Table` annotation is completely optional. By default the object is mapped to a table based on the domain class name. In this case, `TacoOrder` is mapped to a table named "Taco_Order". If that's fine for you, then you can leave the `@Table` annotation off completely or use it without parameters. But if you'd prefer to map it to a different table name, then you can specify the table name as a parameter to `@Table` like this:

```
@Table("Taco_Cloud_Order")
public class TacoOrder {
  ...
}
```

As shown here, `TacoOrder` will be mapped to a table named "Taco_Cloud_Order".

As for the `@Id` annotation, it designates the `id` property as being the identity for a `TacoOrder`. All other properties in `TacoOrder` will be mapped automatically to columns based on their property names. For example, the `deliveryName` property will be automatically mapped to the column named "delivery_name". But if you want to explicitly define the column name mapping, you could annotate the property with `@Column` like this:

```
@Column("customer_name")
@NotBlank(message="Delivery name is required")
private String deliveryName;
```

In this case, `@Column` is specifying that the `deliveryName` property will be mapped to the

column whose name is "customer_name".

You'll also need to apply `@Table` and `@Id` to the other domain classes. This includes `@Ingredient`…

Listing 3.16 Preparing the Ingredient class for persistence.

```
package tacos;

import org.springframework.data.annotation.Id;
import org.springframework.data.domain.Persistable;
import org.springframework.data.relational.core.mapping.Table;

import lombok.AccessLevel;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@Table
@AllArgsConstructor
@NoArgsConstructor(access=AccessLevel.PRIVATE, force=true)
public class Ingredient implements Persistable<String> {

  @Id
  private String id;

  ...

}
```

…and `Taco`…

Listing 3.17 Preparing the Taco class for persistence.

```
package tacos;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;

import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

import org.springframework.data.annotation.Id;
import org.springframework.data.relational.core.mapping.Table;

import lombok.Data;
import lombok.EqualsAndHashCode;

@Data
@Table
public class Taco {

  @Id
  private Long id;

  ...

}
```

As for `IngredientRef`, it will be mapped automatically to the table whose name is "Ingredient_Ref", which is perfect for our application. You can annotate it with `@Table` if you

want, but it's not necessary. And the "Ingredient_Ref" table has no identify column, so there is no need to annotate anything in `IngredientRef` with `@Id`.

With these small changes, not to mention the complete removal of the `JdbcIngredientRepository` and `JdbcOrderRepository` classes, you now have a lot less persistence code. Even so, it still does everything that the repositories using `JdbcTemplate` did. In fact, they have potential for doing even more, because the two repository interfaces extends `CrudRepository` which offer a dozen or so operations for creating, reading, updating, and deleting objects.

3.2.4 Preloading data with CommandLineRunner

When working with `JdbcTemplate`, we preloaded the `Ingredient` data at application startup using `data.sql`, which was executed against the database when the data source bean was created. That same approach will work with Spring Data JDBC. In fact, it will work with any persistence mechanism for which the backing database is a relational database. But let's see another way of populating a database at startup that offers a bit more flexibility.

Spring Boot offers two useful interfaces for executing logic when an application starts up: `CommandLineRunner` and `ApplicationRunner`. These two interfaces are quite similar. Both are functional interfaces that require that a single `run()` method be implemented. When the application starts up, any beans in the application context that implement `CommandLineRunner` or `ApplicationRunner` will have their `run()` methods invoked after the application context and all beans are wired up, but before anything else happens. This gives a convenient place for data to be loaded into the database.

Because both `CommandLineRunner` and `ApplicationRunner` are functional interfaces, they can easily be declared as beans in a configuration class using a `@Bean`-annotated method that returns a lambda function. For example, here's how you might create a data-loading `CommandLineRunner` bean:

```
@Bean
public CommandLineRunner dataLoader(IngredientRepository repo) {
  return args -> {
    repo.save(new Ingredient("FLTO", "Flour Tortilla", Type.WRAP));
    repo.save(new Ingredient("COTO", "Corn Tortilla", Type.WRAP));
    repo.save(new Ingredient("GRBF", "Ground Beef", Type.PROTEIN));
    repo.save(new Ingredient("CARN", "Carnitas", Type.PROTEIN));
    repo.save(new Ingredient("TMTO", "Diced Tomatoes", Type.VEGGIES));
    repo.save(new Ingredient("LETC", "Lettuce", Type.VEGGIES));
    repo.save(new Ingredient("CHED", "Cheddar", Type.CHEESE));
    repo.save(new Ingredient("JACK", "Monterrey Jack", Type.CHEESE));
    repo.save(new Ingredient("SLSA", "Salsa", Type.SAUCE));
    repo.save(new Ingredient("SRCR", "Sour Cream", Type.SAUCE));
  };
}
```

Here, the `IngredientRepository` is injected into the bean method and used within the lambda

to create `Ingredient` objects. The `run()` method of `CommandLineRunner` accepts a single parameter that is a `String` varargs containing all of the command line arguments for the running application. We don't need those to load ingredients into the database, so the `args` parameter is ignored.

Alternatively, we could have defined the data loader bean as a lambda implementation of `ApplicationRunner` like this:

```
@Bean
public ApplicationRunner dataLoader(IngredientRepository repo) {
  return args -> {
    repo.save(new Ingredient("FLTO", "Flour Tortilla", Type.WRAP));
    repo.save(new Ingredient("COTO", "Corn Tortilla", Type.WRAP));
    repo.save(new Ingredient("GRBF", "Ground Beef", Type.PROTEIN));
    repo.save(new Ingredient("CARN", "Carnitas", Type.PROTEIN));
    repo.save(new Ingredient("TMTO", "Diced Tomatoes", Type.VEGGIES));
    repo.save(new Ingredient("LETC", "Lettuce", Type.VEGGIES));
    repo.save(new Ingredient("CHED", "Cheddar", Type.CHEESE));
    repo.save(new Ingredient("JACK", "Monterrey Jack", Type.CHEESE));
    repo.save(new Ingredient("SLSA", "Salsa", Type.SAUCE));
    repo.save(new Ingredient("SRCR", "Sour Cream", Type.SAUCE));
  };
}
```

The key difference between `CommandLineRunner` and `ApplicationRunner` is in the paramter passed to the respective `run()` methods. `CommandLineRunner` accepts a `String` varargs, which is a raw representation of what arguments passed on the command line. But `ApplicationRunner` accepts an `ApplicationArguments` parameter which offers methods for accessing the arguments as parsed components of the command line.

For example, suppose that we want our application to accept a command line with arguments such as "--version 1.2.3" and needed to consider that argument in our loader bean. If using a `CommandLineRunner`, we'd need to search the array for "--version" and then take the very next value from the array. But with `ApplicationRunner`, we can query the given `ApplicationArguments` for the "--version" argument like this:

```
public ApplicationRunner dataLoader(IngredientRepository repo) {
  return args -> {
    List<String> version = args.getOptionValues("version");
    ...
  };
}
```

The `getOptionValues()` method returns a `List<String>` to allow for the option argument to be specified multiple times.

In either case of `CommandLineRunner` or `ApplicationRunner`, however, we don't need command line arguments to load data. So the `args` parameter is ignored in our data loader bean.

What's nice about using `CommandLineRunner` or `ApplicationRunner` to do an initial data load is that they are using the repositories to create the persisted objects instead of an SQL script. This

means that they'll work equally well for relational databases or non-relational databases. This will come in handy in the next chapter when we see how to use Spring Data to persist to non-relational databases.

But before we do that, let's have a look at another Spring Data project for persisting data in relational databases: Spring Data JPA.

3.3 Persisting data with Spring Data JPA

While Spring Data JDBC makes easy work of persisting data, the Java Persistence API (JPA) is another popular option for working with data in a relational database. Spring Data JPA offers a similar approach to persistence with JPA as Spring Data JDBC gave us for JDBC.

To see how Spring Data works, you're going to start over, replacing the JDBC-based repositories from earlier in this chapter with repositories created by Spring Data JPA. But first, you need to add Spring Data JPA to the project build.

3.3.1 Adding Spring Data JPA to the project

Spring Data JPA is available to Spring Boot applications with the JPA starter. This starter dependency not only brings in Spring Data JPA, but also transitively includes Hibernate as the JPA implementation:

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

If you want to use a different JPA implementation, then you'll need to, at least, exclude the Hibernate dependency and include the JPA library of your choice. For example, to use EclipseLink instead of Hibernate, you'll need to alter the build as follows:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
  <exclusions>
    <exclusion>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-core</artifactId>
    </exclusion>
  </exclusions>
</dependency>
<dependency>
  <groupId>org.eclipse.persistence</groupId>
  <artifactId>org.eclipse.persistence.jpa</artifactId>
  <version>2.7.6</version>
</dependency>
```

Note that there may be other changes required, depending on your choice of JPA implementation. Consult the documentation for your chosen JPA implementation for details. Now let's revisit your domain objects and annotate them for JPA persistence.

3.3.2 Annotating the domain as entities

As you've already seen with Spring Data JDBC, Spring Data does some amazing things when it comes to creating repositories. But unfortunately, it doesn't help much when it comes to annotating your domain objects with JPA mapping annotations. You'll need to open up the `Ingredient`, `Taco`, and `TacoOrder` classes and throw in a few annotations. First up is the `Ingredient` class.

Listing 3.18 Annotating `Ingredient` for JPA persistence

```
package tacos;

import javax.persistence.Entity;
import javax.persistence.Id;

import lombok.AccessLevel;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@Entity
@AllArgsConstructor
@NoArgsConstructor(access=AccessLevel.PRIVATE, force=true)
public class Ingredient {

  @Id
  private String id;
  private String name;
  private Type type;

  public static enum Type {
    WRAP, PROTEIN, VEGGIES, CHEESE, SAUCE
  }

}
```

In order to declare this as a JPA entity, `Ingredient` must be annotated with `@Entity`. And its `id` property must be annotated with `@Id` to designate it as the property that will uniquely identify the entity in the database.

In addition to the JPA-specific annotations, you'll also note that you've added a `@NoArgsConstructor` annotation at the class level. JPA requires that entities have a no-arguments constructor, so Lombok's `@NoArgsConstructor` does that for you. You don't want to be able to use it, though, so you make it `private` by setting the access attribute to `AccessLevel.PRIVATE`. And because there are `final` properties that must be set, you also set the `force` attribute to `true`, which results in the Lombok-generated constructor setting them to a default value of `null`, `0`, or `false`, depending on the property type.

You also will add an `@AllArgsConstructor` to make it easy to create an `Ingredient` object with all properties initialized.

Now let's move on to the `Taco` class and see how to annotate it as a JPA entity.

Listing 3.19 Annotating `Taco` as an entity

```
package tacos;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.ManyToMany;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

import lombok.Data;

@Data
@Entity
public class Taco {

  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  private Long id;

  @NotNull
  @Size(min=5, message="Name must be at least 5 characters long")
  private String name;

  private Date createdAt = new Date();

  @Size(min=1, message="You must choose at least 1 ingredient")
  @ManyToMany()
  private List<Ingredient> ingredients = new ArrayList<>();

  public void addIngredient(Ingredient ingredient) {
    this.ingredients.add(ingredient);
  }

}
```

As with `Ingredient`, the `Taco` class is now annotated with `@Entity` and has its `id` property annotated with `@Id`. Because you're relying on the database to automatically generate the ID value, you also annotate the `id` property with `@GeneratedValue`, specifying a `strategy` of `AUTO`.

To declare the relationship between a `Taco` and its associated `Ingredient` list, you annotate ingredients with `@ManyToMany`. A `Taco` can have many `Ingredient` objects, and an `Ingredient` can be a part of many `Taco` s.

You'll also notice that there's a new method, `createdAt()`, which is annotated with `@PrePersist`. You'll use this to set the `createdAt` property to the current date and time before Taco is persisted. Finally, let's annotate the `TacoOrder` object as an entity. The next listing shows the new `TacoOrder` class.

Listing 3.20 Annotating `TacoOrder` as a JPA entity

```
package tacos;
import java.io.Serializable;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;

import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import javax.validation.constraints.Digits;
import javax.validation.constraints.NotBlank;
import javax.validation.constraints.Pattern;

import org.hibernate.validator.constraints.CreditCardNumber;

import lombok.Data;

@Data
@Entity
public class TacoOrder implements Serializable {

  private static final long serialVersionUID = 1L;

  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  private Long id;

  private Date placedAt = new Date();

  ...

  @OneToMany(cascade = CascadeType.ALL)
  private List<Taco> tacos = new ArrayList<>();

  public void addTaco(Taco taco) {
    this.tacos.add(taco);
  }

}
```

As you can see, the changes to `TacoOrder` closely mirror the changes to `Taco`.

3.3.3 Declaring JPA repositories

When you created the `JdbcTemplate`-based versions of the repositories, you explicitly declared the methods you wanted the repository to provide. But with Spring Data JDBC you were able to dismiss the explicit implementation classes and instead extend the `CrudRepository` interface. As it turns out, `CrudRepository` works equally well for Spring Data JPA. For example, here's the new `IngredientRepository` interface:

```
package tacos.data;

import org.springframework.data.repository.CrudRepository;

import tacos.Ingredient;
```

```
public interface IngredientRepository
         extends CrudRepository<Ingredient, String> {

}
```

In fact, the `IngredientRepository` interface we'll use with Spring Data JPA is identical to the one we defined for use with Spring Data JDBC. The `CrudRepository` interface is commonly used across many of Spring Data's projects, regardless of the underlying persistence mechanism. Similarly, you can define `OrderRepository` for Spring Data JPA the same as it was for Spring Data JDBC:

```
package tacos.data;

import org.springframework.data.repository.CrudRepository;

import tacos.TacoOrder;

public interface OrderRepository
         extends CrudRepository<TacoOrder, Long> {

}
```

The methods provided by `CrudRepository` are great for general-purpose persistence of entities. But what if you have some requirements beyond basic persistence? Let's see how to customize the repositories to perform queries unique to your domain.

3.3.4 Customizing repositories

Imagine that in addition to the basic CRUD operations provided by `CrudRepository`, you also need to fetch all the orders delivered to a given ZIP code. As it turns out, this can easily be addressed by adding the following method declaration to `OrderRepository`:

```
List<TacoOrder> findByDeliveryZip(String deliveryZip);
```

When generating the repository implementation, Spring Data examines any methods in the repository interface, parses the method name, and attempts to understand the method's purpose in the context of the persisted object (a `TacoOrder`, in this case). In essence, Spring Data defines a sort of miniature domain-specific language (DSL) where persistence details are expressed in repository method signatures.

Spring Data knows that this method is intended to find `Orders`, because you've parameterized `CrudRepository` with `TacoOrder`. The method name, `findByDeliveryZip()`, makes it clear that this method should find all `TacoOrder` entities by matching their `deliveryZip` property with the value passed in as a parameter to the method.

The `findByDeliveryZip()` method is simple enough, but Spring Data can handle even more-interesting method names as well. Repository methods are composed of a verb, an optional subject, the word *By*, and a predicate. In the case of `findByDeliveryZip()`, the verb is *find* and the predicate is *DeliveryZip*; the subject isn't specified and is implied to be a `TacoOrder`.

Let's consider another, more complex example. Suppose that you need to query for all orders delivered to a given ZIP code within a given date range. In that case, the following method, when added to `OrderRepository`, might prove useful:

```
List<TacoOrder> readOrdersByDeliveryZipAndPlacedAtBetween(
      String deliveryZip, Date startDate, Date endDate);
```

Figure 3.2 illustrates how Spring Data parses and understands the `readOrdersByDeliveryZipAndPlacedAtBetween()` method when generating the repository implementation. As you can see, the verb in `readOrdersByDeliveryZipAndPlacedAtBetween()` is read. Spring Data also understands find, read, and get as synonymous for fetching one or more entities. Alternatively, you can also use count as the verb if you only want the method to return an int with the count of matching entities.

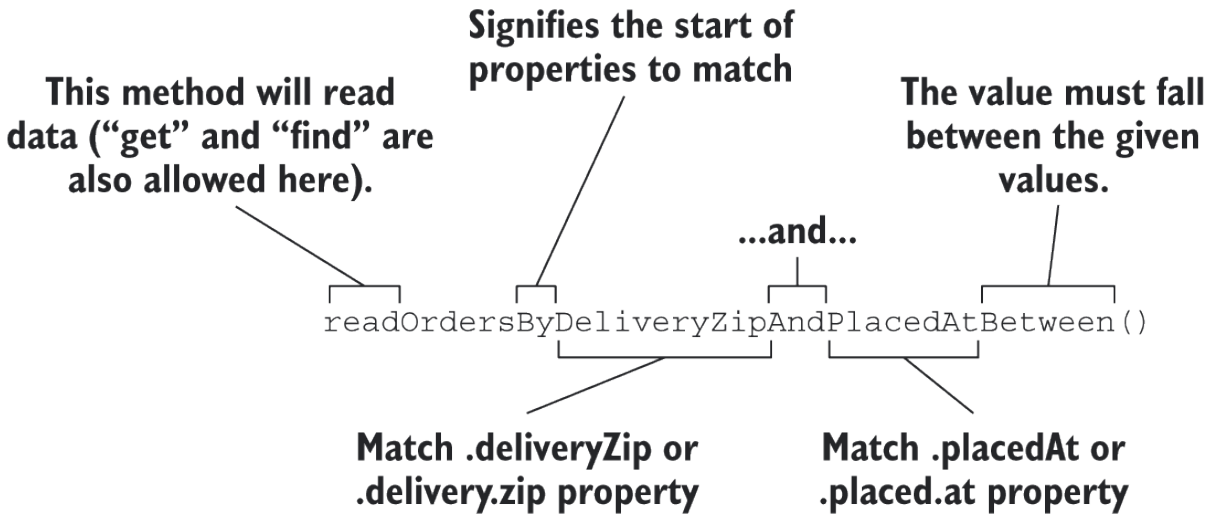

Figure 3.2 Spring Data parses repository method signatures to determine the query that should be performed.

Although the subject of the method is optional, here it says `Orders`. Spring Data ignores most words in a subject, so you could name the method `readPuppiesBy…` and it would still find `TacoOrder` entities, as that is the type that `CrudRepository` is parameterized with.

The predicate follows the word `By` in the method name and is the most interesting part of the method signature. In this case, the predicate refers to two `TacoOrder` properties: `deliveryZip` and `placedAt`. The `deliveryZip` property must be equal to the value passed into the first parameter of the method. The keyword `Between` indicates that the value of `deliveryZip` must fall between the values passed into the last two parameters of the method.

In addition to an implicit `Equals` operation and the `Between` operation, Spring Data method signatures can also include any of these operators:

- `IsAfter`, `After`, `IsGreaterThan`, `GreaterThan`
- `IsGreaterThanEqual`, `GreaterThanEqual`
- `IsBefore`, `Before`, `IsLessThan`, `LessThan`
- `IsLessThanEqual`, `LessThanEqual`
- `IsBetween`, `Between`
- `IsNull`, `Null`
- `IsNotNull`, `NotNull`
- `IsIn`, `In`
- `IsNotIn`, `NotIn`
- `IsStartingWith`, `StartingWith`, `StartsWith`
- `IsEndingWith`, `EndingWith`, `EndsWith`
- `IsContaining`, `Containing`, `Contains`
- `IsLike`, `Like`
- `IsNotLike`, `NotLike`
- `IsTrue`, `True`
- `IsFalse`, `False`
- `Is`, `Equals`
- `IsNot`, `Not`
- `IgnoringCase`, `IgnoresCase`

As alternatives for `IgnoringCase` and `IgnoresCase`, you can place either `AllIgnoringCase` or `AllIgnoresCase` on the method to ignore case for all `String` comparisons. For example, consider the following method:

```
List<TacoOrder> findByDeliveryToAndDeliveryCityAllIgnoresCase(
        String deliveryTo, String deliveryCity);
```

Finally, you can also place `OrderBy` at the end of the method name to sort the results by a specified column. For example, to order by the `deliveryTo` property:

```
List<TacoOrder> findByDeliveryCityOrderByDeliveryTo(String city);
```

Although the naming convention can be useful for relatively simple queries, it doesn't take much imagination to see that method names could get out of hand for more-complex queries. In that case, feel free to name the method anything you want and annotate it with `@Query` to explicitly specify the query to be performed when the method is called, as this example shows:

```
@Query("Order o where o.deliveryCity='Seattle'")
List<TacoOrder> readOrdersDeliveredInSeattle();
```

In this simple usage of `@Query`, you ask for all orders delivered in Seattle. But you can use `@Query` to perform virtually any JPA query you can dream up, even when it's difficult or impossible to achieve the query by following the naming convention.

Custom query methods also work with Spring Data JDBC, but with two key differences:

- All custom query methods require `@Query`. This is because, unlike JPA, there's no mapping metadata to help Spring Data JDBC automatically infer the query from the method name.
- All queries specified in `@Query` must be SQL queries, not JPA queries.

In the next chapter, we'll expand our use of Spring Data to work with non-relational databases. When we do, you'll see that custom query methods work very similarly, although the query language used in `@Query` will be specific to the underlying database.

3.4 Summary

- Spring's `JdbcTemplate` greatly simplifies working with JDBC.
- `PreparedStatementCreator` and `KeyHolder` can be used together when you need to know the value of a database-generated ID.
- Spring Data JPA makes JPA persistence as easy as writing a repository interface.

4 Working with non-relational data

This chapter covers

- Persisting data to Cassandra
- Data modeling in Cassandra
- Working with document data in MongoDB

They say that variety is the spice of life.

You probably have a favorite flavor of ice cream. It's that one flavor that you go for the most often because it satisfies that creamy craving more than any other. But most people, despite having a favorite flavor, try different flavors from time to time to mix things up.

Databases are kind of like ice cream. For decades, the relational database has been the favorite flavor for storing data. But these days there are more options available than ever before. So-called "NoSQL" databases[4] offer different concepts and structures in which data can be stored. And although the choice may still be somewhat based on taste, some databases are better suited at persisting different kinds of data than others.

Fortunately, Spring Data has you covered for many of the NoSQL databases, including MongoDB, Cassandra, Couchbase, Neo4j, Redis, and many more. And fortunately, the programming model is nearly identical, regardless of which database you choose.

There's not enough space in this chapter to cover all of the databases that Spring Data supports. But to give you a sample of Spring Data's other "flavors", we'll look at two popular NoSQL databases, Cassandra and MongoDB, and see how to create repositories to persist data to them. Let's start by looking at how to create Cassandra repositories with Spring Data.

4.1 Working with Cassandra repositories

Cassandra is a distributed, high-performance, always available, eventually consistent, partitioned-column-store, NoSQL database.

That's a mouthful of adjectives to describe a database, but each one accurately speaks to the power of working with Cassandra. To put it in simpler terms, Cassandra deals in rows of data, which are written to tables, which are partitioned across one-to-many distributed nodes. No single node carries all the data, but any given row may be replicated across multiple nodes, thus eliminating any single point of failure.

Spring Data Cassandra provides automatic repository support for the Cassandra database that's quite similar to—and yet quite different from—what's offered by Spring Data JPA for relational databases. In addition, Spring Data Cassandra offers mapping annotations to map application domain types to the backing database structures.

Before we explore Cassandra any further, it's important to understand that although Cassandra shares many similar concepts with relational databases like Oracle and SQL Server, Cassandra isn't a relational database and is in many ways quite a different beast. I'll try to explain the idiosyncrasies of Cassandra as they pertain to working with Spring Data. But I encourage you to read Cassandra's own documentation (http://cassandra.apache.org/doc/latest/) for a thorough understanding of what makes Cassandra tick.

Let's get started by enabling Spring Data Cassandra in the Taco Cloud project.

4.1.1 Enabling Spring Data Cassandra

To get started using Spring Data Cassandra, you'll need to add the Spring Boot starter dependency for reactive Spring Data Cassandra. There are actually two separate Spring Data Cassandra starter dependencies to choose from: One for reactive data persistence and one for standard, non-reactive persistence.

We'll talk more about writing reactive repositories later in chapter 15. For now, though, we'll use the non-reactive starter in our build:

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

This dependency is also available from the Initializr by checking the Cassandra check box.

It's important to understand that this dependency is in lieu of the Spring Data JPA starter or Spring Data JDBC dependencies we used in the previous chapter. Instead of persisting Taco Cloud data to a relational database with JPA or JDBC, you'll be using Spring Data to persist data

to a Cassandra database. Therefore, you'll want to remove the Spring Data JPA or Spring Data JDBC starter dependencies and any relational database dependencies (such as JDBC drivers or the H2 dependency) from the build.

The Spring Data Cassandra starter dependency brings a handful of dependencies to the project, specifically the Spring Data Cassandra library. As a result of Spring Data Cassandra being in the runtime classpath, autoconfiguration for creating Cassandra repositories is triggered. This means you're able to begin writing Cassandra repositories with minimal explicit configuration.

Cassandra operates as a cluster of nodes that together act as a complete database system. If you don't already have a Cassandra cluster to work with, you can start a single-node cluster for development purposes using Docker like this:

```
$ docker network create cassandra-net
$ docker run --name my-cassandra \
             --network cassandra-net \
             -p 9042:9042
             -d cassandra:latest
```

This starts the single-node cluster and exposes the node's port (9042) on the host machine so that your application can access it.

You'll need to provide a small amount of configuration, though. At the very least, you'll need to configure the name of a key space within which your repositories will operate. To do that, you'll first need to create such a key space.

NOTE **In Cassandra, a keyspace is a grouping of tables in a Cassandra node. It's roughly analogous to how tables, views, and constraints are grouped in a relational database.**

Although it's possible to configure Spring Data Cassandra to create the key space automatically, it's typically much easier to manually create it yourself (or to use an existing key space). Using the Cassandra CQL (Cassandra Query Language) shell, you can create a key space for the Taco Cloud application. You can start the CQL shell using Docker like this:

```
$ docker run -it --network some-network --rm cassandra cqlsh some-cassandra
```

When the shell is ready, use the `create keyspace` command like this:

```
cqlsh> create keyspace tacocloud
   ... with replication={'class':'SimpleStrategy', 'replication_factor':1}
   ... and durable_writes=true;
```

Put simply, this will create a key space named `tacocloud` with simple replication and durable writes. By setting the replication factor to 2, you ask Cassandra to keep one copy of each row. The replication strategy determines how replication is handled. The `SimpleStrategy` replication

strategy is fine for single data center use (and for demo code), but you might consider the `NetworkTopologyStrategy` if you have your Cassandra cluster spread across multiple data centers. I refer you to the Cassandra documentation for more details of how replication strategies work and alternative ways of creating key spaces.

Now that you've created a key space, you need to configure the `spring.data.cassandra.keyspace-name` property to tell Spring Data Cassandra to use that key space:

```
spring:
  data:
    cassandra:
      keyspace-name: tacocloud
      schema-action: recreate-drop-unused
      local-datacenter: datacenter1
```

Here, you also set the `spring.data.cassandra.schema-action` to `recreate-drop-unused`. This setting is very useful for development purposes because it ensures that any tables and user-defined types will be dropped and recreated every time the application starts. The default value, `none`, takes no action against the schema and is useful in production settings where you'd rather not drop all tables whenever an application starts up.

Finally, the `spring.data.cassandra.local-datacenter` property identifies the name of the local datacenter for purposes of setting Cassandra's load-balancing policy. In a single-node setup, "datacenter1" is the value to use. For more information on Cassandra load-balancing policies and how to set the local datacenter, see the DataStax Cassandra driver's reference documentation[5].

These are the only properties you'll need for working with a locally running Cassandra database. In addition to these two properties, however, you may wish to set others, depending on how you've configured your Cassandra cluster.

By default, Spring Data Cassandra assumes that Cassandra is running locally and listening on port 9042. If that's not the case, as in a production setting, you may want to set the `spring.data.cassandra.contact-points` and `spring.data.cassandra.port` properties:

```
spring:
  data:
    cassandra:
      keyspace-name: tacocloud
      local-datacenter: datacenter1
      contact-points:
      - casshost-1.tacocloud.com
      - casshost-2.tacocloud.com
      - casshost-3.tacocloud.com
      port: 9043
```

Notice that the `spring.data.cassandra.contact-points` property is where you identify the hostname(s) of Cassandra. A contact point is the host where a Cassandra node is running. By

default, it's set to `localhost`, but you can set it to a list of hostnames. It will try each contact point until it's able to connect to one. This is to ensure that there's no single point of failure in the Cassandra cluster and that the application will be able to connect with the cluster through one of the given contact points.

You may also need to specify a username and password for your Cassandra cluster. This can be done by setting the `spring.data.cassandra.username` and `spring.data.cassandra.password` properties:

```
spring:
  data:
    cassandra:
       ...
      username: tacocloud
      password: s3cr3tP455w0rd
```

Now that Spring Data Cassandra is enabled and configured in your project, you're almost ready to map your domain types to Cassandra tables and write repositories. But first, let's step back and consider a few basic points of Cassandra data modeling.

These are the only properties you'll need for working with a locally running Cassandra database. In addition to these two properties, however, you may wish to set others, depending on how you've configured your Cassandra cluster.

4.1.2 Understanding Cassandra data modeling

As I mentioned, Cassandra is quite different from a relational database. Before you can start mapping your domain types to Cassandra tables, it's important to understand a few of the ways that Cassandra data modeling is different from how you might model your data for persistence in a relational database.

These are a few of the most important things to understand about Cassandra data modeling:

- Cassandra tables may have any number of columns, but not all rows will necessarily use all of those columns.
- Cassandra databases are split across multiple partitions. Any row in a given table may be managed by one or more partitions, but it's unlikely that all partitions will have all rows.
- A Cassandra table has two kinds of keys: partition keys and clustering keys. Hash operations are performed on each row's partition key to determine which partition(s) that row will be managed by. Clustering keys determine the order in which the rows are maintained within a partition (not necessarily the order that they may appear in the results of a query). Refer to Cassandra documentation [6] for a more detailed explanation of data modeling in Cassandra, including partitions, clusters, and their respective keys.
- Cassandra is highly optimized for read operations. As such, it's common and desirable for tables to be highly denormalized and for data to be duplicated across multiple tables. (For example, customer information may be kept in a customer table as well as duplicated in a table containing orders placed by customers.)

Suffice it to say that adapting the Taco Cloud domain types to work with Cassandra won't be a matter of simply swapping out a few JPA annotations for Cassandra annotations. You'll have to rethink how you model the data.

4.1.3 Mapping domain types for Cassandra persistence

In chapter 3, you marked up your domain types (`Taco`, `Ingredient`, `TacoOrder`, and so on) with annotations provided by the JPA specification. These annotations mapped your domain types as entities to be persisted to a relational database. Although those annotations won't work for Cassandra persistence, Spring Data Cassandra provides its own set of mapping annotations for a similar purpose.

Let's start with the `Ingredient` class, as it's the simplest to map for Cassandra. The new Cassandra-ready `Ingredient` class looks like this:

```
package tacos;

import org.springframework.data.cassandra.core.mapping.PrimaryKey;
import org.springframework.data.cassandra.core.mapping.Table;

import lombok.AccessLevel;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.RequiredArgsConstructor;

@Data
@AllArgsConstructor
@NoArgsConstructor(access=AccessLevel.PRIVATE, force=true)
@Table("ingredients")
public class Ingredient {

  @PrimaryKey
  private String id;
  private String name;
  private Type type;

  public static enum Type {
    WRAP, PROTEIN, VEGGIES, CHEESE, SAUCE
  }

}
```

The `Ingredient` class seems to contradict everything I said about just swapping out a few annotations. Rather than annotating the class with `@Entity` as you did for JPA persistence, it's annotated with `@Table` to indicate that ingredients should be persisted to a table named `ingredients`. And rather than annotate the `id` property with `@Id`, this time it's annotated with `@PrimaryKey`. So far, it seems that you're only swapping out a few annotations.

But don't let the `Ingredient` mapping fool you. The `Ingredient` class is one of your simplest domain types. Things get more interesting when you map the `Taco` class for Cassandra persistence.

Listing 4.1 Annotating the `Taco` class for Cassandra persistence

```
package tacos;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;
import java.util.UUID;

import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

import org.springframework.data.cassandra.core.cql.Ordering;
import org.springframework.data.cassandra.core.cql.PrimaryKeyType;
import org.springframework.data.cassandra.core.mapping.Column;
import org.springframework.data.cassandra.core.mapping.PrimaryKeyColumn;
import org.springframework.data.cassandra.core.mapping.Table;

import com.datastax.oss.driver.api.core.uuid.Uuids;

import lombok.Data;

@Data
@Table("tacos")   ❶
public class Taco {

  @PrimaryKeyColumn(type=PrimaryKeyType.PARTITIONED)   ❷
  private UUID id = Uuids.timeBased();

  @NotNull
  @Size(min = 5, message = "Name must be at least 5 characters long")
  private String name;

  @PrimaryKeyColumn(type=PrimaryKeyType.CLUSTERED,    ❸
                    ordering=Ordering.DESCENDING)
  private Date createdAt = new Date();

  @Size(min=1, message="You must choose at least 1 ingredient")
  @Column("ingredients")                              ❹
  private List<IngredientUDT> ingredients = new ArrayList<>();

  public void addIngredient(Ingredient ingredient) {
    this.ingredients.add(TacoUDRUtils.toIngredientUDT(ingredient));
  }
}
```

❶ Persist to "tacos" table

❷ Define the partition key

❸ Define the clustering key

❹ Map list to "ingredients" column.

As you can see, mapping the `Taco` class is a bit more involved. As with `Ingredient`, the `@Table` annotation is used to identify `tacos` as the name of the table that tacos should be written to. But that's the only thing similar to `Ingredient`.

The `id` property is still your primary key, but it's only one of two primary key columns. More specifically, the `id` property is annotated with `@PrimaryKeyColumn` with a `type` of `PrimaryKeyType.PARTITIONED`. This specifies that the `id` property serves as the partition key,

used to determine which Cassandra partition(s) each row of taco data will be written to.

You'll also notice that the `id` property is now a `UUID` instead of a `Long`. Although it's not required, properties that hold a generated ID value are commonly of type `UUID`. Moreover, the `UUID` is initialized with a time-based UUID value for new `Taco` objects (but which may be overridden when reading an existing `Taco` from the database).

A little further down, you see the `createdAt` property that's mapped as another primary key column. But in this case, the `type` attribute of `@PrimaryKeyColumn` is set to `PrimaryKeyType.CLUSTERED`, which designates the `createdAt` property as a clustering key. As mentioned earlier, clustering keys are used to determine the ordering of rows *within a partition*. More specifically, the ordering is set to descending order—therefore, within a given partition, newer rows appear first in the `tacos` table.

Finally, the `ingredients` property is now a `List` of `IngredientUDT` objects instead of a `List` of `Ingredient` objects. As you'll recall, Cassandra tables are highly denormalized and may contain data that's duplicated from other tables. Although the `ingredient` table will serve as the table of record for all available `ingredients`, the ingredients chosen for a taco will be duplicated in the `ingredients` column. Rather than simply reference one or more rows in the `ingredients` table, the `ingredients` property will contain full data for each chosen ingredient.

But why do you need to introduce a new `IngredientUDT` class? Why can't you just reuse the `Ingredient` class? Put simply, columns that contain collections of data, such as the `ingredients` column, must be collections of native types (integers, strings, and so on) or must be collections of user-defined types.

In Cassandra, user-defined types enable you to declare table columns that are richer than simple native types. Often they're used as a denormalized analog for relational foreign keys. In contrast to foreign keys, which only hold a reference to a row in another table, columns with user-defined types actually carry data that may be copied from a row in another table. In the case of the `ingredients` column in the `tacos` table, it will contain a collection of data structures that define the ingredients themselves.

You can't use the `Ingredient` class as a user-defined type, because the `@Table` annotation has already mapped it as an entity for persistence in Cassandra. Therefore, you must create a new class to define how ingredients will be stored in the `ingredients` column of the `taco` table. `IngredientUDT` (where "UDT" means *user-defined type*) is the class for the job:

```
package tacos;

import org.springframework.data.cassandra.core.mapping.UserDefinedType;

import lombok.AccessLevel;
import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.RequiredArgsConstructor;
```

```
@Data
@RequiredArgsConstructor
@NoArgsConstructor(access = AccessLevel.PRIVATE, force = true)
@UserDefinedType("ingredient")
public class IngredientUDT {

  private final String name;

  private final Ingredient.Type type;

}
```

Although `IngredientUDT` looks a lot like `Ingredient`, its mapping requirements are much simpler. It's annotated with `@UserDefinedType` to identify it as a user-defined type in Cassandra. But otherwise, it's a simple class with a few properties.

You'll also note that the `IngredientUDT` class doesn't include an `id` property. Although it could include a copy of the `id` property from the source `Ingredient`, that's not necessary. In fact, the user-defined type may include any properties you wish—it doesn't need to be a one-to-one mapping with any table definition.

I realize that it might be difficult to visualize how data in a user-defined type relates to data that's persisted to a table. Figure 4.1 shows the data model for the entire Taco Cloud database, including user-defined types.

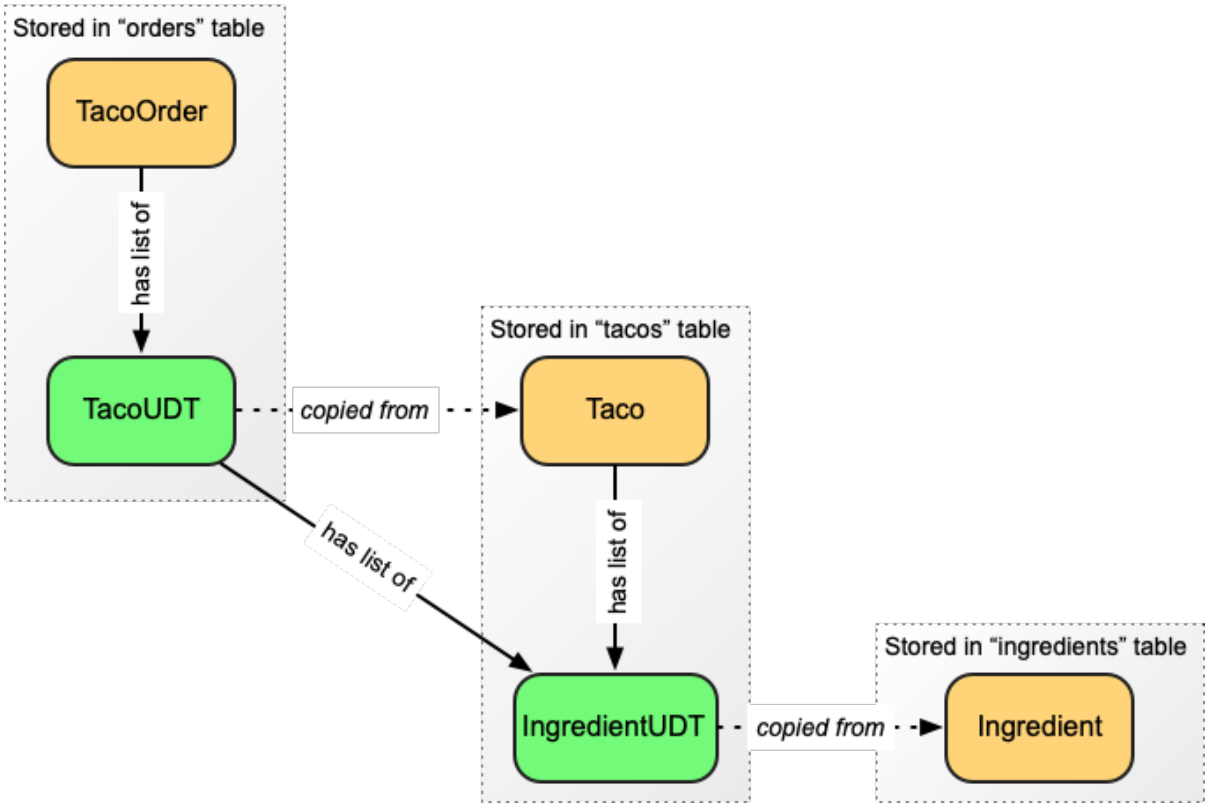

Figure 4.1 Instead of using foreign keys and joins, Cassandra tables are denormalized, with user-defined types containing data copied from related tables.

Specific to the user-defined type that you just created, notice how `Taco` has a list of `IngredientUDT`, which holds data copied from `Ingredient` objects. When a `Taco` is persisted, it's the `Taco` object and the list of `IngredientUDT` that's persisted to the `tacos` table. The list of `IngredientUDT` is persisted entirely within the `ingredients` column.

Another way of looking at this that might help you understand how user-defined types are used is to query the database for rows from the `tacos` table. Using CQL and the `cqlsh` tool that comes with Cassandra, you see the following results:

```
cqlsh:tacocloud> select id, name, createdAt, ingredients from tacos;

 id       | name      | createdat | ingredients
----------+-----------+-----------+----------------------------------------
 827390...| Carnivore | 2018-04...| [{name: 'Flour Tortilla', type: 'WRAP'},
                                    {name: 'Carnitas', type: 'PROTEIN'},
                                    {name: 'Sour Cream', type: 'SAUCE'},
                                    {name: 'Salsa', type: 'SAUCE'},
                                    {name: 'Cheddar', type: 'CHEESE'}]

(1 rows)
```

As you can see, the `id`, `name`, and `createdat` columns contain simple values. In that regard, they aren't much different than what you'd expect from a similar query against a relational database. But the `ingredients` column is a little different. Because it's defined as containing a collection of the user-defined `ingredient` type (defined by `IngredientUDT`), its value appears as a JSON array filled with JSON objects.

You likely noticed other user-defined types in figure 4.1. You'll certainly be creating some more as you continue mapping your domain to Cassandra tables, including some that will be used by the `TacoOrder` class. The next listing shows the `TacoOrder` class, modified for Cassandra persistence.

Listing 4.2 Mapping the `TacoOrder` class to a Cassandra `orders` table

```
package tacos;
import java.io.Serializable;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;
import java.util.UUID;

import javax.validation.constraints.Digits;
import javax.validation.constraints.NotBlank;
import javax.validation.constraints.Pattern;

import org.hibernate.validator.constraints.CreditCardNumber;
import org.springframework.data.cassandra.core.mapping.Column;
import org.springframework.data.cassandra.core.mapping.PrimaryKey;
import org.springframework.data.cassandra.core.mapping.Table;

import com.datastax.oss.driver.api.core.uuid.Uuids;

import lombok.Data;

@Data
@Table("orders")   ❶
public class TacoOrder implements Serializable {

  private static final long serialVersionUID = 1L;

  @PrimaryKey    ❷
  private UUID id = Uuids.timeBased();

  private Date placedAt = new Date();

  // delivery and credit card properties omitted for brevity's sake

  @Column("tacos")    ❸
  private List<TacoUDT> tacos = new ArrayList<>();

  public void addTaco(TacoUDT taco) {
    this.tacos.add(taco);
  }

}
```

❶ Maps to orders table

❷ Declares the primary key

❸ Maps a list to the tacos column

Listing 4.2 purposefully omits many of the properties of `TacoOrder` that don't lend themselves to a discussion of Cassandra data modeling. What's left are a few properties and mappings, similar to how `Taco` was defined. `@Table` is used to map `TacoOrder` to the `orders` table, much as `@Table` has been used before. In this case, you're unconcerned with ordering, so the `id` property is simply annotated with `@PrimaryKey`, designating it as both a partition key and a clustering key with default ordering.

The `tacos` property is of some interest in that it's a `List<TacoUDT>` instead of a list of `Taco` objects. The relationship between `TacoOrder` and `Taco/TacoUDT` here is similar to the

relationship between `Taco` and `Ingredient/IngredientUDT`. That is, rather than joining data from several rows in a separate table through foreign keys, the "orders" table will contain all of the pertinent taco data, optimizing the table for quick reads.

The `TacoUDT` class is quite similar to the `IngredientUDT` class, although it does include a collection that references another user-defined type:

```
package tacos;

import java.util.List;
import org.springframework.data.cassandra.core.mapping.UserDefinedType;
import lombok.Data;

@Data
@UserDefinedType("taco")
public class TacoUDT {

  private final String name;
  private final List<IngredientUDT> ingredients;

}
```

Although it would have been nice to reuse the same domain classes you created in chapter 3, or at most to swap out some JPA annotations for Cassandra annotations, the nature of Cassandra persistence is such that it requires you to rethink how your data is modeled. But now that you've mapped your domain, you're ready to write repositories.

4.1.4 Writing Cassandra repositories

As you saw in chapter 3, writing a repository with Spring Data involves simply declaring an interface that extends one of Spring Data's base repository interfaces and optionally declaring additional query methods for custom queries. As it turns out, writing repositories isn't much different.

In fact, there's very little that you'll need to change in the repositories we've already written to make them work for Cassandra persistence. For example, consider the `IngedientRepository` we created in chapter 3:

```
package tacos.data;

import org.springframework.data.repository.CrudRepository;

import tacos.Ingredient;

public interface IngredientRepository
         extends CrudRepository<Ingredient, String> {

}
```

By extending `CrudRepository` as shown here, `IngredientRepository` is ready to persist `Ingredient` objects whose ID property (or, in the case of Cassandra, the primary key property) is a `String`. That's perfect! No changes are needed for `IngredientRepository`.

The changes required for `OrderRepository` are only slightly more involved. Instead of a `Long` parameter, the ID parameter type specified when extending `CrudRepository` will be changed to `UUID`:

```
package tacos.data;

import java.util.UUID;

import org.springframework.data.repository.CrudRepository;

import tacos.TacoOrder;

public interface OrderRepository
         extends CrudRepository<TacoOrder, UUID> {

}
```

There's a lot of power in Cassandra, and when it's teamed up with Spring Data, you can wield that power in your Spring applications. But let's shift our attention to another database for which Spring Data repository support is available: MongoDB.

4.2 Writing MongoDB repositories

MongoDB is a another well-known NoSQL database. Whereas Cassandra is a column-store database, MongoDB is considered a document database. More specifically, MongoDB stores documents in BSON (Binary JSON) format, which can be queried for and retrieved in a way that's roughly similar to how you might query for data in any other database.

As with Cassandra, it's important to understand that MongoDB isn't a relational database. The way you manage your MongoDB server cluster, as well as how you model your data, requires a different mindset than when working with other kinds of databases.

That said, working with MongoDB and Spring Data isn't dramatically different from how you might use Spring Data for working with JPA or Cassandra. You'll annotate your domain classes with annotations that map the domain type to a document structure. And you'll write repository interfaces that very much follow the same programming model as those you've seen for JPA and Cassandra. Before you can do any of that, though, you must enable Spring Data MongoDB in your project.

4.2.1 Enabling Spring Data MongoDB

To get started with Spring Data MongoDB, you'll need to add the Spring Data MongoDB starter to the project build. As with Spring Data Cassandra, Spring Data MongoDB has two separate starters to choose from: One reactive and one non-reactive. We'll look at the reactive options for persistence in chapter 15. For now, add the following dependency to the build to work with the non-reactive MongoDB starter:

```
<dependency>
```

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

This dependency is also available from the Spring Initializr by checking the MongoDB check box.

By adding the starter to the build, autoconfiguration will be triggered to enable Spring Data support for writing automatic repository interfaces, such as those you wrote for JPA in chapter 3 or for Cassandra earlier in this chapter.

By default, Spring Data MongoDB assumes that you have a MongoDB server running locally and listening on port 27017. If you have Docker installed on your machine, an easy way to get a MongoDB server running is with the following command line:

```
$ docker run -p27017:27017 -d mongo:latest
```

But for convenience in testing or developing, you can choose to work with an embedded Mongo database instead. To do that, add the Flapdoodle Embedded MongoDB dependency to your build:

```
<dependency>
  <groupId>de.flapdoodle.embed</groupId>
  <artifactId>de.flapdoodle.embed.mongo</artifactId>
  <!-- <scope>test</scope> -->
</dependency>
```

The Flapdoodle embedded database affords you all of the same convenience of working with an in-memory Mongo database as you'd get with H2 when working with relational data. That is, you won't need to have a separate database running, but all data will be wiped clean when you restart the application.

Embedded databases are fine for development and testing, but once you take your application to production, you'll want to be sure you set a few properties to let Spring Data MongoDB know where and how your production Mongo database can be accessed:

```
spring:
  data:
    mongodb:
      host: mongodb.tacocloud.com
      port: 27018
      username: tacocloud
      password: s3cr3tp455w0rd
      database: tacoclouddb
```

Not all of these properties are required, but they're available to help point Spring Data MongoDB in the right direction in the event that your Mongo database isn't running locally. Breaking it down, here's what each property configures:

- `spring.data.mongodb.host` —The hostname where Mongo is running (default: `localhost`)
- `spring.data.mongodb.port` — The port that the Mongo server is listening on (default: `27017`)
- `spring.data.mongodb.username` — The username to use to access a secured Mongo database
- `spring.data.mongodb.password` — The password to use to access a secured Mongo database
- `spring.data.mongodb.database` — The database name (default: test)

Now that you have Spring Data MongoDB enabled in your project, you need to annotate your domain objects for persistence as documents in MongoDB.

4.2.2 Mapping domain types to documents

Spring Data MongoDB offers a handful of annotations that are useful for mapping domain types to document structures to be persisted in MongoDB. Although Spring Data MongoDB provides a half dozen annotations for mapping, only four of them are useful for most common use cases:

- `@Id` — Designates a property as the document ID (from Spring Data Commons)
- `@Document` — Declares a domain type as a document to be persisted to MongoDB
- `@Field` — Specifies the field name (and optionally the order) for storing a property in the persisted document
- `@Transient` - Specifies that a property is not to be persisted.

Of those three annotations, only the `@Id` and `@Document` annotations are strictly required. Unless you specify otherwise, properties that aren't annotated with `@Field` or `@Transient` will assume a field name equal to the property name.

Applying these annotations to the `Ingredient` class, you get the following:

```
package tacos;

import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

import lombok.AccessLevel;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@Document
@AllArgsConstructor
@NoArgsConstructor(access=AccessLevel.PRIVATE, force=true)
public class Ingredient {

  @Id
  private String id;
  private String name;
  private Type type;

  public static enum Type {
    WRAP, PROTEIN, VEGGIES, CHEESE, SAUCE
```

```
  }

}
```

As you can see, you place the `@Document` annotation at the class level to indicate that `Ingredient` is a document entity that can be written to and read from a Mongo database. By default, the collection name (the Mongo analog to a relational database table) is based on the class name, with the first letter lowercased. Because you haven't specified otherwise, `Ingredient` objects will be persisted to a collection named `ingredient`. But you can change that by setting the `collection` attribute of `@Document`:

```
@Data
@AllArgsConstructor
@NoArgsConstructor(access=AccessLevel.PRIVATE, force=true)
@Document(collection="ingredients")
public class Ingredient {
...
}
```

You'll also notice that the `id` property has been annotated with `@Id`. This designates the property as being the ID of the persisted document. You can use `@Id` on any property whose type is `Serializable`, including `String` and `Long`. In this case, you're already using the `String`-defined `id` property as a natural identifier, so there's no need to change it to any other type.

So far, so good. But you'll recall from earlier in this chapter that `Ingredient` was the easy domain type to map for Cassandra. The other domain types, such as `Taco`, were a bit more challenging. Let's look at how you can map the `Taco` class to see what surprises it might hold.

MongoDB's approach to document persistence lends itself very well to the domain-driven design way of applying persistence at the aggregate root level. Documents in MongoDB tend to be defined as aggregate roots with members of the aggregate as sub-documents.

What that means for Taco Cloud is that since `Taco` is only ever persisted as a member of the `TacoOrder`-rooted aggregate, the `Taco` class doesn't need to be annotated as a `@Document`, nor does it need an `@Id` property. The `Taco` class can remain clean of any persistence annotations:

```
package tacos;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;

import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

import lombok.Data;

@Data
public class Taco {

  @NotNull
  @Size(min=5, message="Name must be at least 5 characters long")
  private String name;
```

```
  private Date createdAt = new Date();

  @Size(min=1, message="You must choose at least 1 ingredient")
  private List<Ingredient> ingredients = new ArrayList<>();

  public void addIngredient(Ingredient ingredient) {
    this.ingredients.add(ingredient);
  }

}
```

The `TacoOrder` class, however, being the root of the aggregate, will need to be annotated with `@Document` and have an `@Id` property.

```
package tacos;
import java.io.Serializable;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;

import javax.validation.constraints.Digits;
import javax.validation.constraints.NotBlank;
import javax.validation.constraints.Pattern;

import org.hibernate.validator.constraints.CreditCardNumber;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

import lombok.Data;

@Data
@Document
public class TacoOrder implements Serializable {

  private static final long serialVersionUID = 1L;

  @Id
  private String id;

  private Date placedAt = new Date();

  // other properties omitted for brevity's sake

  private List<Taco> tacos = new ArrayList<>();

  public void addTaco(Taco taco) {
    this.tacos.add(taco);
  }

}
```

For brevity's sake, I've snipped out the various delivery and credit card fields. But from what's left, it's clear that all you need is `@Document` and `@Id`, as with the other domain types.

Notice, however, that the `id` property has been changed to be a `String` (as opposed to a `Long` in the JPA version or a `UUID` in the Cassandra version). As I said earlier, `@Id` can be applied to any `Serializable` type. But if you choose to use a `String` property as the ID, you get the benefit of Mongo automatically assigning a value to it when it's saved (assuming that it's `null`). By choosing `String`, you get a database-managed ID assignment and needn't worry about setting that property manually.

Although there are some more-advanced and unusual use cases that require additional mapping, you'll find that for most cases, `@Document` and `@Id`, along with an occasional `@Field` or `@Transient`, are sufficient for MongoDB mapping. They certainly do the job for the Taco Cloud domain types.

All that's left is to write the repository interfaces.

4.2.3 Writing MongoDB repository interfaces

Spring Data MongoDB offers automatic repository support similar to what's provided by Spring Data JPA and Spring Data Cassandra.

You'll start by defining a repository for persisting `Ingredient` objects as documents. As before, you can write `IngredientRepository` to extend `CrudRepository`:

```
package tacos.data;

import org.springframework.data.repository.CrudRepository;

import tacos.Ingredient;

public interface IngredientRepository
         extends CrudRepository<Ingredient, String> {

}
```

Wait a minute! That looks *identical* to the `IngredientRepository` interface you wrote in section 4.1 for Cassandra! Indeed, it's the same interface, with no changes. This highlights one of the benefits of extending `CrudRepository`—it's more portable across various database types and works equally well for MongoDB as for Cassandra.

Moving on to the `OrderRepository` interface, you can see that it's quite straightforward:

```
package tacos.data;

import org.springframework.data.repository.CrudRepository;

import tacos.TacoOrder;

public interface OrderRepository
         extends CrudRepository<TacoOrder, String> {

}
```

Just like `IngredientRepository`, `OrderRepository` extends `CrudRepository` to gain the optimizations afforded in its `insert()` methods. Otherwise, there's nothing terribly special about this repository, compared to some of the other repositories you've defined thus far. Note, however, that the ID parameter when extending `CrudRepository` is now `String` instead of `Long` (as for JPA) or `UUID` (as for Cassandra). This reflects the change we made in `TacoOrder` to support automatic assignment of IDs.

In the end, working with Spring Data MongoDB isn't drastically different than the other Spring Data projects we've worked with. The domain types are annotated differently. But aside from the ID parameter specified when extending `CrudRepository`, the repository interfaces are nearly identical.

4.3 Summary

- Spring Data supports repositories for a variety of NoSQL databases, including Cassandra, MongoDB, Neo4j, and Redis.
- The programming model for creating repositories differs very little across different underlying databases.
- Working with non-relational databases demands an understanding of how to model data appropriately for how the database ultimately stores the data.

5 Securing Spring

This chapter covers

- Autoconfiguring Spring Security
- Defining custom user storage
- Customizing the login page
- Securing against CSRF attacks
- Knowing your user

Have you ever noticed that most people in television sitcoms don't lock their doors? In the days of *Leave it to Beaver*, it wasn't so unusual for people to leave their doors unlocked. But it seems crazy that in a day when we're concerned with privacy and security, we see television characters enabling unhindered access to their apartments and homes.

Information is probably the most valuable item we now have; crooks are looking for ways to steal our data and identities by sneaking into unsecured applications. As software developers, we must take steps to protect the information that resides in our applications. Whether it's an email account protected with a username-password pair or a brokerage account protected with a trading PIN, security is a crucial aspect of most applications.

5.1 Enabling Spring Security

The very first step in securing your Spring application is to add the Spring Boot security starter dependency to your build. In the project's pom.xml file, add the following `<dependency>` entry:

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

If you're using Spring Tool Suite, this is even easier. Right-click on the pom.xml file and select Edit Starters from the Spring context menu. The Starter Dependencies dialog box will appear. Check the Spring Security entry under the Security category, as shown in figure 5.1.

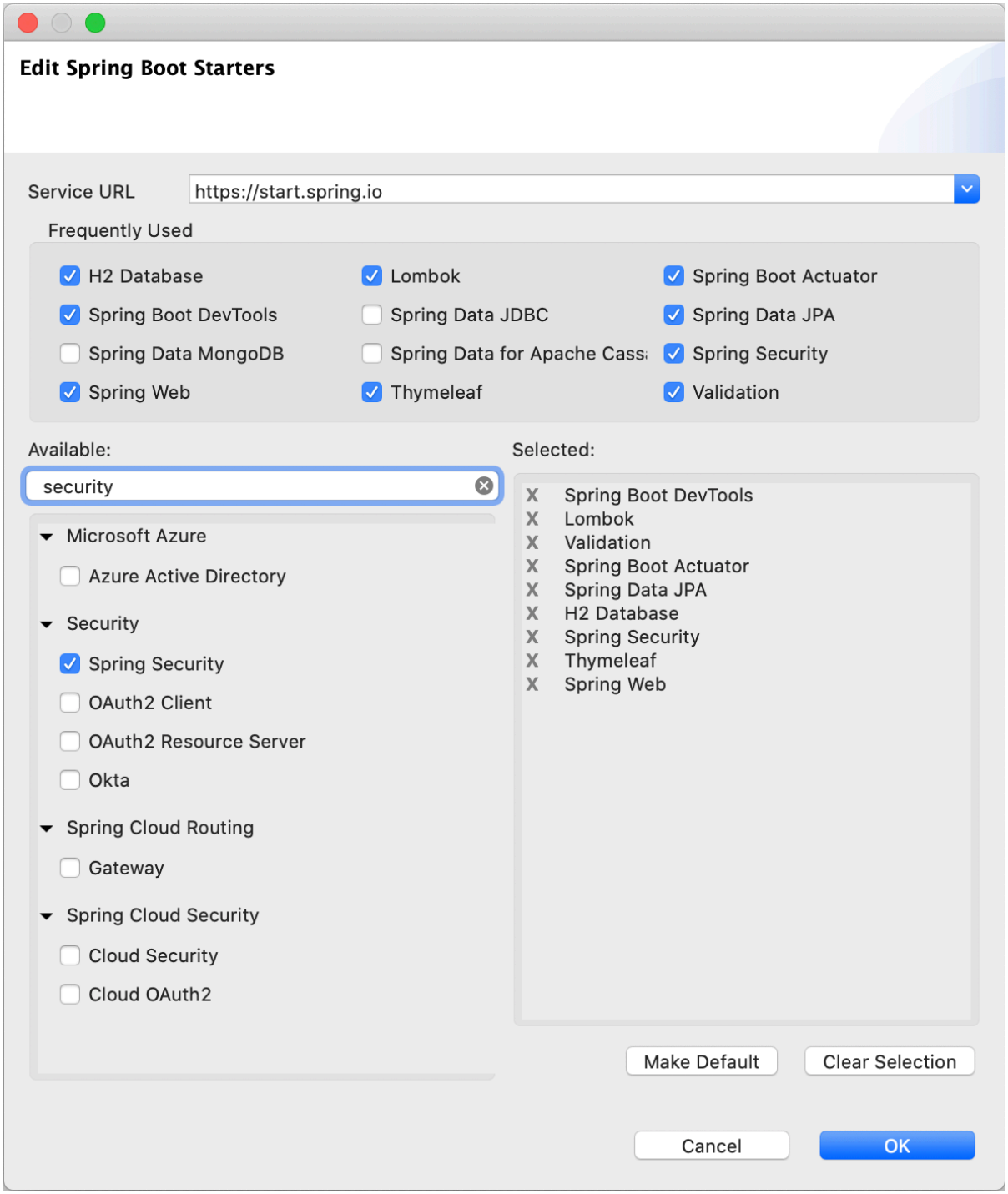

Figure 5.1 Adding the security starter with Spring Tool Suite

Believe it or not, that dependency is the only thing that's required to secure an application. When the application starts, autoconfiguration will detect that Spring Security is in the classpath and will set up some basic security configuration.

If you want to try it out, fire up the application and try to visit the homepage (or any page for that matter). You'll be prompted for authentication with a rather plain login page that looks something like figure 5.2.

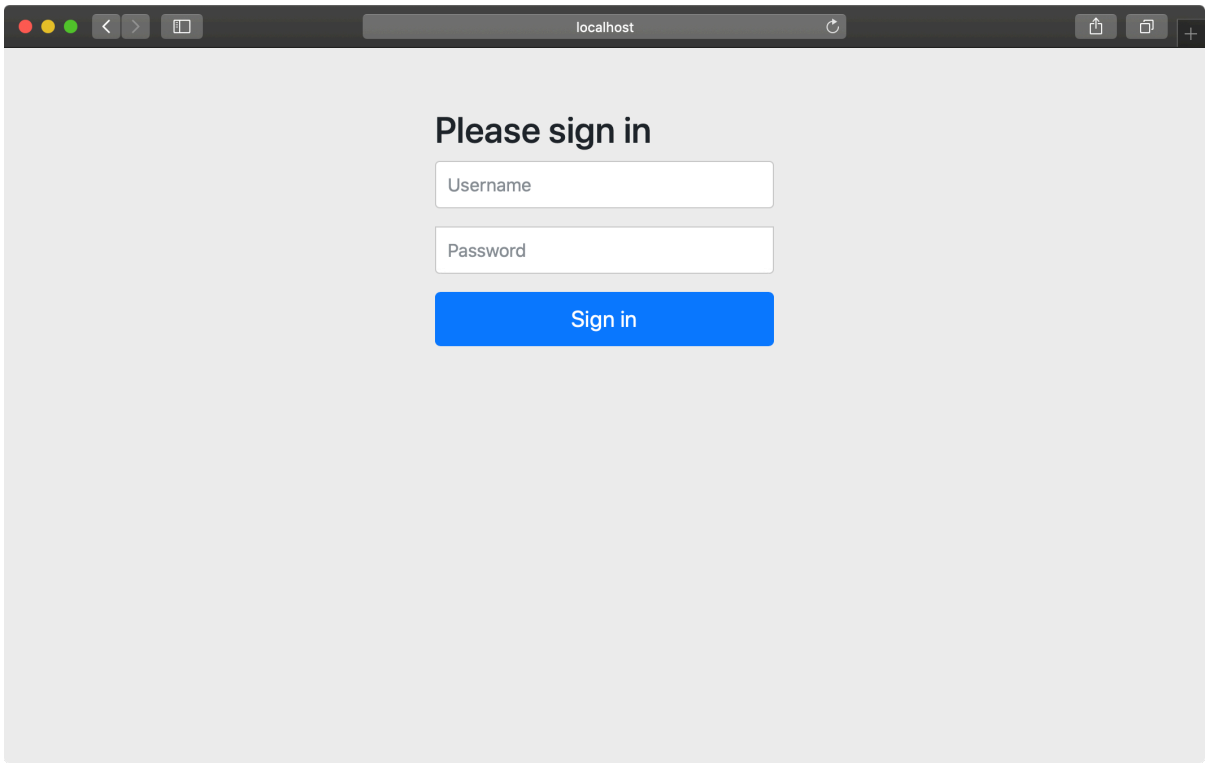

Figure 5.2 Spring Security gives you a plain login page for free.

> **TIP** **Going incognito: You may find it useful to set your browser to private or incognito mode when manually testing security. This will ensure that you have a fresh session each time you open a private/incognito window. You'll have to sign in to the application each time, but you can be assured that any changes you've made in security are applied, and that there aren't any remnants of an older session preventing you from seeing your changes.**

To get past it, you'll need to provide a username and password. The username is *user*. As for the password, it's randomly generated and written to the application log file. The log entry will look something like this:

```
Using generated security password: 087cfc6a-027d-44bc-95d7-cbb3a798a1ea
```

Assuming you enter the username and password correctly, you'll be granted access to the application.

It seems that securing Spring applications is pretty easy work. With the Taco Cloud application secured, I suppose I could end this chapter now and move on to the next topic. But before we get ahead of ourselves, let's consider what kind of security autoconfiguration has provided.

By doing nothing more than adding the security starter to the project build, you get the following security features:

- All HTTP request paths require authentication.
- No specific roles or authorities are required.
- Authentication is prompted with a simple login page.
- There's only one user; the username is *user*.

This is a good start, but I think that the security needs of most applications (Taco Cloud included) will be quite different from these rudimentary security features.

You have more work to do if you're going to properly secure the Taco Cloud application. You'll need to at least configure Spring Security to do the following:

- Provide a login page that is designed to match the website.
- Provide for multiple users, and enable a registration page so new Taco Cloud customers can sign up.
- Apply different security rules for different request paths. The homepage and registration pages, for example, shouldn't require authentication at all.

To meet your security needs for Taco Cloud, you'll have to write some explicit configuration, overriding what autoconfiguration has given you. You'll start by configuring a proper user store so that you can have more than one user.

5.2 Configuring authentication

Over the years there have been several ways of configuring Spring Security, including lengthy XML-based configuration. Fortunately, several recent versions of Spring Security have supported Java-based configuration, which is much easier to read and write.

Before this chapter is finished, you'll have configured all of your Taco Cloud security needs in Java-based Spring Security configuration. But to get started, you'll ease into it by writing a configuration class shown in the following listing.

Listing 5.1 A barebones configuration class for Spring Security

```
package tacos.security;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;

@Configuration
public class SecurityConfig {

  @Bean
  public PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder();
  }

}
```

What does this barebones security configuration do for you? Not much, actually. The main thing it does is declare a `PasswordEncoder` bean, which we'll use both when creating new users and when authenticating users at login. In this case, we're using `BCryptPasswordEncoder`, one of a handful of password encoders provided by Spring Security, including the following:

- `BCryptPasswordEncoder` — Applies bcrypt strong hashing encryption
- `NoOpPasswordEncoder` — Applies no encoding
- `Pbkdf2PasswordEncoder` — Applies PBKDF2 encryption
- `SCryptPasswordEncoder` — Applies scrypt hashing encryption
- `StandardPasswordEncoder` — Applies SHA-256 hashing encryption

No matter which password encoder you use, it's important to understand that the password in the database is never decoded. Instead, the password that the user enters at login is encoded using the same algorithm, and it's then compared with the encoded password in the database. That comparison is performed in the `PasswordEncoder`'s `matches()` method.

In addition to the password encoder, we'll fill in this configuration class with more beans to define the specifics of security for our application. We'll start by configuring a user store that can handle more than one user.

In order to configure a user store for authentication purposes, you'll need to declare a `UserDetailsService` bean. The `UserDetailsService` interface is relatively simple including only one method that must be implemented. Here's what `UserDetailsService` looks like:

```
public interface UserDetailsService {

    UserDetails loadUserByUsername(String username) throws UsernameNotFoundException;

}
```

The `loadUserByUsername()` method accepts a username and uses it to look up a `UserDetails` object. If no user can be found for the given username, then it will throw a

`UsernameNotFoundException`.

As it turns out, Spring Security offers several out of the box implementations of `UserDetailsService`, including:

- An in-memory user store
- A JDBC-based user store
- An LDAP-backed user store

Or, you can also create your own implementation to suit your application's specific security needs.

To get started, let's try out the in-memory implementation of `UserDetailsService`.

5.2.1 In-memory user details service

One place where user information can be kept is in memory. Suppose you have only a handful of users, none of which are likely to change. In that case, it may be simple enough to define those users as part of the security configuration.

The following bean method shows how to create an `InMemoryUserDetailsManager` with two users, "buzz" and "woody", for that purpose:

Listing 5.2 Declaring users in an in-memory user service bean

```
@Bean
public UserDetailsService userDetailsService(PasswordEncoder encoder) {
  List<UserDetails> usersList = new ArrayList<>();
  usersList.add(new User(
      "buzz", encoder.encode("password"),
          Arrays.asList(new SimpleGrantedAuthority("ROLE_USER"))));
  usersList.add(new User(
      "woody", encoder.encode("password"),
          Arrays.asList(new SimpleGrantedAuthority("ROLE_USER"))));
  return new InMemoryUserDetailsManager(usersList);
}
```

Here, a list of Spring Security `User` objects are created, each with a username, password, and a list of one or more authorities. Then an `InMemoryUserDetailsManager` is created using that list.

If you try out the application now, you should be able to log in as either "woody" or "buzz", using "password" as the password.

The in-memory user details service is convenient for testing purposes or for very simple applications, but it doesn't allow for easy editing of users. If you need to add, remove, or change a user, you'll have to make the necessary changes and then rebuild and redeploy the application.

For the Taco Cloud application, you want customers to be able to register with the application

and manage their own user accounts. That doesn't fit with the limitations of the in-memory user details service. So let's take a look at how to create our own implementation of `UserDetailsService` that allows for a database-backed user store.

5.2.2 Customizing user authentication

In the last chapter, you settled on using Spring Data JPA as your persistence option for all taco, ingredient, and order data. It would thus make sense to persist user data in the same way. If you do so, the data will ultimately reside in a relational database, so you could use JDBC-based authentication. But it'd be even better to leverage the Spring Data repository used to store users.

First things first, though. Let's create the domain object and repository interface that represents and persists user information.

DEFINING THE USER DOMAIN AND PERSISTENCE

When Taco Cloud customers register with the application, they'll need to provide more than just a username and password. They'll also give you their full name, address, and phone number. This information can be used for a variety of purposes, including prepopulating the order form (not to mention potential marketing opportunities).

To capture all of that information, you'll create a `User` class, as follows.

Listing 5.3 Defining a user entity

```
package tacos;
import java.util.Arrays;
import java.util.Collection;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.
                                          SimpleGrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;
import lombok.AccessLevel;
import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.RequiredArgsConstructor;

@Entity
@Data
@NoArgsConstructor(access=AccessLevel.PRIVATE, force=true)
@RequiredArgsConstructor
public class User implements UserDetails {

  private static final long serialVersionUID = 1L;

  @Id
  @GeneratedValue(strategy=GenerationType.AUTO)
  private Long id;

  private final String username;
  private final String password;
  private final String fullname;
  private final String street;
  private final String city;
  private final String state;
  private final String zip;
  private final String phoneNumber;

  @Override
  public Collection<? extends GrantedAuthority> getAuthorities() {
    return Arrays.asList(new SimpleGrantedAuthority("ROLE_USER"));
  }

  @Override
  public boolean isAccountNonExpired() {
    return true;
  }

  @Override
  public boolean isAccountNonLocked() {
    return true;
  }

  @Override
  public boolean isCredentialsNonExpired() {
    return true;
  }

  @Override
  public boolean isEnabled() {
    return true;
  }

}
```

The first thing to notice about this `User` type is that it's not the same as the `User` class we used when creating the in-memory user details service. This one has more details about the user that we'll need to fulfill taco orders, including the user's address and contact information.

You've also no doubt noticed that the `User` class is a bit more involved than any of the other entities defined in chapter 3. In addition to defining a handful of properties, `User` also implements the `UserDetails` interface from Spring Security.

Implementations of `UserDetails` will provide some essential user information to the framework, such as what authorities are granted to the user and whether the user's account is enabled or not.

The `getAuthorities()` method should return a collection of authorities granted to the user. The various `is___Expired()` methods return a `boolean` to indicate whether or not the user's account is enabled or expired.

For your `User` entity, the `getAuthorities()` method simply returns a collection indicating that all users will have been granted `ROLE_USER` authority. And, at least for now, Taco Cloud has no need to disable users, so all of the `is___Expired()` methods return `true` to indicate that the users are active.

With the `User` entity defined, you now can define the repository interface:

```
package tacos.data;
import org.springframework.data.repository.CrudRepository;
import tacos.User;

public interface UserRepository extends CrudRepository<User, Long> {

  User findByUsername(String username);

}
```

In addition to the CRUD operations provided by extending `CrudRepository`, `UserRepository` defines a `findByUsername()` method that you'll use in the user details service to look up a `User` by their username.

As you learned in chapter 3, Spring Data JPA will automatically generate the implementation of this interface at runtime. Therefore, you're now ready to write a custom user details service that uses this repository.

CREATING A USER DETAILS SERVICE

As you'll recall, the `UserDetailsService` interface defines only a single `loadUserByUsername()` method. That means it is a functional interface and can be implemented as a lambda instead of as a full-blown implementation class. Since all we really need is for our custom `UserDetailsService` to delegate to the `UserRepository`, it can be simply declared as a bean using the following configuration method:

Listing 5.4 Defining a custom user details service bean

```
@Bean
public UserDetailsService userDetailsService(UserRepository userRepo) {
  return username -> {
    User user = userRepo.findByUsername(username);
    if (user != null) return user;

    throw new UsernameNotFoundException("User '" + username + "' not found");
  };
}
```

The `userDetailsService()` method is given a `UserRepository` as a parameter. To create the bean, it returns a lambda that takes a `username` parameter and uses it to call `findByUsername()` on the given `UserRepository`.

The `loadByUsername()` method has one simple rule: it must never return `null`. Therefore, if the call to `findByUsername()` returns `null`, the lambda will throw a `UsernameNotFoundException`. Otherwise, the `User` that was found will be returned.

Now that you have a custom user details service that reads user information via a JPA repository, you just need a way to get users into the database in the first place. You need to create a registration page for Taco Cloud patrons to register with the application.

REGISTERING USERS

Although Spring Security handles many aspects of security, it really isn't directly involved in the process of user registration, so you're going to rely on a little bit of Spring MVC to handle that task. The `RegistrationController` class in the following listing presents and processes registration forms.

Listing 5.5 A user registration controller

```
package tacos.security;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import tacos.data.UserRepository;

@Controller
@RequestMapping("/register")
public class RegistrationController {

  private UserRepository userRepo;
  private PasswordEncoder passwordEncoder;

  public RegistrationController(
      UserRepository userRepo, PasswordEncoder passwordEncoder) {
    this.userRepo = userRepo;
    this.passwordEncoder = passwordEncoder;
  }

  @GetMapping
  public String registerForm() {
    return "registration";
  }

  @PostMapping
  public String processRegistration(RegistrationForm form) {
    userRepo.save(form.toUser(passwordEncoder));
    return "redirect:/login";
  }

}
```

Like any typical Spring MVC controller, `RegistrationController` is annotated with `@Controller` to designate it as a controller and to mark it for component scanning. It's also annotated with `@RequestMapping` such that it will handle requests whose path is /register.

More specifically, a GET request for /register will be handled by the `registerForm()` method, which simply returns a logical view name of `registration`. The following listing shows a Thymeleaf template that defines the `registration` view.

Listing 5.6 A Thymeleaf registration form view

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">
  <head>
    <title>Taco Cloud</title>
  </head>

  <body>
    <h1>Register</h1>
    <img th:src="@{/images/TacoCloud.png}"/>

    <form method="POST" th:action="@{/register}" id="registerForm">

        <label for="username">Username: </label>
        <input type="text" name="username"/><br/>

        <label for="password">Password: </label>
        <input type="password" name="password"/><br/>

        <label for="confirm">Confirm password: </label>
        <input type="password" name="confirm"/><br/>

        <label for="fullname">Full name: </label>
        <input type="text" name="fullname"/><br/>

        <label for="street">Street: </label>
        <input type="text" name="street"/><br/>

        <label for="city">City: </label>
        <input type="text" name="city"/><br/>

        <label for="state">State: </label>
        <input type="text" name="state"/><br/>

        <label for="zip">Zip: </label>
        <input type="text" name="zip"/><br/>

        <label for="phone">Phone: </label>
        <input type="text" name="phone"/><br/>

        <input type="submit" value="Register"/>
    </form>

  </body>
</html>
```

When the form is submitted, the HTTP POST request will be handled by the `processRegistration()` method. The `RegistrationForm` object given to `processRegistration()` is bound to the request data and is defined with the following class:

```
package tacos.security;
import org.springframework.security.crypto.password.PasswordEncoder;
import lombok.Data;
import tacos.User;

@Data
public class RegistrationForm {

  private String username;
  private String password;
  private String fullname;
  private String street;
```

```
  private String city;
  private String state;
  private String zip;
  private String phone;

  public User toUser(PasswordEncoder passwordEncoder) {
    return new User(
        username, passwordEncoder.encode(password),
        fullname, street, city, state, zip, phone);
  }

}
```

For the most part, `RegistrationForm` is just a basic Lombok-enabled class with a handful of properties. But the `toUser()` method uses those properties to create a new `User` object, which is what `processRegistration()` will save, using the injected `UserRepository`.

You've no doubt noticed that `RegistrationController` is injected with a `PasswordEncoder`. This is the exact same `PasswordEncoder` bean you declared before. When processing a form submission, `RegistrationController` passes it to the `toUser()` method, which uses it to encode the password before saving it to the database. In this way, the submitted password is written in an encoded form, and the user details service will be able to authenticate against that encoded password.

Now the Taco Cloud application has complete user registration and authentication support. But if you start it up at this point, you'll notice that you can't even get to the registration page without being prompted to log in. That's because, by default, all requests require authentication. Let's look at how web requests are intercepted and secured so you can fix this strange chicken-and-egg situation.

5.3 Securing web requests

The security requirements for Taco Cloud should require that a user be authenticated before designing tacos or placing orders. But the homepage, login page, and registration page should be available to unauthenticated users.

To configure these security rules, we'll need to declare a `SecurityFilterChain` bean. The following `@Bean` method shows a minimal (but not useful) `SecurityFilterChain` bean declaration:

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
  return http.build();
}
```

The `filterChain()` method accepts an `HttpSecurity` object, which acts as a builder that can be used to configure how security is handled at the web level. Once security configuration is setup via the `HttpSecurity` object, a call to `build()` will create a `SecurityFilterChain` that is returned from the bean method.

Among the many things you can configure with `HttpSecurity` are these:

- Requiring that certain security conditions be met before allowing a request to be served
- Configuring a custom login page
- Enabling users to log out of the application
- Configuring cross-site request forgery protection

Intercepting requests to ensure that the user has proper authority is one of the most common things you'll configure `HttpSecurity` to do. Let's ensure that your Taco Cloud customers meet those requirements.

5.3.1 Securing requests

You need to ensure that requests for /design and /orders are only available to authenticated users; all other requests should be permitted for all users. The following configuration does exactly that:

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
  return http
    .authorizeRequests()
      .antMatchers("/design", "/orders").hasRole("USER")
      .antMatchers("/", "/**").permitAll()

    .and()
    .build();
}
```

The call to `authorizeRequests()` returns an object (`ExpressionUrlAuthorizationConfigurer.ExpressionInterceptUrlRegistry`) on which you can specify URL paths and patterns and the security requirements for those paths. In this case, you specify two security rules:

- Requests for /design and /orders should be for users with a granted authority of `ROLE_USER`. Don't include the "ROLE_" prefix on roles passed to `hasRole()`; it will be assumed by `hasRole()`.
- All requests should be permitted to all users.

The order of these rules is important. Security rules declared first take precedence over those declared lower down. If you were to swap the order of those two security rules, all requests would have `permitAll()` applied to them; the rule for /design and /orders requests would have no effect.

The `hasRole()` and `permitAll()` methods are just a couple of the methods for declaring security requirements for request paths. Table 5.1 describes all the available methods.

Table 5.1 Configuration methods to define how a path is to be secured (continued)

Method	What it does
`access(String)`	Allows access if the given SpEL expression evaluates to `true`
`anonymous()`	Allows access to anonymous users
`authenticated()`	Allows access to authenticated users
`denyAll()`	Denies access unconditionally
`fullyAuthenticated()`	Allows access if the user is fully authenticated (not remembered)
`hasAnyAuthority(String…)`	Allows access if the user has any of the given authorities
`hasAnyRole(String…)`	Allows access if the user has any of the given roles
`hasAuthority(String)`	Allows access if the user has the given authority
`hasIpAddress(String)`	Allows access if the request comes from the given IP address
`hasRole(String)`	Allows access if the user has the given role
`not()`	Negates the effect of any of the other access methods
`permitAll()`	Allows access unconditionally
`rememberMe()`	Allows access for users who are authenticated via remember-me

Most of the methods in table 5.1 provide essential security rules for request handling, but they're self-limiting, only enabling security rules as defined by those methods. Alternatively, you can use the `access()` method to provide a SpEL expression to declare richer security rules. Spring Security extends SpEL to include several security-specific values and functions, as listed in table 5.2.

Table 5.2 Spring Security extensions to the Spring Expression Language

Security expression	What it evaluates to
`authentication`	The user's authentication object
`denyAll`	Always evaluates to `false`
`hasAnyAuthority(String… authorities)`	`true` if the user has been granted any of the given authorities
`hasAnyRole(String… roles)`	`true` if the user has any of the given roles
`hasAuthority(String authority)`	`true` if the user has been granted the specified authority
`hasPermission(Object target, Object permission)`	`true` if the user has access to the provided target for the given permission
`hasPermission(Object target, String targetType, Object permission)`	`true` if the user has access to the provided target for the given permission
`hasRole(String role)`	`true` if the user has the given role
`hasIpAddress(String ipAddress)`	`true` if the request comes from the given IP address
`isAnonymous()`	`true` if the user is anonymous
`isAuthenticated()`	`true` if the user is authenticated
`isFullyAuthenticated()`	`true` if the user is fully authenticated (not authenticated with remember-me)
`isRememberMe()`	`true` if the user was authenticated via remember-me
`permitAll`	Always evaluates to `true`
`principal`	The user's principal object

As you can see, most of the security expression extensions in table 5.2 correspond to similar methods in table 5.1. In fact, using the access() method along with the `hasRole()` and `permitAll` expressions, you can rewrite the `SecurityFilterChain` configuration as follows.

Listing 5.7 Using Spring expressions to define authorization rules

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
  return http
    .authorizeRequests()
      .antMatchers("/design", "/orders").access("hasRole('USER')")
      .antMatchers("/", "/**").access("permitAll()")

    .and()
    .build();
}
```

This may not seem like a big deal at first. After all, these expressions only mirror what you already did with method calls. But expressions can be much more flexible. For instance, suppose that (for some crazy reason) you only wanted to allow users with `ROLE_USER` authority to create new tacos on Tuesdays (for example, on Taco Tuesday); you could rewrite the expression as shown in this modified version of the `SecurityFilterChain` bean method:

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
  return http
    .authorizeRequests()
      .antMatchers("/design", "/orders")
        .access("hasRole('USER') && " +
          "T(java.util.Calendar).getInstance().get("+
          "T(java.util.Calendar).DAY_OF_WEEK) == " +
          "T(java.util.Calendar).TUESDAY")
      .antMatchers("/", "/**").access("permitAll")

    .and()
    .build();
}
```

With SpEL-based security constraints, the possibilities are virtually endless. I'll bet that you're already dreaming up interesting security constraints based on SpEL.

The authorization needs for the Taco Cloud application are met by the simple use of `access()` and the SpEL expressions in listing 5.9. Now let's see about customizing the login page to fit the look of the Taco Cloud application.

5.3.2 Creating a custom login page

The default login page is much better than the clunky HTTP basic dialog box you started with, but it's still rather plain and doesn't quite fit into the look of the rest of the Taco Cloud application.

To replace the built-in login page, you first need to tell Spring Security what path your custom

login page will be at. That can be done by calling `formLogin()` on the `HttpSecurity` object:

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
  return http
    .authorizeRequests()
      .antMatchers("/design", "/orders").access("hasRole('USER')")
      .antMatchers("/", "/**").access("permitAll()")

    .and()
      .formLogin()
        .loginPage("/login")

    .and()
    .build();
}
```

Notice that before you call `formLogin()`, you bridge this section of configuration and the previous section with a call to `and()`. The `and()` method signifies that you're finished with the authorization configuration and are ready to apply some additional HTTP configuration. You'll use `and()` several times as you begin new sections of configuration.

After the bridge, you call `formLogin()` to start configuring your custom login form. The call to `loginPage()` after that designates the path where your custom login page will be provided. When Spring Security determines that the user is unauthenticated and needs to log in, it will redirect them to this path.

Now you need to provide a controller that handles requests at that path. Because your login page will be fairly simple—nothing but a view—it's easy enough to declare it as a view controller in `WebConfig`. The following `addViewControllers()` method sets up the login page view controller alongside the view controller that maps "/" to the home controller:

```
@Override
public void addViewControllers(ViewControllerRegistry registry) {
  registry.addViewController("/").setViewName("home");
  registry.addViewController("/login");
}
```

Finally, you need to define the login page view itself. Because you're using Thymeleaf as your template engine, the following Thymeleaf template should do fine:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">
  <head>
    <title>Taco Cloud</title>
  </head>

  <body>
    <h1>Login</h1>
    <img th:src="@{/images/TacoCloud.png}"/>

    <div th:if="${error}">
      Unable to login. Check your username and password.
    </div>
```

```
    <p>New here? Click
       <a th:href="@{/register}">here</a> to register.</p>

    <form method="POST" th:action="@{/login}" id="loginForm">
      <label for="username">Username: </label>
      <input type="text" name="username" id="username" /><br/>

      <label for="password">Password: </label>
      <input type="password" name="password" id="password" /><br/>

      <input type="submit" value="Login"/>
    </form>
  </body>
</html>
```

The key things to note about this login page are the path it posts to and the names of the username and password fields. By default, Spring Security listens for login requests at /login and expects that the username and password fields be named `username` and `password`. This is configurable, however. For example, the following configuration customizes the path and field names:

```
.and()
  .formLogin()
    .loginPage("/login")
    .loginProcessingUrl("/authenticate")
    .usernameParameter("user")
    .passwordParameter("pwd")
```

Here, you specify that Spring Security should listen for requests to /authenticate to handle login submissions. Also, the username and password fields should now be named `user` and `pwd`.

By default, a successful login will take the user directly to the page that they were navigating to when Spring Security determined that they needed to log in. If the user were to directly navigate to the login page, a successful login would take them to the root path (for example, the homepage). But you can change that by specifying a default success page:

```
.and()
  .formLogin()
    .loginPage("/login")
    .defaultSuccessUrl("/design")
```

As configured here, if the user were to successfully log in after directly going to the login page, they would be directed to the /design page.

Optionally, you can force the user to the design page after login, even if they were navigating elsewhere prior to logging in, by passing `true` as a second parameter to `defaultSuccessUrl`:

```
.and()
  .formLogin()
    .loginPage("/login")
    .defaultSuccessUrl("/design", true)
```

Signing in with a username and password is the most common way to authenticate in a web application. But let’s have a look at another way to authenticate users that uses someone else’s

login page.

5.3.3 Enabling third-party authentication

You may have seen links or buttons on your favorite website that say "Sign In with Facebook", "Login with Twitter", or something similar. Rather than asking a user to enter their credentials on a login page specific to the website, they offer a way to sign via another website like Facebook that they may already be logged into.

This type of authentication is based on OAuth2 or OpenID Connect (OIDC). Although OAuth2 is an authorization specification—and we'll talk more about how to use it to secure REST APIs in chapter 9—it can be also used to perform authentication via a third-party website. OpenID Connect is another security specification that is based on OAuth2 to formalize the interaction that takes place during a third-party authentication.

In order to employ this type of authentication in your Spring application, you'll need to add the OAuth2 client starter to the build:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-oauth2-client</artifactId>
</dependency>
```

Then, at very least you'll need to configure details about one or more OAuth2 or OpenID Connect servers that you want to be able to authenticate against. Spring Security supports sign in with Facebook, Google, GitHub, and Okta out of the box, but you can configure other clients by specifying a few extra properties.

The general set of properties you'll need to set for your application to act as an OAuth2/OpenID Connect client is as follows:

```
spring:
  security:
    oauth2:
      client:
        registration:
          <oauth2 or openid provider name>:
            clientId: <client id>
            clientSecret: <client secret>
            scope: <comma-separated list of requested scopes>
```

For example, suppose that for Taco Cloud, we want users to be able to sign in using Facebook. The following configuration in `application.yml` will setup the OAuth2 client:

```
spring:
  security:
    oauth2:
      client:
        registration:
          facebook:
            clientId: <facebook client id>
```

```
            clientSecret: <facebook client secret>
            scope: email, public_profile
```

The client ID and secret are the credentials that identify your application to Facebook. You can obtain a client ID and secret by creating a new application entry at https://developers.facebook.com/. The `scope` property specifies the access that the application will be granted. In this case, the application will have access the user's email address and the essential information from their public Facebook profile.

In a very simple application, this is all you will need. When the user attempts to access a page that requires authentication, their browser will redirect to Facebook. If they're not already logged into Facebook, they'll be greeted with the Facebook sign in page. After signing into Facebook, they'll be asked to authorize your application and grant the requested scope. Finally, they'll be redirected back to your application where they will have been authenticated.

If, however, you've customized security by declaring a `SecurityFilterChain` bean, then you'll need to enable OAuth2 login along with the rest of the security configuration:

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
  return http
    .authorizeRequests()
      .mvcMatchers("/design", "/orders").hasRole("USER")
      .anyRequest().permitAll()

    .and()
      .formLogin()
        .loginPage("/login")

    .and()
      .oauth2Login()

  ...

    .and()
    .build();
}
```

You may also want to offer both a traditional username/password login and third-party login. In that case, you can specify the login page in the configuration like this:

```
.and()
  .oauth2Login()
    .loginPage("/login")
```

This will cause the application to always take the user to the application-provided login page where they may choose to login with their username and password as usual. But you can also provide a link on that same login page that offers them the opportunity to login with Facebook. Such a link could look like this in the login page's HTML template:

```
<a th:href="/oauth2/authorization/facebook">Sign in with Facebook</a>
```

Now that you've dealt with logging in, let's flip to the other side of the authentication coin and

see how you can enable a user to log out.

5.3.4 Logging out

Just as important as logging into an application is logging out. To enable logout, you simply need to call `logout` on the `HttpSecurity` object:

```
.and()
  .logout()
```

This sets up a security filter that intercepts POST requests to /logout. Therefore, to provide logout capability, you just need to add a logout form and button to the views in your application:

```
<form method="POST" th:action="@{/logout}">
  <input type="submit" value="Logout"/>
</form>
```

When the user clicks the button, their session will be cleared, and they will be logged out of the application. By default, they'll be redirected to the login page where they can log in again. But if you'd rather they be sent to a different page, you can call `logoutSuccessUrl()` to specify a different post-logout landing page:

```
.and()
  .logout()
    .logoutSuccessUrl("/")
```

In this case, users will be sent to the homepage following logout.

5.3.5 Preventing cross-site request forgery

Cross-site request forgery (CSRF) is a common security attack. It involves subjecting a user to code on a maliciously designed web page that automatically (and usually secretly) submits a form to another application on behalf of a user who is often the victim of the attack. For example, a user may be presented with a form on an attacker's website that automatically posts to a URL on the user's banking website (which is presumably poorly designed and vulnerable to such an attack) to transfer money. The user may not even know that the attack happened until they notice money missing from their account.

To protect against such attacks, applications can generate a CSRF token upon displaying a form, place that token in a hidden field, and then stow it for later use on the server. When the form is submitted, the token is sent back to the server along with the rest of the form data. The request is then intercepted by the server and compared with the token that was originally generated. If the token matches, the request is allowed to proceed. Otherwise, the form must have been rendered by an evil website without knowledge of the token generated by the server.

Fortunately, Spring Security has built-in CSRF protection. Even more fortunate is that it's enabled by default and you don't need to explicitly configure it. You only need to make sure that

any forms your application submits include a field named `_csrf` that contains the CSRF token.

Spring Security even makes that easy by placing the CSRF token in a request attribute with the name `_csrf`. Therefore, you could render the CSRF token in a hidden field with the following in a Thymeleaf template:

```
<input type="hidden" name="_csrf" th:value="${_csrf.token}"/>
```

If you're using Spring MVC's JSP tag library or Thymeleaf with the Spring Security dialect, you needn't even bother explicitly including a hidden field. The hidden field will be rendered automatically for you.

In Thymeleaf, you just need to make sure that one of the attributes of the `<form>` element is prefixed as a Thymeleaf attribute. That's usually not a concern, as it's quite common to let Thymeleaf render the path as context relative. For example, the `th:action` attribute is all you need for Thymeleaf to render the hidden field for you:

```
<form method="POST" th:action="@{/login}" id="loginForm">
```

It's possible to disable CSRF support, but I'm hesitant to show you how. CSRF protection is important and easily handled in forms, so there's little reason to disable it. But if you insist on disabling it, you can do so by calling `disable()` like this:

```
.and()
  .csrf()
    .disable()
```

Again, I caution you not to disable CSRF protection, especially for production applications.

All of your web layer security is now configured for Taco Cloud. Among other things, you now have a custom login page and the ability to authenticate users against a JPA-backed user repository. Now let's see how you can obtain information about the logged-in user.

5.4 Applying method-level security

Although it's easy to think about security at the web request level, that's not always where security constraints are best applied. Sometimes it's better to verify that the user is authenticated and has been granted adequate authority at the point where the secured action will be performed.

For example, let's say that for administrative purposes there is a service class that includes a method for clearing out all orders from the database. Using an injected `OrderRepository`, that method might look a little something like this:

```
public void deleteAllOrders() {
  orderRepository.deleteAll();
}
```

Now, suppose that there is a controller that calls the `deleteAllOrders()` method as the result of a POST request:

```
@Controller
@RequestMapping("/admin")
public class AdminController {

  private OrderAdminService adminService;

  public AdminController(OrderAdminService adminService) {
    this.adminService = adminService;
  }

  @PostMapping("/deleteOrders")
  public String deleteAllOrders() {
    adminService.deleteAllOrders();
    return "redirect:/admin";
  }

}
```

It'd be easy enough to tweak `SecurityConfig` to ensure that only authorized users are allowed to perform that POST request:

```
.authorizeRequests()
  ...
  .antMatchers(HttpMethod.POST, "/admin/**")
          .access("hasRole('ADMIN')")
  ....
```

That's great and would prevent any unauthorized user from making a POST request to "/admin/deleteOrders" that would result in all orders disappearing from the database.

But suppose that some other controller method also calls `deleteAllOrders()`. You'd need to add more matchers to secure the requests for the other controllers that will need to be secured.

Instead, we can apply security directly on the `deleteAllOrders()` method like this:

```
@PreAuthorize("hasRole('ADMIN')")
public void deleteAllOrders() {
  orderRepository.deleteAll();
}
```

The `@PreAuthorize` annotation takes a SpEL expression and, if the expression evaluates to `false`, the method will not be invoked. On the other hand, if the expression evaluates to `true`, then the method will be allowed. In this case, `@PreAuthorize` is checking that the user has `ROLE_ADMIN` privilege. If so, then the method will be called and all orders will be deleted. Otherwise, it will be stopped in its tracks.

In the event that `@PreAuthorize` blocks the call, then Spring Security's `AccessDeniedException` will be thrown. This is an unchecked exception, so you don't need to catch it, unless you want to apply some custom behavior around the exception handling. If left uncaught, it will bubble up and eventually be caught by Spring Security's filters and handled

accordingly, either with an HTTP 403 page or perhaps redirecting to the login page if the user is unauthenticated.

In order for `@PreAuthorize` to work, you'll need to enable global method security. For that, you'll need to annotate the security configuration class with `@EnableGlobalMethodSecurity`:

```
@Configuration
@EnableGlobalMethodSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
  ...
}
```

You'll find `@PreAuthorize` to be a useful annotation for most method-level security needs. But know that it has a slightly less useful after-invocation counterpart in `@PostAuthorize`. The `@PostAuthorize` annotation works almost the same as the `@PreAuthorize` annotation, except that its expression won't be evaluated until after the target method is invoked and returns. This allows the expression to consider the return value of the method in deciding whether or not to permit the method invocation.

For example, suppose that there's a method that fetches an order by its ID. But you want to restrict it from being used except by admins or by the user who the order belongs to. You can use `@PostAuthorize` like this:

```
@PostAuthorize("hasRole('ADMIN') || " +
    "returnObject.user.username == authentication.name")
public TacoOrder getOrder(long id) {
  ...
}
```

In this case, the `returnObject` in the `TacoOrder` returned from the method. If it's `user` property has a `username` that is equal to the authentication's `name` property, then it will be allowed. In order to know that, though, the method will need to be executed so that it can return the `TacoOrder` object for consideration.

But wait! How can you secure a method from being invoked if the condition for applying security relies on the return value from the method invocation? That chicken and egg riddle is solved by allowing the method to be invoked, then throwing an `AccessDeniedException` if the expression returns `false`.

5.5 Knowing your user

Often, it's not enough to simply know that the user has logged in and what permissions they have been granted. It's usually important to also know who they are, so that you can tailor their experience.

For example, in `OrderController`, when you initially create the `TacoOrder` object that's bound to the order form, it'd be nice if you could prepopulate the `TacoOrder` with the user's name and

address, so they don't have to reenter it for each order. Perhaps even more important, when you save their order, you should associate the `TacoOrder` entity with the `User` that created the order.

To achieve the desired connection between an `TacoOrder` entity and a `User` entity, you need to add a new property to the `TacoOrder` class:

```
@Data
@Entity
@Table(name="Taco_Order")
public class TacoOrder implements Serializable {

...

  @ManyToOne
  private User user;

...

}
```

The `@ManyToOne` annotation on this property indicates that an order belongs to a single user, and, conversely, that a user may have many orders. (Because you're using Lombok, you won't need to explicitly define accessor methods for the property.)

In `OrderController`, the `processOrder()` method is responsible for saving an order. It will need to be modified to determine who the authenticated user is and to call `setUser()` on the `TacoOrder` object to connect the order with the user.

There are several ways to determine who the user is. These are a few of the most common ways:

- Inject a `Principal` object into the controller method
- Inject an `Authentication` object into the controller method
- Use `SecurityContextHolder` to get at the security context
- Use an `@AuthenticationPrincipal` annotated method

For example, you could modify `processOrder()` to accept a `java.security.Principal` as a parameter. You could then use the principal name to look up the user from a `UserRepository`:

```
@PostMapping
public String processOrder(@Valid TacoOrder order, Errors errors,
    SessionStatus sessionStatus,
    Principal principal) {

...

  User user = userRepository.findByUsername(
          principal.getName());

  order.setUser(user);

...

}
```

This works fine, but it litters code that's otherwise unrelated to security with security code. You can trim down some of the security-specific code by modifying `processOrder()` to accept an `Authentication` object as a parameter instead of a `Principal`:

```
@PostMapping
public String processOrder(@Valid TacoOrder order, Errors errors,
    SessionStatus sessionStatus,
    Authentication authentication) {

...

  User user = (User) authentication.getPrincipal();
  order.setUser(user);

...

}
```

With the `Authentication` in hand, you can call `getPrincipal()` to get the principal object which, in this case, is a `User`. Note that `getPrincipal()` returns a `java.util .Object`, so you need to cast it to `User`.

Perhaps the cleanest solution of all, however, is to simply accept a `User` object in `processOrder()`, but annotate it with `@AuthenticationPrincipal` so that it will be the authentication's principal:

```
@PostMapping
public String processOrder(@Valid Order order, Errors errors,
    SessionStatus sessionStatus,
    @AuthenticationPrincipal User user) {

  if (errors.hasErrors()) {
    return "orderForm";
  }

  order.setUser(user);

  orderRepo.save(order);
  sessionStatus.setComplete();

  return "redirect:/";
}
```

What's nice about `@AuthenticationPrincipal` is that it doesn't require a cast (as with `Authentication`), and it limits the security-specific code to the annotation itself. By the time you get the `User` object in `processOrder()`, it's ready to be used and assigned to the `TacoOrder`.

There's one other way of identifying who the authenticated user is, although it's a bit messy in the sense that it's very heavy with security-specific code. You can obtain an `Authentication` object from the security context and then request its principal like this:

```
Authentication authentication =
    SecurityContextHolder.getContext().getAuthentication();
User user = (User) authentication.getPrincipal();
```

Although this snippet is thick with security-specific code, it has one advantage over the other approaches described: it can be used anywhere in the application, not only in a controller's handler methods. This makes it suitable for use in lower levels of the code.

5.6 Summary

- Spring Security autoconfiguration is a great way to get started with security, but most applications will need to explicitly configure security to meet their unique security requirements.
- User details can be managed in user stores backed by relational databases, LDAP, or completely custom implementations.
- Spring Security automatically protects against CSRF attacks.
- Information about the authenticated user can be obtained via the `SecurityContext` object (returned from `SecurityContextHolder.getContext()`) or injected into controllers using `@AuthenticationPrincipal`.

6 Working with configuration properties

This chapter covers

- Fine-tuning autoconfigured beans
- Applying configuration properties to application components
- Working with Spring profiles

Do you remember when the iPhone first came out? A small slab of metal and glass hardly fit the description of what the world had come to recognize as a phone. And yet, it pioneered the modern smartphone era, changing everything about how we communicate. Although touch phones are in many ways easier and more powerful than their predecessor, the flip phone, when the iPhone was first announced, it was hard to imagine how a device with a single button could be used to place calls.

In some ways, Spring Boot autoconfiguration is like this. Autoconfiguration greatly simplifies Spring application development. But after a decade of setting property values in Spring XML configuration and calling setter methods on bean instances, it's not immediately apparent how to set properties on beans for which there's no explicit configuration.

Fortunately, Spring Boot provides a way with configuration properties. Configuration properties are nothing more than properties on `@ConfigurationProperties`-annotated beans in the Spring application context. Spring will inject values from one of several property sources—including JVM system properties, command-line arguments, and environment variables—into the bean properties. We'll see how to use `@ConfigurationProperties` on our own beans in section 6.2. But Spring Boot itself provides several `@ConfigurationProperties`-annotated beans that we'll configure first.

In this chapter, you're going to take a step back from implementing new features in the Taco

Cloud application to explore configuration properties. What you take away will no doubt prove useful as you move forward in the chapters that follow. We'll start by seeing how to employ configuration properties to fine-tune what Spring Boot automatically configures.

6.1 Fine-tuning autoconfiguration

Before we dive in too deeply with configuration properties, it's important to establish that there are two different (but related) kinds of configurations in Spring:

- *Bean wiring*— Configuration that declares application components to be created as beans in the Spring application context and how they should be injected into each other.
- *Property injection*— Configuration that sets values on beans in the Spring application context.

In Spring's XML and Java-based configuration, these two types of configurations are often declared explicitly in the same place. In Java configuration, an `@Bean`-annotated method is likely to both instantiate a bean and then set values to its properties. For example, consider the following `@Bean` method that declares a `DataSource` for an embedded H2 database:

```
@Bean
public DataSource dataSource() {
  return new EmbeddedDatabaseBuilder()
      .setType(H2)
      .addScript("taco_schema.sql")
      .addScripts("user_data.sql", "ingredient_data.sql")
      .build();
}
```

Here the `addScript()` and `addScripts()` methods set some `String` properties with the name of SQL scripts that should be applied to the database once the data source is ready. Whereas this is how you might configure a `DataSource` bean if you aren't using Spring Boot, autoconfiguration makes this method completely unnecessary.

If the H2 dependency is available in the run-time classpath, then Spring Boot automatically creates an appropriate `DataSource` bean in the Spring application context. The bean applies the SQL scripts schema.sql and data.sql.

But what if you want to name the SQL scripts something else? Or what if you need to specify more than two SQL scripts? That's where configuration properties come in. But before you can start using configuration properties, you need to understand where those properties come from.

6.1.1 Understanding Spring's environment abstraction

The Spring environment abstraction is a one-stop shop for any configurable property. It abstracts the origins of properties so that beans needing those properties can consume them from Spring itself. The Spring environment pulls from several property sources, including

- JVM system properties
- Operating system environment variables
- Command-line arguments
- Application property configuration files

It then aggregates those properties into a single source from which Spring beans can be injected. Figure 6.1 illustrates how properties from property sources flow through the Spring environment abstraction to Spring beans.

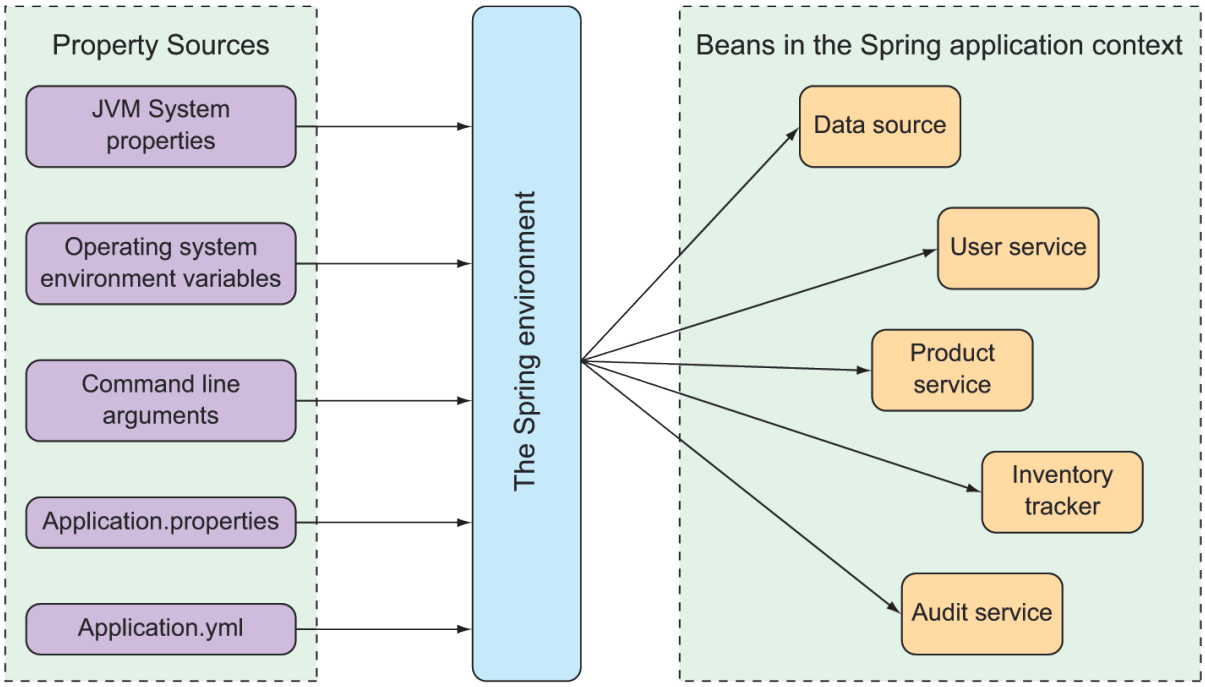

Figure 6.1 The Spring environment pulls properties from property sources and makes them available to beans in the application context.

The beans that are automatically configured by Spring Boot are all configurable by properties drawn from the Spring environment. As a simple example, suppose that you would like the application's underlying servlet container to listen for requests on some port other than the default port of 8080. To do that, specify a different port by setting the `server.port` property in src/main/resources/application.properties like this:

```
server.port=9090
```

Personally, I prefer using YAML when setting configuration properties. Therefore, instead of using application.properties, I might set the `server.port` value in src/main/resources/application.yml like this:

```
server:
  port: 9090
```

If you'd prefer to configure that property externally, you could also specify the port when

starting the application using a command-line argument:

```
$ java -jar tacocloud-0.0.5-SNAPSHOT.jar --server.port=9090
```

If you want the application to always start on a specific port, you could set it one time as an operating system environment variable:

```
$ export SERVER_PORT=9090
```

Notice that when setting properties as environment variables, the naming style is slightly different to accommodate restrictions placed on environment variable names by the operating system. That's OK. Spring is able to sort it out and interpret `SERVER_PORT` as `server.port` with no problems.

As I said, there are several ways of setting configuration properties. And when we get to chapter 14, you'll see yet another way of setting configuration properties in a centralized configuration server. In fact, there are several hundred configuration properties you can use to tweak and adjust how Spring beans behave. You've already seen a few: `server.port` in this chapter and `spring.security.user.name` and `spring.security.user.password` in the previous chapter.

It's impossible to examine all of the available configuration properties in this chapter. Even so, let's take a look at a few of the most useful configuration properties you might commonly encounter. We'll start with a few properties that let you tweak the autoconfigured data source.

6.1.2 Configuring a data source

At this point, the Taco Cloud application is still unfinished, but you'll have several more chapters to take care of that before you're ready to deploy the application. As such, the embedded H2 database you're using as a data source is perfect for your needs—for now. But once you take the application into production, you'll probably want to consider a more permanent database solution.

Although you could explicitly configure your own `DataSource` bean, that's usually unnecessary. Instead, it's simpler to configure the URL and credentials for your database via configuration properties. For example, if you were to start using a MySQL database, you might add the following configuration properties to application.yml:

```
spring:
  datasource:
    url: jdbc:mysql://localhost/tacocloud
    username: tacouser
    password: tacopassword
```

Although you'll need to add the appropriate JDBC driver to the build, you won't usually need to specify the JDBC driver class; Spring Boot can figure it out from the structure of the database

URL. But if there's a problem, you can try setting the `spring.datasource.driver-class-name` property:

```
spring:
  datasource:
    url: jdbc:mysql://localhost/tacocloud
    username: tacouser
    password: tacopassword
    driver-class-name: com.mysql.jdbc.Driver
```

Spring Boot uses this connection data when autoconfiguring the `DataSource` bean. The `DataSource` bean will be pooled using the HikariCP connection pool if it's available on the classpath. If not, Spring Boot looks for and uses one of these other connection pool implementations on the classpath:

- Tomcat JDBC Connection Pool
- Apache Commons DBCP2

Although these are the only connection pool options available through autoconfiguration, you're always welcome to explicitly configure a `DataSource` bean to use whatever connection pool implementation you'd like.

Earlier in this chapter, we suggested that there might be a way to specify the database initialization scripts to run when the application starts. In that case, the `spring.datasource.schema` and `spring.datasource.data` properties prove useful:

```
spring:
  datasource:
    schema:
    - order-schema.sql
    - ingredient-schema.sql
    - taco-schema.sql
    - user-schema.sql
    data:
    - ingredients.sql
```

Maybe explicit data source configuration isn't your style. Instead, perhaps you'd prefer to configure your data source in JNDI and have Spring look it up from there. In that case, set up your data source by configuring `spring.datasource.jndi-name`:

```
spring:
  datasource:
    jndi-name: java:/comp/env/jdbc/tacoCloudDS
```

If you set the `spring.datasource.jndi-name` property, the other data source connection properties (if set) are ignored.

6.1.3 Configuring the embedded server

You've already seen how to set the servlet container's port by setting `server.port`. What I didn't show you is what happens if `server.port` is set to 0:

Although you're explicitly setting `server.port` to 0, the server won't start on port 0. Instead, it'll start on a randomly chosen available port. This is useful when running automated integration tests to ensure that any concurrently running tests don't clash on a hard-coded port number. As you'll see in chapter 13, it's also useful when you don't care what port your application starts on because it's a microservice that will be looked up from a service registry.

But there's more to the underlying server than just a port. One of the most common things you'll need to do with the underlying container is to set it up to handle HTTPS requests. To do that, the first thing you must do is create a keystore using the JDK's `keytool` command-line utility:

```
$ keytool -keystore mykeys.jks -genkey -alias tomcat -keyalg RSA
```

You'll be asked several questions about your name and organization, most of which are irrelevant. But when asked for a password, remember what you choose. For the sake of this example, I chose letmein as the password.

Next, you'll need to set a few properties to enable HTTPS in the embedded server. You could specify them all on the command line, but that would be terribly inconvenient. Instead, you'll probably set them in the file's application.properties or application.yml. In application.yml, the properties might look like this:

```
server:
  port: 8443
  ssl:
    key-store: file:///path/to/mykeys.jks
    key-store-password: letmein
    key-password: letmein
```

Here the `server.port` property is set to 8443, a common choice for development HTTPS servers. The `server.ssl.key-store` property should be set to the path where the keystore file is created. Here it's shown with a `file://` URL to load it from the filesystem, but if you package it within the application JAR file, you'll use a `classpath:` URL to reference it. And both the `server.ssl.key-store-password` and `server .ssl.key-password` properties are set to the password that was given when creating the keystore.

With these properties in place, your application should be listening for HTTPS requests on port 8443. Depending on which browser you're using, you may encounter a warning about the server not being able to verify its identity. This is nothing to worry about when serving from localhost during development.

6.1.4 Configuring logging

Most applications provide some form of logging. And even if your application doesn't log anything directly, the libraries that your application uses will certainly log their activity.

By default, Spring Boot configures logging via Logback (http://logback.qos.ch) to write to the console at an INFO level. You've probably already seen plenty of INFO-level entries in the application logs as you've run the application and other examples.

For full control over the logging configuration, you can create a logback.xml file at the root of the classpath (in src/main/resources). Here's an example of a simple logback.xml file you might use:

```
<configuration>
  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
      <pattern>
        %d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n
      </pattern>
    </encoder>
  </appender>
  <logger name="root" level="INFO"/>
  <root level="INFO">
    <appender-ref ref="STDOUT" />
  </root>
</configuration>
```

Aside from the pattern used for logging, this Logback configuration is more or less equivalent to the default you'll get if you have no logback.xml file. But by editing logback.xml you can gain full control over your application's log files.

NOTE **The specifics of what can go into logback.xml are outside the scope of this book. Refer to Logback's documentation for more information.**

The most common changes you'll make to a logging configuration are to change the logging levels and perhaps to specify a file where the logs should be written. With Spring Boot configuration properties, you can make those changes without having to create a logback.xml file.

To set the logging levels, you create properties that are prefixed with `logging.level`, followed by the name of the logger for which you want to set the logging level. For instance, suppose you'd like to set the root logging level to WARN, but log Spring Security logs at a DEBUG level. The following entries in application.yml will take care of that for you:

```
logging:
  level:
    root: WARN
    org:
      springframework:
        security: DEBUG
```

Optionally, you can collapse the Spring Security package name to a single line for easier reading:

```
logging:
  level:
    root: WARN
    org.springframework.security: DEBUG
```

Now suppose that you want to write the log entries to the file TacoCloud.log at /var/logs/. The `logging.file.path` and `logging.file.name` properties can help achieve that:

```
logging:
  file:
    path: /var/logs/
    file: TacoCloud.log
  level:
    root: WARN
    org:
      springframework:
        security: DEBUG
```

Assuming that the application has write permissions to /var/logs/, the log entries will be written to /var/logs/TacoCloud.log. By default, the log files rotate once they reach 10 MB in size.

6.1.5 Using special property values

When setting properties, you aren't limited to declaring their values as hard-coded `String` and numeric values. Instead, you can derive their values from other configuration properties.

For example, suppose (for whatever reason) you want to set a property named `greeting.welcome` to echo the value of another property named `spring.application.name`. To achieve this, you could use the `${}` placeholder markers when setting `greeting.welcome`:

```
greeting:
  welcome: ${spring.application.name}
```

You can even embed that placeholder amidst other text:

```
greeting:
  welcome: You are using ${spring.application.name}.
```

As you've seen, configuring Spring's own components with configuration properties makes it easy to inject values into those components' properties and to fine-tune autoconfiguration. Configuration properties aren't exclusive to the beans that Spring creates. With a small amount of effort, you can take advantage of configuration properties in your own beans. Let's see how.

6.2 Creating your own configuration properties

As I mentioned earlier, configuration properties are nothing more than properties of beans that have been designated to accept configurations from Spring's environment abstraction. What I didn't mention is how those beans are designated to consume those configurations.

To support property injection of configuration properties, Spring Boot provides the `@ConfigurationProperties` annotation. When placed on any Spring bean, it specifies that the properties of that bean can be injected from properties in the Spring environment.

To demonstrate how `@ConfigurationProperties` works, suppose that you've added the following method to `OrderController` to list the authenticated user's past orders:

```
@GetMapping
public String ordersForUser(
    @AuthenticationPrincipal User user, Model model) {

  model.addAttribute("orders",
      orderRepo.findByUserOrderByPlacedAtDesc(user));

  return "orderList";
}
```

Along with that, you've also added the necessary `findByUserOrderByPlacedAtDesc()` method to `OrderRepository`:

```
List<Order> findByUserOrderByPlacedAtDesc(User user);
```

Notice that this repository method is named with a clause of `OrderByPlacedAtDesc`. The `OrderBy` portion specifies a property by which the results will be ordered—in this case, the `placedAt` property. The `Desc` at the end causes the ordering to be in descending order. Therefore, the list of orders returned will be sorted most recent to least recent.

As written, this controller method may be useful after the user has placed a handful of orders. But it could become a bit unwieldy for the most avid of taco connoisseurs. A few orders displayed in the browser are useful; a never-ending list of hundreds of orders is just noise. Let's say that you want to limit the number of orders displayed to the most recent 20 orders. You can change `ordersForUser()`

```
@GetMapping
public String ordersForUser(
    @AuthenticationPrincipal User user, Model model) {

  Pageable pageable = PageRequest.of(0, 20);
  model.addAttribute("orders",
      orderRepo.findByUserOrderByPlacedAtDesc(user, pageable));

  return "orderList";
}
```

along with the corresponding changes to `OrderRepository`:

```
List<Order> findByUserOrderByPlacedAtDesc(
        User user, Pageable pageable);
```

Here you've changed the signature of the `findByUserOrderByPlacedAtDesc()` method to accept a `Pageable` as a parameter. `Pageable` is Spring Data's way of selecting some subset of the results by a page number and page size. In the `ordersForUser()` controller method, you constructed a `PageRequest` object that implemented `Pageable` to request the first page (page zero) with a page size of 20 to get up to 20 of the most recently placed orders for the user.

Although this works fantastically, it leaves me a bit uneasy that you've hard-coded the page size. What if you later decide that 20 is too many orders to list, and you decide to change it to 10? Because it's hard-coded, you'd have to rebuild and redeploy the application.

Rather than hardcode the page size, you can set it with a custom configuration property. First, you need to add a new property called `pageSize` to `OrderController` and then annotate `OrderController` with `@ConfigurationProperties` as shown in the next listing.

Listing 6.1 Enabling configuration properties in `OrderController`

```
@Controller
@RequestMapping("/orders")
@SessionAttributes("order")
@ConfigurationProperties(prefix="taco.orders")
public class OrderController {

  private int pageSize = 20;

  public void setPageSize(int pageSize) {
    this.pageSize = pageSize;
  }

  ...
  @GetMapping
  public String ordersForUser(
      @AuthenticationPrincipal User user, Model model) {

    Pageable pageable = PageRequest.of(0, pageSize);
    model.addAttribute("orders",
        orderRepo.findByUserOrderByPlacedAtDesc(user, pageable));
    return "orderList";
  }

}
```

The most significant change made in listing 6.1 is the addition of the `@ConfigurationProperties` annotation. Its prefix attribute is set to `taco.orders`, which means that when setting the `pageSize` property, you need to use a configuration property named `taco.orders.pageSize`.

The new `pageSize` property defaults to 20. But you can easily change it to any value you want by setting a `taco.orders.pageSize` property. For example, you could set this property in

application.yml like this:

```
taco:
  orders:
    pageSize: 10
```

Or, if you need to make a quick change while in production, you can do so without having to rebuild and redeploy the application by setting the `taco.orders.pageSize` property as an environment variable:

```
$ export TACO_ORDERS_PAGESIZE=10
```

Any means by which a configuration property can be set can be used to adjust the page size of the recent orders page. Next, we'll look at how to set configuration data in property holders.

6.2.1 Defining configuration properties holders

There's nothing that says `@ConfigurationProperties` must be set on a controller or any other specific kind of bean. `@ConfigurationProperties` are in fact often placed on beans whose sole purpose in the application is to be holders of configuration data. This keeps configuration-specific details out of the controllers and other application classes. It also makes it easy to share common configuration properties among several beans that may make use of that information.

In the case of the `pageSize` property in `OrderController`, you could extract it to a separate class. The following listing uses the `OrderProps` class in such a way.

Listing 6.2 Extracting `pageSize` to a holder class

```
package tacos.web;
import org.springframework.boot.context.properties.
                                        ConfigurationProperties;
import org.springframework.stereotype.Component;
import lombok.Data;

@Component
@ConfigurationProperties(prefix="taco.orders")
@Data
public class OrderProps {

  private int pageSize = 20;

}
```

As you did with `OrderController`, the `pageSize` property defaults to 20 and `OrderProps` is annotated with `@ConfigurationProperties` to have a prefix of `taco.orders`. It's also annotated with `@Component` so that Spring component scanning will automatically discover it and create it as a bean in the Spring application context. This is important, as the next step is to inject the `OrderProps` bean into `OrderController`.

There's nothing particularly special about configuration property holders. They're beans that have their properties injected from the Spring environment. They can be injected into any other bean that needs those properties. For `OrderController`, this means removing the `pageSize` property from `OrderController` and instead injecting and using the `OrderProps` bean:

```
private OrderProps props;

public OrderController(OrderRepository orderRepo,
        OrderProps props) {
  this.orderRepo = orderRepo;
  this.props = props;
}

  ...

@GetMapping
public String ordersForUser(
    @AuthenticationPrincipal User user, Model model) {

  Pageable pageable = PageRequest.of(0, props.getPageSize());
  model.addAttribute("orders",
      orderRepo.findByUserOrderByPlacedAtDesc(user, pageable));

  return "orderList";
}
```

Now `OrderController` is no longer responsible for handling its own configuration properties. This keeps the code in `OrderController` slightly neater and allows you to reuse the properties in `OrderProps` in any other bean that may need them. Moreover, you're collecting configuration properties that pertain to orders in one place: the `OrderProps` class. If you need to add, remove, rename, or otherwise change the properties therein, you only need to apply those changes in `OrderProps`. And for testing purposes, it's easy to set configuration properties directly on a test-specific `OrderProps` and give it to the controller prior to the test.

For example, let's pretend that you're using the `pageSize` property in several other beans when you decide it would be best to apply some validation to that property to limit its values to no less than 5 and no more than 25. Without a holder bean, you'd have to apply validation annotations to `OrderController`, the `pageSize` property, and all other classes using that property. But because you've extracted `pageSize` into `OrderProps`, you only must make the changes to `OrderProps`:

```
package tacos.web;
import javax.validation.constraints.Max;
import javax.validation.constraints.Min;

import org.springframework.boot.context.properties.
                                        ConfigurationProperties;
import org.springframework.stereotype.Component;
import org.springframework.validation.annotation.Validated;

import lombok.Data;

@Component
@ConfigurationProperties(prefix="taco.orders")
@Data
@Validated
```

```
public class OrderProps {

  @Min(value=5, message="must be between 5 and 25")
  @Max(value=25, message="must be between 5 and 25")
  private int pageSize = 20;

}
```

Although you could as easily apply the `@Validated`, `@Min`, and `@Max` annotations to `OrderController` (and any other beans that can be injected with `OrderProps`), it would just clutter up `OrderController` that much more. With a configuration property holder bean, you've collected configuration property specifics in one place, leaving the classes that need those properties relatively clean.

6.2.2 Declaring configuration property metadata

Depending on your IDE, you may have noticed that the `taco.orders.pageSize` entry in application.yml (or application.properties) has a warning saying something like Unknown Property 'taco'. This warning appears because there's missing metadata concerning the configuration property you just created. Figure 6.2 shows what this looks like when I hover over the `taco` portion of the property in the Spring Tool Suite.

Configuration property metadata is completely optional and doesn't prevent configuration properties from working. But the metadata can be useful for providing some minimal documentation around the configuration properties, especially in the IDE. For example, when I hover over the `spring.security.user.password` property, I see what's shown in figure 6.3. Although the hover help you get is minimal, it can be enough to help understand what the property is used for and how to use it.

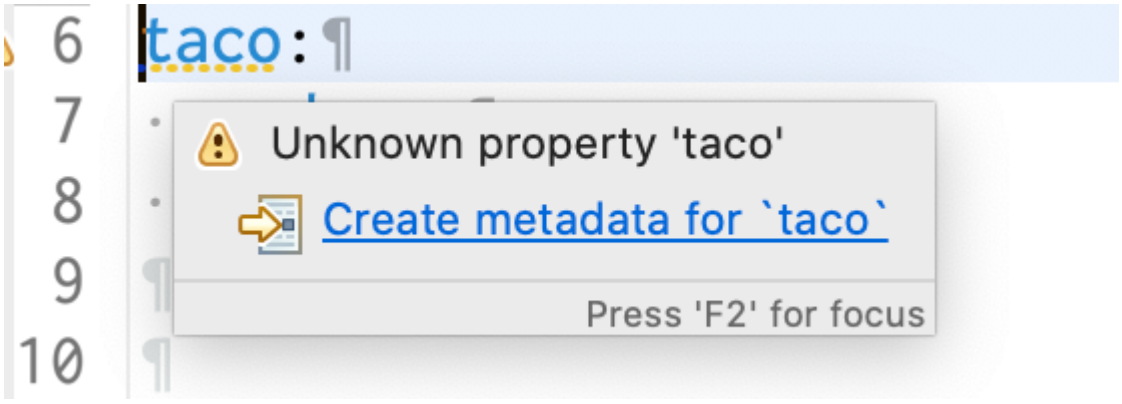

Figure 6.2 A warning for missing configuration property metadata

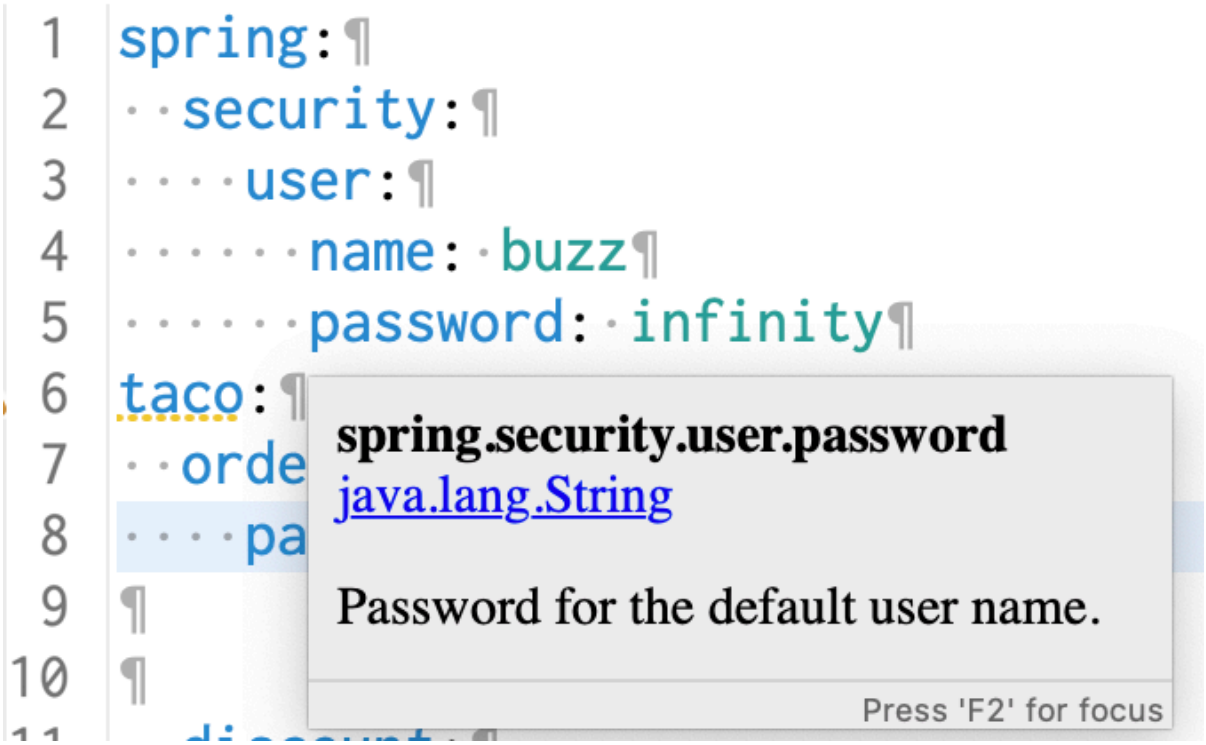

Figure 6.3 Hover documentation for configuration properties in the Spring Tool Suite

To help those who might use the configuration properties that you define—which might even be you—it's generally a good idea to create some metadata around those properties. At least it gets rid of those annoying yellow warnings in the IDE.

To create metadata for your custom configuration properties, you'll need to create a file under the META-INF (for example, in the project under src/main/resources/ META-INF) named additional-spring-configuration-metadata.json.

QUICK-FIXING MISSING METADATA.

If you're using the Spring Tool Suite, there's a quick-fix option for creating missing property metadata. Place your cursor on the line with the missing metadata warning and open the quick-fix pop up with CMD-1 on Mac or Ctrl-1 on Windows and Linux (see figure 6.4).

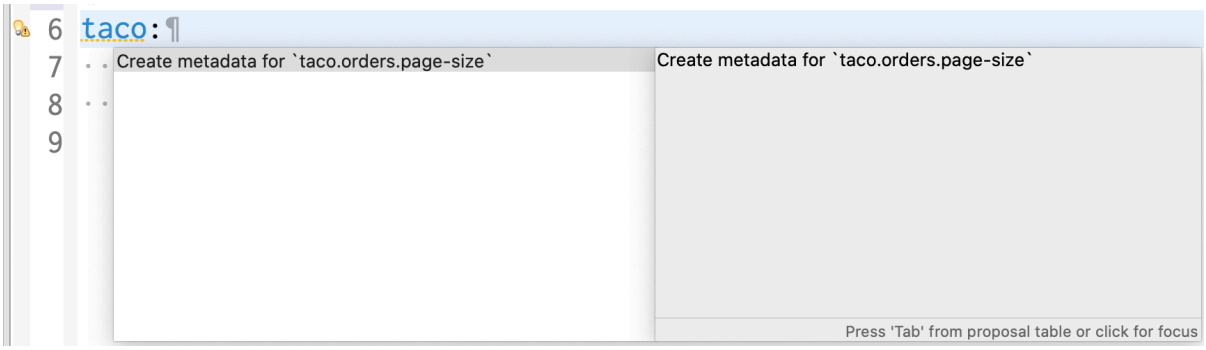

Figure 6.4 Creating configuration property metadata with the quick-fix pop up in Spring Tool Suite

Then select the "Create Metadata for …" option to add some metadata for the property. If it doesn't already exist, this quick fix will create a file in `META-INF/additional-spring-configuration-metadata.json` and fill it in with some metadata for the `pageSize` property:

```
{"properties": [{
  "name": "taco.orders.page-size",
  "type": "java.lang.String",
  "description": "A description for 'taco.orders.page-size'"
}]}
```

Notice that the property name referenced in the metadata is `taco.orders.page-size`, whereas the actual property name in `application.yml` is `pageSize`. Spring Boot's flexible property naming allows for variations in property names such that `taco.orders.page-size` is equivalent to `taco.orders.pageSize`, so it doesn't matter much which form you use.

The initial metadata written to `additional-spring-configuration-metadata.json` is a fine start, but you'll probably want to edit it a little. Firstly, the `pageSize` property isn't a `java.lang.String`, so you'll want to change it to `java.lang.Integer`. And the `description` property should be changed to be more descriptive of what `pageSize` is for. The following JSON shows what the metadata might look like after a few edits:

```
{"properties": [{
  "name": "taco.orders.page-size",
  "type": "java.lang.Integer",
  "description": "Sets the maximum number of orders to display in a list."
}]}
```

With that metadata in place, the warnings should be gone. What's more, if you hover over the `taco.orders.pageSize` property, you'll see the description shown in figure 6.5.

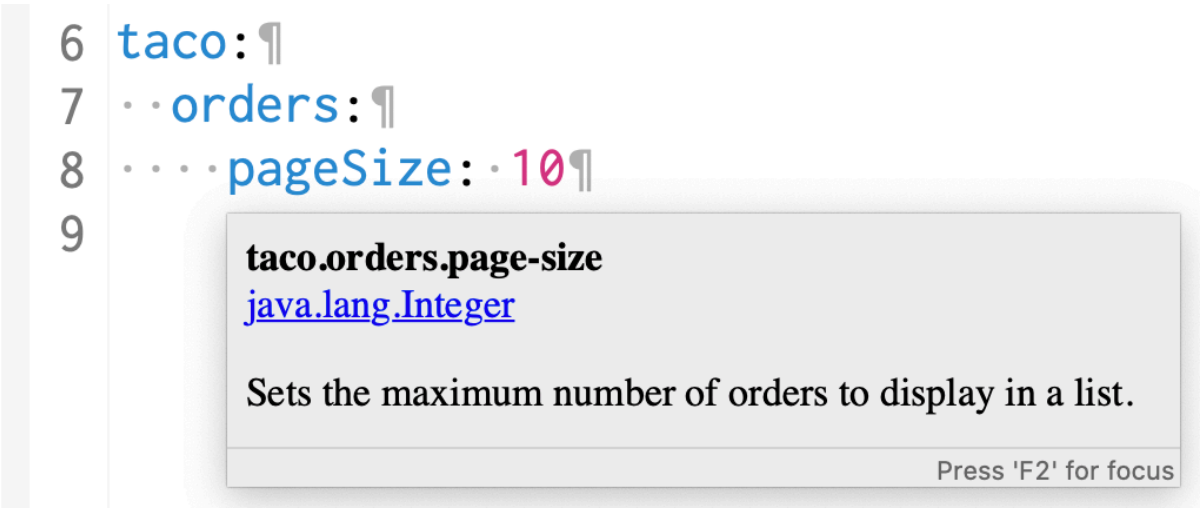

Figure 6.5 Hover help for custom configuration properties

Also, as shown in figure 6.6, you get autocompletion help from the IDE, just like Spring-provided configuration properties.

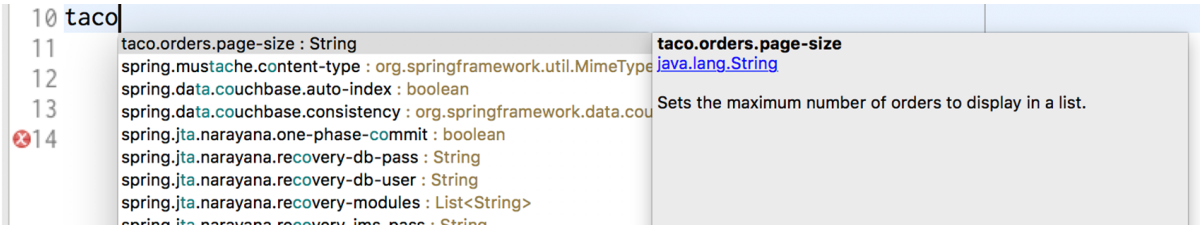

Figure 6.6 Configuration property metadata enables autocompletion of properties.

As you've seen, configuration properties are useful for tweaking both autoconfigured components as well as the details injected into your own application beans. But what if you need to configure different properties for different deployment environments? Let's take a look at how to use Spring profiles to set up environment-specific configuration.

6.3 Configuring with profiles

When applications are deployed to different run-time environments, usually some configuration details differ. The details of a database connection, for instance, are likely not the same in a development environment as in a quality assurance environment, and different still in a production environment. One way to configure properties uniquely in one environment over another is to use environment variables to specify configuration properties instead of defining them in application.properties and application.yml.

For instance, during development you can lean on the autoconfigured embedded H2 database. But in production you can set database configuration properties as environment variables like this:

```
% export SPRING_DATASOURCE_URL=jdbc:mysql://localhost/tacocloud
% export SPRING_DATASOURCE_USERNAME=tacouser
% export SPRING_DATASOURCE_PASSWORD=tacopassword
```

Although this will work, it's somewhat cumbersome to specify more than one or two configuration properties as environment variables. Moreover, there's no good way to track changes to environment variables or to easily roll back changes if there's a mistake.

Instead, I prefer to take advantage of Spring profiles. Profiles are a type of conditional configuration where different beans, configuration classes, and configuration properties are applied or ignored based on what profiles are active at runtime.

For instance, let's say that for development and debugging purposes, you want to use the embedded H2 database, and you want the logging levels for the Taco Cloud code to be set to DEBUG. But in production, you want to use an external MySQL database and set the logging levels to WARN. In the development situation, it's easy enough to not set any data-source properties and get the autoconfigured H2 database. And as for debug-level logging, you can set the `logging.level.tacos` property for the `tacos` base package to DEBUG in application.yml:

```
logging:
  level:
    tacos: DEBUG
```

This is precisely what you need for development purposes. But if you were to deploy this application in a production setting with no further changes to application.yml, you'd still have debug logging for the `tacos` package and an embedded H2 database. What you need is to define a profile with properties suited for production.

6.3.1 Defining profile-specific properties

One way to define profile-specific properties is to create yet another YAML or properties file containing only the properties for production. The name of the file should follow this convention: application-{profile name}.yml or application-{profile name}.properties. Then you can specify the configuration properties appropriate to that profile. For example, you could create a new file named application-prod.yml that contains the following properties:

```
spring:
  datasource:
    url: jdbc:mysql://localhost/tacocloud
    username: tacouser
    password: tacopassword
logging:
  level:
    tacos: WARN
```

Another way to specify profile-specific properties works only with YAML configuration. It involves placing profile-specific properties alongside non-profiled properties in application.yml, separated by three hyphens and the `spring.profiles` property to name the profile. When applying the production properties to application.yml in this way, the entire application.yml would look like this:

```
logging:
  level:
    tacos: DEBUG

---
spring:
  profiles: prod

  datasource:
    url: jdbc:mysql://localhost/tacocloud
    username: tacouser
    password: tacopassword

logging:
  level:
    tacos: WARN
```

As you can see, this application.yml file is divided into two sections by a set of triple hyphens (---). The second section specifies a value for `spring.profiles`, indicating that the properties

that follow apply to the prod profile. The first section, on the other hand, doesn't specify a value for `spring.profiles`. Therefore, its properties are common to all profiles or are defaults if the active profile doesn't otherwise have the properties set.

Regardless of which profiles are active when the application runs, the logging level for the `tacos` package will be set to DEBUG by the property set in the default profile. But if the profile named `prod` is active, then the `logging.level.tacos` property will be overridden with `WARN`. Likewise, if the `prod` profile is active, then the data-source properties will be set to use the external MySQL database.

You can define properties for as many profiles as you need by creating additional YAML or properties files named with the pattern application-{profile name}.yml or application-{profile name}.properties. Or, if you prefer, type three more dashes in application.yml along with another `spring.profiles` property to specify the profile name. Then add all of the profile-specific properties you need. Although there's no benefit to either approach, you might find that putting all profile configurations in a single YAML file works best when the number of properties is small, while distinct files for each profile is better when there are a large number of properties.

6.3.2 Activating profiles

Setting profile-specific properties will do no good unless those profiles are active. But how can you make a profile active? All it takes to make a profile active is to include it in the list of profile names given to the `spring.profiles.active` property. For example, you could set it in application.yml like this:

```
spring:
  profiles:
    active:
    - prod
```

But that's perhaps the worst possible way to set an active profile. If you set the active profile in application.yml, then that profile becomes the default profile, and you achieve none of the benefits of using profiles to separate the production-specific properties from development properties. Instead, I recommend that you set the active profile(s) with environment variables. On the production environment, you would set `SPRING_PROFILES_ACTIVE` like this:

```
% export SPRING_PROFILES_ACTIVE=prod
```

From then on, any applications deployed to that machine will have the `prod` profile active and the corresponding configuration properties would take precedence over the properties in the default profile.

If you're running the application as an executable JAR file, you might also set the active profile with a command-line argument like this:

```
% java -jar taco-cloud.jar --spring.profiles.active=prod
```

Note that the `spring.profiles.active` property name contains the plural word profiles. This means you can specify more than one active profile. Often, this is with a comma-separated list as when setting it with an environment variable:

```
% export SPRING_PROFILES_ACTIVE=prod,audit,ha
```

But in YAML, you'd specify it as a list like this:

```
spring:
  profiles:
    active:
    - prod
    - audit
    - ha
```

It's also worth noting that if you deploy a Spring application to Cloud Foundry, a profile named `cloud` is automatically activated for you. If Cloud Foundry is your production environment, you'll want to be sure to specify production-specific properties under the `cloud` profile.

As it turns out, profiles aren't useful only for conditionally setting configuration properties in a Spring application. Let's see how to declare beans specific to an active profile.

6.3.3 Conditionally creating beans with profiles

Sometimes it's useful to provide a unique set of beans for different profiles. Normally, any bean declared in a Java configuration class is created, regardless of which profile is active. But suppose there are some beans that you only need to be created if a certain profile is active. In that case, the `@Profile` annotation can designate beans as only being applicable to a given profile.

For instance, you have a `CommandLineRunner` bean declared in `TacoCloudApplication` that's used to load the embedded database with ingredient data when the application starts. That's great for development, but would be unnecessary (and undesirable) in a production application. To prevent the ingredient data from being loaded every time the application starts in a production deployment, you could annotate the `CommandLineRunner` bean method with `@Profile` like this:

```
@Bean
@Profile("dev")
public CommandLineRunner dataLoader(IngredientRepository repo,
      UserRepository userRepo, PasswordEncoder encoder) {

  ...

}
```

Or suppose that you need the `CommandLineRunner` created if either the dev profile or qa profile is active. In that case, you can list the profiles for which the bean should be created:

```
@Bean
```

```
@Profile({"dev", "qa"})
public CommandLineRunner dataLoader(IngredientRepository repo,
      UserRepository userRepo, PasswordEncoder encoder) {

  ...

}
```

Now the ingredient data will only be loaded if the `dev` or `qa` profiles are active. That would mean that you'd need to activate the `dev` profile when running the application in the development environment. It would be even more convenient if that `CommandLineRunner` bean were always created unless the `prod` profile is active. In that case, you can apply `@Profile` like this:

```
@Bean
@Profile("!prod")
public CommandLineRunner dataLoader(IngredientRepository repo,
      UserRepository userRepo, PasswordEncoder encoder) {

  ...

}
```

Here, the exclamation mark (`!`) negates the profile name. Effectively, it states that the `CommandLineRunner` bean will be created if the `prod` profile isn't active.

It's also possible to use `@Profile` on an entire `@Configuration`-annotated class. For example, suppose that you were to extract the `CommandLineRunner` bean into a separate configuration class named `DevelopmentConfig`. Then you could annotate `DevelopmentConfig` with `@Profile`:

```
@Profile({"!prod", "!qa"})
@Configuration
public class DevelopmentConfig {

  @Bean
  public CommandLineRunner dataLoader(IngredientRepository repo,
        UserRepository userRepo, PasswordEncoder encoder) {

    ...

  }

}
```

Here, the `CommandLineRunner` bean (as well as any other beans defined in `DevelopmentConfig`) will only be created if neither the `prod` nor `qa` profiles are active.

6.4 Summary

- Spring beans can be annotated with `@ConfigurationProperties` to enable injection of values from one of several property sources.
- Configuration properties can be set in command-line arguments, environment variables, JVM system properties, properties files, or YAML files, among other options.
- Configuration properties can be used to override autoconfiguration settings, including the ability to specify a data-source URL and logging levels.
- Spring profiles can be used with property sources to conditionally set configuration properties based on the active profile(s).

7 Creating REST services

This chapter covers

- Defining REST endpoints in Spring MVC
- Enabling hyperlinked REST resources
- Automatic repository-based REST endpoints

"The web browser is dead. What now?"

Several years ago, I heard someone suggest that the web browser was nearing legacy status and that something else would take over. But how could this be? What could possibly dethrone the near-ubiquitous web browser? How would we consume the growing number of sites and online services if not with a web browser? Surely these were the ramblings of a madman!

Fast-forward to the present day and it's clear that the web browser hasn't gone away. But it no longer reigns as the primary means of accessing the internet. Mobile devices, tablets, smart watches, and voice-based devices are now commonplace. And even many browser-based applications are actually running JavaScript applications rather than letting the browser be a dumb terminal for server-rendered content.

With such a vast selection of client-side options, many applications have adopted a common design where the user interface is pushed closer to the client and the server exposes an API through which all kinds of clients can interact with the backend functionality.

In this chapter, you're going to use Spring to provide a REST API for the Taco Cloud application. You'll use what you learned about Spring MVC in chapter 2 to create RESTful

endpoints with Spring MVC controllers. You'll also automatically expose REST endpoints for the Spring Data repositories you defined in chapters 3 and 4. Finally, we'll look at ways to test and secure those endpoints.

But first, you'll start by writing a few new Spring MVC controllers that expose backend functionality with REST endpoints to be consumed by a rich web frontend.

7.1 Writing RESTful controllers

I hope you don't mind, but while you were turning the page and reading the introduction to this chapter, I took it upon myself to reimagine the user interface for Taco Cloud. What you've been working with has been fine for getting started, but it lacked in the aesthetics department.

Figure 7.1 is just a sample of what the new Taco Cloud looks like. Pretty snazzy, huh?

And while I was spiffing up the Taco Cloud look, I also decided to build the frontend as a single-page application using the popular Angular framework. Ultimately, this new browser UI will replace the server-rendered pages you created in chapter 2. But for that to work, you'll need to create a REST API that the Angular-based[7] UI will communicate with to save and fetch taco data.

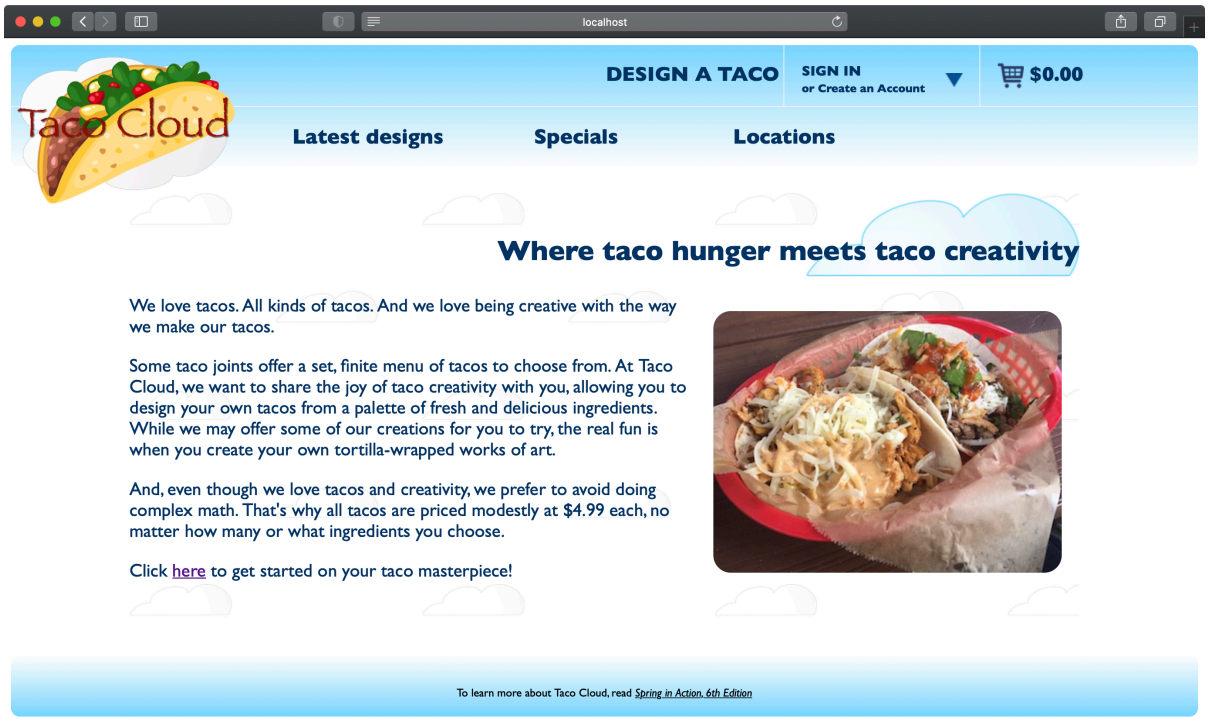

Figure 7.1 The new Taco Cloud home page

SIDEBAR **To SPA or not to SPA?**

You developed a traditional multipage application (MPA) with Spring MVC in chapter 2, and now you're replacing that with a single-page application (SPA) based on Angular. But I'm not suggesting that SPA is always a better choice than MPA.

Because presentation is largely decoupled from backend processing in a SPA, it affords the opportunity to develop more than one user interface (such as a native mobile application) for the same backend functionality. It also opens up the opportunity for integration with other applications that can consume the API. But not all applications require that flexibility, and MPA is a simpler design if all you need is to display information on a web page.

This isn't a book on Angular, so the code in this chapter will focus primarily on the backend Spring code. I'll show just enough Angular code to give you a feel for how the client side works. Rest assured that the complete set of code, including the Angular frontend, is available as part of the downloadable code for the book and at https://github.com/habuma/spring-in-action-6-samples. You may also be interested in reading Angular in Action by Jeremy Wilken (Manning, 2018) and *Angular Development with TypeScript, Second Edition* by Yakov Fain and Anton Moiseev (Manning, 2018).

In a nutshell, the Angular client code will communicate with an API that you'll create throughout this chapter by way of HTTP requests. In chapter 2 you used `@GetMapping` and `@PostMapping` annotations to fetch and post data to the server. Those same annotations will still come in handy as you define your REST API. In addition, Spring MVC supports a handful of other annotations for various types of HTTP requests, as listed in table 7.1.

Table 7.1 Spring MVC's HTTP request-handling annotations

Annotation	HTTP method	Typical use [8]
`@GetMapping`	HTTP `GET` requests	Reading resource data
`@PostMapping`	HTTP `POST` requests	Creating a resource
`@PutMapping`	HTTP `PUT` requests	Updating a resource
`@PatchMapping`	HTTP `PATCH` requests	Updating a resource
`@DeleteMapping`	HTTP `DELETE` requests	Deleting a resource
`@RequestMapping`	General purpose request handling; HTTP `method` specified in the method attribute	

To see these annotations in action, you'll start by creating a simple REST endpoint that fetches a few of the most recently created tacos.

7.1.1 Retrieving data from the server

One of the coolest things about Taco Cloud is that it allows taco fanatics to design their own taco creations and share them with their fellow taco lovers. To this end, Taco Cloud needs to be able to display a list of the most recently created tacos when the Latest Designs link is clicked.

In the Angular code I've defined a `RecentTacosComponent` that will display the most recently created tacos. The complete TypeScript code for `RecentTacosComponent` is shown in the next listing.

Listing 7.1 Angular component for displaying recent tacos

```
import { Component, OnInit, Injectable } from '@angular/core';
import { Http } from '@angular/http';
import { HttpClient } from '@angular/common/http';

@Component({
  selector: 'recent-tacos',
  templateUrl: 'recents.component.html',
  styleUrls: ['./recents.component.css']
})

@Injectable()
export class RecentTacosComponent implements OnInit {
  recentTacos: any;

  constructor(private httpClient: HttpClient) { }

  ngOnInit() {
    this.httpClient.get('http://localhost:8080/design/recent')   ❶
        .subscribe(data => this.recentTacos = data);
  }
}
```

❶ Fetches recent tacos from the server

Turn your attention to the `ngOnInit()` method. In that method, `RecentTacosComponent` uses the injected `Http` module to perform an HTTP `GET` request to http://localhost:8080/design/recent, expecting that the response will contain a list of taco designs, which will be placed in the `recentTacos` model variable. The view (in recents .component.html) will present that model data as HTML to be rendered in the browser. The end result might look something like figure 7.2, after three tacos have been created.

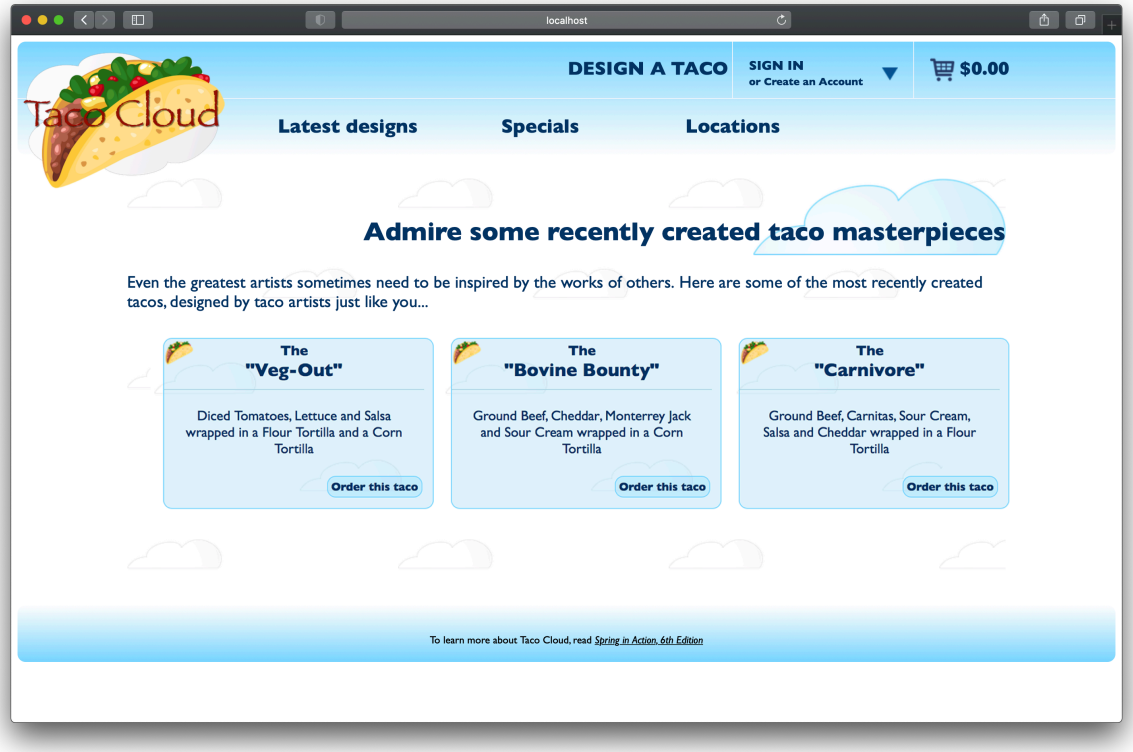

Figure 7.2 Displaying the most recently created tacos

The missing piece in this puzzle is an endpoint that handles `GET` requests for /design/ recent and responds with a list of recently designed tacos. You'll create a new controller to handle such a request. The next listing shows the controller for the job.

Listing 7.2 A RESTful controller for taco design API requests

```
package tacos.web.api;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.PageRequest;
import org.springframework.data.domain.Sort;
import org.springframework.web.bind.annotation.CrossOrigin;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import tacos.Taco;
import tacos.data.TacoRepository;

@RestController
@RequestMapping(path="/design",                          (1)
                produces="application/json")
@CrossOrigin(origins="*")          (2)
public class DesignTacoController {
  private TacoRepository tacoRepo;


  public DesignTacoController(TacoRepository tacoRepo) {
    this.tacoRepo = tacoRepo;
  }

  @GetMapping("/recent")
  public Iterable<Taco> recentTacos() {                  (3)
    PageRequest page = PageRequest.of(
            0, 12, Sort.by("createdAt").descending());
    return tacoRepo.findAll(page).getContent();
  }
}
```

(1) Handles requests for /design

(2) Allows cross-origin requests

(3) Fetches and returns recent taco designs

You may be thinking that this controller's name sounds familiar. In chapter 2 you created a `DesignTacoController` that handled similar types of requests. But where that controller was for the multipage Taco Cloud application, this new `DesignTacoController` is a REST controller, as indicated by the `@RestController` annotation.

The `@RestController` annotation serves two purposes. First, it's a stereotype annotation like `@Controller` and `@Service` that marks a class for discovery by component scanning. But most relevant to the discussion of REST, the `@RestController` annotation tells Spring that all handler methods in the controller should have their return value written directly to the body of the response, rather than being carried in the model to a view for rendering.

Alternatively, you could have annotated `DesignTacoController` with `@Controller`, just like with any Spring MVC controller. But then you'd need to also annotate all of the handler methods with `@ResponseBody` to achieve the same result. Yet another option would be to return a `ResponseEntity` object, which we'll discuss in a moment.

The `@RequestMapping` annotation at the class level works with the `@GetMapping` annotation on the `recentTacos()` method to specify that the `recentTacos()` method is responsible for handling `GET` requests for /design/recent (which is exactly what your Angular code needs).

You'll notice that the `@RequestMapping` annotation also sets a `produces` attribute. This specifies that any of the handler methods in `DesignTacoController` will only handle requests if the request's `Accept` header includes "application/json". Not only does this limit your API to only producing JSON results, it also allows for another controller (perhaps the `DesignTacoController` from chapter 2) to handle requests with the same paths, so long as those requests don't require JSON output. Even though this limits your API to being JSON-based (which is fine for your needs), you're welcome to set `produces` to an array of `String` for multiple content types. For example, to allow for XML output, you could add "text/xml" to the `produces` attribute:

```
@RequestMapping(path="/design",
                produces={"application/json", "text/xml"})
```

The other thing you may have noticed in listing 7.2 is that the class is annotated with `@CrossOrigin`. Because the Angular portion of the application will be running on a separate host and/or port from the API (at least for now), the web browser will prevent your Angular client from consuming the API. This restriction can be overcome by including CORS (Cross-Origin Resource Sharing) headers in the server responses. Spring makes it easy to apply CORS with the `@CrossOrigin` annotation. As applied here, `@CrossOrigin` allows clients from any domain to consume the API.

The logic within the `recentTacos()` method is fairly straightforward. It constructs a `PageRequest` object that specifies that you only want the first (0th) page of 12 results, sorted in descending order by the taco's creation date. In short, you want a dozen of the most recently created taco designs. The `PageRequest` is passed into the call to the `findAll()` method of `TacoRepository`, and the content of that page of results is returned to the client (which, as you saw in listing 7.1, will be used as model data to display to the user).

Now let's say that you want to offer an endpoint that fetches a single taco by its ID. By using a placeholder variable in the handler method's path and accepting a path variable, you can capture the ID and use it to look up the `Taco` object through the repository:

```
@GetMapping("/{id}")
public Taco tacoById(@PathVariable("id") Long id) {
  Optional<Taco> optTaco = tacoRepo.findById(id);
  if (optTaco.isPresent()) {
    return optTaco.get();
  }
  return null;
}
```

Because the controller's base path is /design, this controller method handles `GET` requests for `/design/{id}`, where the `{id}` portion of the path is a placeholder. The actual value in the request is given to the `id` parameter, which is mapped to the `{id}` placeholder by `@PathVariable`.

Inside of `tacoById()`, the `id` parameter is passed to the repository's `findById()` method to fetch the `Taco. findById()` returns an `Optional<Taco>` because there may not be a taco with the given ID. Therefore, you need to determine whether the ID matched a taco or not before returning a value. If it matches, you call `get()` on the `Optional<Taco>` object to return the actual `Taco`.

If the ID doesn't match any known tacos, you return `null`. This, however, is less than ideal. By returning `null`, the client receives a response with an empty body and an HTTP status code of 200 (OK). The client is handed a response it can't use, but the status code indicates everything is fine. A better approach would be to return a response with an HTTP 404 (NOT FOUND) status.

As it's currently written, there's no easy way to return a 404 status code from `tacoById()`. But if you make a few small tweaks, you can set the status code appropriately:

```
@GetMapping("/{id}")
public ResponseEntity<Taco> tacoById(@PathVariable("id") Long id) {
  Optional<Taco> optTaco = tacoRepo.findById(id);
  if (optTaco.isPresent()) {
    return new ResponseEntity<>(optTaco.get(), HttpStatus.OK);
  }
  return new ResponseEntity<>(null, HttpStatus.NOT_FOUND);
}
```

Now, instead of returning a `Taco` object, `tacoById()` returns a `ResponseEntity<Taco>`. If the taco is found, you wrap the `Taco` object in a `ResponseEntity` with an HTTP status of OK (which is what the behavior was before). But if the taco isn't found, you wrap a `null` in a `ResponseEntity` along with an HTTP status of NOT FOUND to indicate that the client is trying to fetch a taco that doesn't exist.

You now have the start of a Taco Cloud API for your Angular client—or any other kind of client, for that matter. For development testing purposes, you may also want to use command-line utilities like curl or HTTPie (https://httpie.org/) to poke about the API. For example, the following command line shows how you might fetch recently created tacos with curl:

```
$ curl localhost:8080/design/recent
```

Or like this if you prefer HTTPie:

```
$ http :8080/design/recent
```

But defining an endpoint that returns information is only the start. What if your API needs to

receive data from the client? Let's see how you can write controller methods that handle input on the requests.

7.1.2 Sending data to the server

So far your API is able to return a dozen of the most recently created tacos. But how did those tacos get created in the first place?

You haven't deleted any code from chapter 2 yet, so you still have the original `DesignTacoController` that displays a taco design form and handles form submission. That's a great way to get some test data in place to test the API you've created. But if you're going to transform Taco Cloud into a single-page application, you'll need to create Angular components and corresponding endpoints to replace that taco design form from chapter 2.

I've already handled the client code for the taco design form by defining a new Angular component named `DesignComponent` (in a file named design.component.ts). As it pertains to handling form submission, `DesignComponent` has an `onSubmit()` method that looks like this:

```
onSubmit() {
  this.httpClient.post(
      'http://localhost:8080/design',
      this.model, {
          headers: new HttpHeaders().set('Content-type', 'application/json'),
      }).subscribe(taco => this.cart.addToCart(taco));

  this.router.navigate(['/cart']);
}
```

In the `onSubmit()` method, the `post()` method of `HttpClient` is called instead of `get()`. This means that instead of fetching data from the API, you're sending data to the API. Specifically, you're sending a taco design, which is held in the `model` variable, to the API endpoint at /design with an HTTP `POST` request.

This means that you'll need to write a method in `DesignTacoController` to handle that request and save the design. By adding the following `postTaco()` method to `DesignTacoController`, you enable the controller to do exactly that:

```
@PostMapping(consumes="application/json")
@ResponseStatus(HttpStatus.CREATED)
public Taco postTaco(@RequestBody Taco taco) {
  return tacoRepo.save(taco);
}
```

Because `postTaco()` will handle an HTTP `POST` request, it's annotated with `@PostMapping` instead of `@GetMapping`. You're not specifying a `path` attribute here, so the `postTaco()` method will handle requests for /design as specified in the class-level `@RequestMapping` on `DesignTacoController`.

You do set the `consumes` attribute, however. The `consumes` attribute is to request input what

`produces` is to request output. Here you use `consumes` to say that the method will only handle requests whose `Content-type` matches `application/json`.

The `Taco` parameter to the method is annotated with `@RequestBody` to indicate that the body of the request should be converted to a `Taco` object and bound to the parameter. This annotation is important—without it, Spring MVC would assume that you want request parameters (either query parameters or form parameters) to be bound to the `Taco` object. But the `@RequestBody` annotation ensures that JSON in the request body is bound to the `Taco` object instead.

Once `postTaco()` has received the `Taco` object, it passes it to the `save()` method on the `TacoRepository`.

You may have also noticed that I've annotated the `postTaco()` method with `@ResponseStatus(HttpStatus.CREATED)`. Under normal circumstances (when no exceptions are thrown), all responses will have an HTTP status code of 200 (OK), indicating that the request was successful. Although an HTTP 200 response is always welcome, it's not always descriptive enough. In the case of a `POST` request, an HTTP status of 201 (CREATED) is more descriptive. It tells the client that not only was the request successful, but a resource was created as a result. It's always a good idea to use `@ResponseStatus` where appropriate to communicate the most descriptive and accurate HTTP status code to the client.

Although you've used `@PostMapping` to create a new `Taco` resource, `POST` requests can also be used to update resources. Even so, `POST` requests are typically used for resource creation and `PUT` and `PATCH` requests are used to update resources. Let's see how you can update data using `@PutMapping` and `@PatchMapping`.

7.1.3 Updating data on the server

Before you write any controller code for handling HTTP `PUT` or `PATCH` commands, you should take a moment to consider the elephant in the room: Why are there two different HTTP methods for updating resources?

Although it's true that `PUT` is often used to update resource data, it's actually the semantic opposite of `GET`. Whereas `GET` requests are for transferring data from the server to the client, `PUT` requests are for sending data from the client to the server.

In that sense, `PUT` is really intended to perform a wholesale *replacement* operation rather than an update operation. In contrast, the purpose of HTTP `PATCH` is to perform a patch or partial update of resource data.

For example, suppose you want to be able to change the address on an order. One way we could achieve this through the REST API is with a `PUT` request handled like this:

```
@PutMapping(path="/{orderId}", consumes="application/json")
public Order putOrder(
                      @PathVariable("orderId") Long orderId,
                      @RequestBody Order order) {
  order.setId(orderId);
  return repo.save(order);
}
```

This could work, but it would require that the client submit the complete order data in the `PUT` request. Semantically, `PUT` means "put this data at this URL," essentially replacing any data that's already there. If any of the order's properties are omitted, that property's value would be overwritten with `null`. Even the tacos in the order would need to be set along with the order data or else they'd be removed from the order.

If `PUT` does a wholesale replacement of the resource data, then how should you handle requests to do just a partial update? That's what HTTP `PATCH` requests and Spring's `@PatchMapping` are good for. Here's how you might write a controller method to handle a `PATCH` request for an order:

```
@PatchMapping(path="/{orderId}", consumes="application/json")
public Order patchOrder(@PathVariable("orderId") Long orderId,
                        @RequestBody Order patch) {

  Order order = repo.findById(orderId).get();
  if (patch.getDeliveryName() != null) {
    order.setDeliveryName(patch.getDeliveryName());
  }
  if (patch.getDeliveryStreet() != null) {
    order.setDeliveryStreet(patch.getDeliveryStreet());
  }
  if (patch.getDeliveryCity() != null) {
    order.setDeliveryCity(patch.getDeliveryCity());
  }
  if (patch.getDeliveryState() != null) {
    order.setDeliveryState(patch.getDeliveryState());
  }
  if (patch.getDeliveryZip() != null) {
    order.setDeliveryZip(patch.getDeliveryState());
  }
  if (patch.getCcNumber() != null) {
    order.setCcNumber(patch.getCcNumber());
  }
  if (patch.getCcExpiration() != null) {
    order.setCcExpiration(patch.getCcExpiration());
  }
  if (patch.getCcCVV() != null) {
    order.setCcCVV(patch.getCcCVV());
  }
  return repo.save(order);
}
```

The first thing to note here is that the `patchOrder()` method is annotated with `@PatchMapping` instead of `@PutMapping`, indicating that it should handle HTTP `PATCH` requests instead of `PUT` requests.

But the one thing you've no doubt noticed is that the `patchOrder()` method is a bit more involved than the `putOrder()` method. That's because Spring MVC's mapping annotations,

including `@PatchMapping` and `@PutMapping`, only specify what kinds of requests a method should handle. These annotations don't dictate how the request will be handled. Even though `PATCH` semantically implies a partial update, it's up to you to write code in the handler method that actually performs such an update.

In the case of the `putOrder()` method, you accepted the complete data for an order and saved it, adhering to the semantics of HTTP `PUT`. But in order for `patchMapping()` to adhere to the semantics of HTTP `PATCH`, the body of the method requires more intelligence. Instead of completely replacing the order with the new data sent in, it inspects each field of the incoming `TacoOrder` object and applies any non-`null` values to the existing order. This approach allows the client to only send the properties that should be changed and enables the server to retain existing data for any properties not specified by the client.

SIDEBAR **There's more than one way to PATCH**

The patching approach applied in the `patchOrder()` method has a couple of limitations:

- **If `null` values are meant to specify no change, how can the client indicate that a field should be set to `null`?**
- **There's no way of removing or adding a subset of items from a collection. If the client wants to add or remove an entry from a collection, it must send the complete altered collection.**

There's really no hard-and-fast rule about how `PATCH` requests should be handled or what the incoming data should look like. Rather than sending the actual domain data, a client could send a patch-specific description of the changes to be applied. Of course, the request handler would have to be written to handle patch instructions instead of the domain data.

In both `@PutMapping` and `@PatchMapping`, notice that the request path references the resource that's to be changed. This is the same way paths are handled by `@GetMapping`-annotated methods.

You've now seen how to fetch and post resources with `@GetMapping` and `@PostMapping`. And you've seen two different ways of updating a resource with `@PutMapping` and `@PatchMapping`. All that's left is handling requests to delete a resource.

7.1.4 Deleting data from the server

Sometimes data simply isn't needed anymore. In those cases, a client should be able to request that a resource be removed with an HTTP `DELETE` request.

Spring MVC's `@DeleteMapping` comes in handy for declaring methods that handle `DELETE`

requests. For example, let's say you want your API to allow for an order resource to be deleted. The following controller method should do the trick:

```
@DeleteMapping("/{orderId}")
@ResponseStatus(HttpStatus.NO_CONTENT)
public void deleteOrder(@PathVariable("orderId") Long orderId) {
  try {
    repo.deleteById(orderId);
  } catch (EmptyResultDataAccessException e) {}
}
```

By this point, the idea of another mapping annotation should be old hat to you. You've already seen `@GetMapping`, `@PostMapping`, `@PutMapping`, and `@PatchMapping`—each specifying that a method should handle requests for their corresponding HTTP methods. It will probably come as no surprise to you that `@DeleteMapping` is used to specify that the `deleteOrder()` method is responsible for handling `DELETE` requests for /orders/{orderId}.

The code within the method is what does the actual work of deleting an order. In this case, it takes the order ID, provided as a path variable in the URL, and passes it to the repository's `deleteById()` method. If the order exists when that method is called, it will be deleted. If the order doesn't exist, an `EmptyResultDataAccessException` will be thrown.

I've chosen to catch the `EmptyResultDataAccessException` and do nothing with it. My thinking here is that if you try to delete a resource that doesn't exist, the outcome is the same as if it did exist prior to deletion. That is, the resource will be nonexistent. Whether it existed before or not is irrelevant. Alternatively, I could've written `deleteOrder()` to return a `ResponseEntity`, setting the body to `null` and the HTTP status code to NOT FOUND.

The only other thing to take note of in the `deleteOrder()` method is that it's annotated with `@ResponseStatus` to ensure that the response's HTTP status is 204 (NO CONTENT). There's no need to communicate any resource data back to the client for a resource that no longer exists, so responses to `DELETE` requests typically have no body and therefore should communicate an HTTP status code to let the client know not to expect any content.

Your Taco Cloud API is starting to take shape. The client-side code can now easily consume this API to present ingredients, accept orders, and display recently created tacos. But there's something you can do that will make your API even easier for the client to consume. Let's look at how you can add hypermedia to the Tacoa Cloud API.

7.2 Enabling hypermedia

The API you've created thus far is fairly basic, but it does work as long as the client that consumes it is aware of the API's URL scheme. For example, a client may be hardcoded to know that it can obtain a list of recently created tacos by issuing a `GET` request for /design/recent. Likewise, it may be hardcoded to know that it can append the ID of any taco in that list to /design to get the URL for that particular taco resource.

Using hardcoded URL patterns and string manipulation is common among API client code. But imagine for a moment what would happen if the API's URL scheme were to change. The hardcoded client code would have an obsolete understanding of the API and would thus be broken. Hardcoding API URLs and using string manipulation on them makes the client code brittle.

Hypermedia as the Engine of Application State, or HATEOAS, is a means of creating self-describing APIs wherein resources returned from an API contain links to related resources. This enables clients to navigate an API with minimal understanding of the API's URLs. Instead, it understands relationships between the resources served by the API and uses its understanding of those relationships to discover the API's URLs as it traverses those relationships.

For example, suppose a client were to request a list of recently designed tacos. In its raw form, with no hyperlinks, the list of recent tacos would be received in the client with JSON that looks like this (with all but the first taco in the list clipped out for brevity's sake):

```
[
  {
    "id": 4,
    "name": "Veg-Out",
    "createdAt": "2018-01-31T20:15:53.219+0000",
    "ingredients": [
      {"id": "FLTO", "name": "Flour Tortilla", "type": "WRAP"},
      {"id": "COTO", "name": "Corn Tortilla", "type": "WRAP"},
      {"id": "TMTO", "name": "Diced Tomatoes", "type": "VEGGIES"},
      {"id": "LETC", "name": "Lettuce", "type": "VEGGIES"},
      {"id": "SLSA", "name": "Salsa", "type": "SAUCE"}
    ]
  },
  ...
]
```

If the client wished to fetch or perform some other HTTP operation on the taco itself, it would need to know (via hardcoding) that it could append the value of the `id` property to a URL whose path is /design. Likewise, if it wanted to perform an HTTP operation on one of the ingredients, it would need to know that it could append the value of the ingredient's `id` property to a URL whose path is /ingredients. In either case, it would also need to prefix that path with http:// or https:// and the hostname of the API.

In contrast, if the API is enabled with hypermedia, the API will describe its own URLs, relieving

the client of needing to be hardcoded with that knowledge. The same list of recently created tacos might look like the next listing if hyperlinks were embedded.

Listing 7.3 A list of taco resources that includes hyperlinks

```
{
  "_embedded": {
    "tacoResourceList": [
      {
        "name": "Veg-Out",
        "createdAt": "2018-01-31T20:15:53.219+0000",
        "ingredients": [
          {
            "name": "Flour Tortilla", "type": "WRAP",
            "_links": {
              "self": { "href": "http://localhost:8080/ingredients/FLTO" }
            }
          },
          {
            "name": "Corn Tortilla", "type": "WRAP",
            "_links": {
              "self": { "href": "http://localhost:8080/ingredients/COTO" }
            }
          },
          {
            "name": "Diced Tomatoes", "type": "VEGGIES",
            "_links": {
              "self": { "href": "http://localhost:8080/ingredients/TMTO" }
            }
          },
          {
            "name": "Lettuce", "type": "VEGGIES",
            "_links": {
              "self": { "href": "http://localhost:8080/ingredients/LETC" }
            }
          },
          {
            "name": "Salsa", "type": "SAUCE",
            "_links": {
              "self": { "href": "http://localhost:8080/ingredients/SLSA" }
            }
          }
        ],
        "_links": {
          "self": { "href": "http://localhost:8080/design/4" }
        }
      },

      ...
    ]
  },
  "_links": {
    "recents": {
      "href": "http://localhost:8080/design/recent"
    }
  }
}
```

This particular flavor of HATEOAS is known as HAL (Hypertext Application Language; http://stateless.co/hal_specification.html), a simple and commonly used format for embedding hyperlinks in JSON responses.

Although this list isn't as succinct as before, it does provide some useful information. Each

element in this new list of tacos includes a property named `_links` that contains hyperlinks for the client to navigate the API. In this example, both tacos and ingredients each have `self` links to reference those resources, and the entire list has a `recents` link that references itself.

Should a client application need to perform an HTTP request against a taco in the list, it doesn't need to be developed with any knowledge of what the taco resource's URL would look like. Instead, it knows to ask for the `self` link, which maps to http:// localhost:8080/design/4. If the client wants to deal with a particular ingredient, it only needs to follow the `self` link for that ingredient.

The Spring HATEOAS project brings hyperlink support to Spring. It offers a set of classes and resource assemblers that can be used to add links to resources before returning them from a Spring MVC controller.

To enable hypermedia in the Taco Cloud API, you'll need to add the Spring HATEOAS starter dependency to the build:

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

This starter not only adds Spring HATEOAS to the project's classpath, but also provides for autoconfiguration to enable Spring HATEOAS. All you need to do is rework your controllers to return resource types instead of domain types.

You'll start by adding hypermedia links to the list of recent tacos returned by a `GET` request to /design/recent.

7.2.1 Adding hyperlinks

Spring HATEOAS provides two primary types that represent hyperlinked resources: `EntityModel` and `CollectionModel`. The `EntityModel` type represents a single resource, whereas `CollectionModel` is a collection of resources. Both types are capable of carrying links to other resources. When returned from a Spring MVC REST controller method, the links they carry will be included in the JSON (or XML) received by the client.

To add hyperlinks to the list of recently created tacos, you'll need to revisit the `recentTacos()` method shown in listing 7.2. The original implementation returned a `List<Taco>`, which was fine at the time, but you're going to need it to return a `CollectionModel` object instead. The following listing shows a new implementation of `recentTacos()` that includes the first steps toward enabling hyperlinks in the recent tacos list.

Listing 7.4 Adding hyperlinks to resources

```
@GetMapping("/recent")
public CollectionModel<EntityModel<Taco>> recentTacos() {
  PageRequest page = PageRequest.of(
          0, 12, Sort.by("createdAt").descending());

  List<Taco> tacos = tacoRepo.findAll(page).getContent();
  CollectionModel<EntityModel<Taco>> recentResources =
      CollectionModel.wrap(tacos);

  recentResources.add(
      new Link("http://localhost:8080/design/recent", "recents"));
  return recentResources;
}
```

In this new version of `recentTacos()`, you no longer return the list of tacos directly. Instead, you use `CollectionModel.wrap()` to wrap the list of tacos as an instance of `CollectionModel<EntityModel<Taco>>`, which is ultimately returned from the method. But before returning the `CollectionModel` object, you add a link whose relationship name is `recents` and whose URL is http://localhost:8080/design/recent. As a consequence, the following snippet of JSON is included in the resource returned from the API request:

```
"_links": {
  "recents": {
    "href": "http://localhost:8080/design/recent"
  }
}
```

This is a good start, but you've still got some work to do. At this point, the only link you've added is to the entire list; no links are added to the taco resources themselves or to the ingredients of each taco. You'll add those soon. But first, let's address the hardcoded URL that you've given for the `recents` link.

Hardcoding a URL like this is a really bad idea. Unless your Taco Cloud ambitions are limited to only ever running the application on your own development machines, you need a way to not hardcode a URL with localhost:8080 in it. Fortunately, Spring HATEOAS provides help in the form of link builders.

The most useful of the Spring HATEOAS link builders is `WebMvcLinkBuilder`. This link builder is smart enough to know what the hostname is without you having to hardcode it. And it provides a handy fluent API to help you build links relative to the base URL of any controller.

Using `WebMvcLinkBuilder`, you can rewrite the hardcoded Link creation in `recentTacos()` with the following lines:

```
CollectionModel<EntityModel<Taco>> recentResources = CollectionModel.wrap(tacos);
recentResources.add(
    WebMvcLinkBuilder.linkTo(DesignTacoController.class)
                     .slash("recent")
                     .withRel("recents"));
```

Not only do you no longer need to hardcode the hostname, you also don't have to specify the /design path. Instead, you ask for a link to `DesignTacoController`, whose base path is /design. `WebMvcLinkBuilder` uses the controller's base path as the foundation of the `Link` object you're creating.

What's next is a call to one of my favorite methods in any Spring project: `slash()`. I love this method because it so succinctly describes exactly what it's going to do. It quite literally appends a slash (`/`) and the given value to the URL. As a result, the URL's path is /design/recent.

Finally, you specify a relation name for the `Link`. In this example, the relation is named `recents`.

Although I'm a big fan of the `slash()` method, `WebMvcLinkBuilder` has another method that can help eliminate any hardcoding associated with link URLs. Instead of calling `slash()`, you can call `linkTo()` by giving it a method on the controller to have `WebMvcLinkBuilder` derive the base URL from both the controller's base path and the method's mapped path. The following code uses the `linkTo()` method this way:

```
CollectionModel<EntityModel<Taco>> recentResources = CollectionModel.wrap(tacos);
recentResources.add(
    linkTo(methodOn(DesignTacoController.class).recentTacos())
.withRel("recents"));
```

Here I've decided to statically include the `linkTo()` and `methodOn()` methods (both from `WebMvcLinkBuilder`) to keep the code easier to read. The `methodOn()` method takes the controller class and lets you make a call to the `recentTacos()` method, which is intercepted by `WebMvcLinkBuilder` and used to determine not only the controller's base path, but also the path mapped to `recentTacos()`. Now the entire URL is derived from the controller's mappings, and absolutely no portion is hardcoded. Sweet!

7.2.2 Creating resource assemblers

Now you need to add links to the taco resource contained within the list. One option is to loop through each of the `EntityModel<Taco>` elements carried in the `CollectionModel` object, adding a `Link` to each individually. But that's a bit tedious and you'd need to repeat that looping code in the API wherever you return a list of taco resources.

We need a different tactic.

Rather than let `CollectionModel.wrap()` create a `EntityModel` object for each taco in the list, you're going to define a utility class that converts `Taco` objects to a new `TacoEntityModel` object. The `TacoEntityModel` object will look a lot like a `Taco`, but it will also be able to carry links. The next listing shows what a `TacoEntityModel` might look like.

Listing 7.5 A taco resource carrying domain data and a list of hyperlinks

```
package tacos.web.api;
import java.util.Date;
import java.util.List;

import org.springframework.hateoas.server.core.Relation;

import lombok.Getter;
import tacos.Ingredient;
import tacos.Taco;

@Relation(value="taco", collectionRelation="tacos")
public class TacoEntityModel extends EntityModel<TacoEntityModel> {

  private static final IngredientEntityModelAssembler
           ingredientAssembler = new IngredientEntityModelAssembler();

  @Getter
  private final String name;

  @Getter
  private final Date createdAt;

  @Getter
  private List<Ingredient> ingredients;

  public TacoResource(Taco taco) {
    this.name = taco.getName();
    this.createdAt = taco.getCreatedAt();
    this.ingredients = taco.getIngredients();
  }

}
```

In a lot of ways, `TacoEntityModel` isn't that different from the `Taco` domain type. They both have `name`, `createdAt`, and `ingredients` properties. But `TacoEntityModel` extends `CollectionModelSupport` to inherit a list of `Link` object and methods to manage the list of links.

What's more, `TacoEntityModel` doesn't include the `id` property from `Taco`. That's because there's no need to expose any database-specific IDs in the API. The resource's `self` link will serve as the identifier for the resource from the perspective of an API client.

NOTE **Domains and resources: separate or the same? Some Spring developers may choose to combine their domain and resource types into a single type by having their domain types extend `CollectionModelSupport`. There's no right or wrong answer as to which is the correct way. I chose to create a separate resource type so that `Taco` isn't unnecessarily cluttered with resource links for use cases where links aren't needed. Also, by creating a separate resource type, I was able to easily leave the `id` property out so that it won't be exposed in the API.**

`TacoEntityModel` has a single constructor that accepts a `Taco` and copies the pertinent

properties from the `Taco` to its own properties. This makes it easy to convert a single `Taco` object to a `TacoEntityModel`. But if you stop there, you'd still need looping to convert a list of `Taco` objects to a `CollectionModel<TacoEntityModel>`.

To aid in converting `Taco` objects to `TacoEntityModel` objects, you're also going to create a resource assembler. The following listing is what you'll need.

Listing 7.6 A resource assembler that assembles taco resources

```
package tacos.web.api;

import org.springframework.hateoas.server.mvc.RepresentationModelAssemblerSupport;

import tacos.Taco;

public class TacoEntityModelAssembler
       extends RepresentationModelAssemblerSupport<Taco, TacoEntityModel> {

  public TacoEntityModelAssembler() {
    super(DesignTacoController.class, TacoEntityModel.class);
  }

  @Override
  protected TacoEntityModel instantiateModel(Taco taco) {
    return new TacoEntityModel(taco);
  }

  @Override
  public TacoEntityModel toModel(Taco taco) {
    return createModelWithId(taco.getId(), taco);
  }

}
```

`TacoEntityModelAssembler` has a default constructor that informs the superclass (`RepresentationModelAssemblerSupport`) that it will be using `DesignTacoController` to determine the base path for any URLs in links it creates when creating a `TacoEntityModel`.

The `instantiateModel()` method is overridden to instantiate a `TacoEntityModel` given a `Taco`. This method would be optional if `TacoEntityModel` had a default constructor. In this case, however, `TacoEntityModel` requires construction with a `Taco`, so you're required to override it.

Finally, the `toModel()` method is the only method that's strictly mandatory when extending `RepresentationModelAssemblerSupport`. Here you're telling it to create a `TacoEntityModel` object from a `Taco`, and to automatically give it a `self` link with the URL being derived from the `Taco` object's `id` property.

On the surface, `toModel()` appears to have a similar purpose to `instantiateModel()`, but they serve slightly different purposes. Whereas `instantiateModel()` is intended to only instantiate a `EntityModel` object, `toModel()` is intended not only to create the `EntityModel` object, but also to populate it with links. Under the covers, `toModel()` will call `instantiateModel()`.

Now tweak the `recentTacos()` method to make use of `TacoEntityModelAssembler`:

```
@GetMapping("/recent")
public Resources<TacoEntityModel> recentTacos() {
  PageRequest page = PageRequest.of(
          0, 12, Sort.by("createdAt").descending());
  List<Taco> tacos = tacoRepo.findAll(page).getContent();
  List<TacoEntityModel> tacoResources =
      new TacoEntityModelAssembler().toModels(tacos);
  Resources<TacoEntityModel> recentResources =
      new Resources<TacoEntityModel>(tacoResources);
  recentResources.add(
      linkTo(methodOn(DesignTacoController.class).recentTacos())
      .withRel("recents"));
  return recentResources;
}
```

Rather than return a `CollectionModel<EntityModel<Taco>>`, `recentTacos()` now returns a `CollectionModel<TacoEntityModel>` to take advantage of your new `TacoEntityModel` type. After fetching the tacos from the repository, you pass the list of Taco objects to the `toModels()` method on a `TacoEntityModelAssembler`. This handy method cycles through all of the `Taco` objects, calling the `toModel()` method that you overrode in `TacoEntityModelAssembler` to create a list of `TacoEntityModel` objects.

With that `TacoEntityModel` list, you next create a `CollectionModel<TacoEntityModel>` object and then populate it with the `recents` links as in the prior version of `recentTacos()`.

At this point, a `GET` request to /design/recent will produce a list of tacos, each with a `self` link and a `recents` link on the list itself. But the ingredients will still be without a link. To address that, you'll create a new resource assembler for ingredients:

```
package tacos.web.api;

import org.springframework.hateoas.server.mvc.RepresentationModelAssemblerSupport;

import tacos.Ingredient;

class IngredientEntityModelAssembler extends
          RepresentationModelAssemblerSupport<Ingredient, IngredientEntityModel> {

  public IngredientEntityModelAssembler() {
    super(IngredientController.class, IngredientEntityModel.class);
  }

  @Override
  public IngredientEntityModel toModel(Ingredient ingredient) {
    return createModelWithId(ingredient.getId(), ingredient);
  }

  @Override
  protected IngredientEntityModel instantiateModel(Ingredient ingredient) {
    return new IngredientEntityModel(ingredient);
  }

}
```

As you can see, `IngredientEntityModelAssembler` is much like

`TacoEntityModelAssembler`, but it works with `Ingredient` and `IngredientEntityModel` objects instead of `Taco` and `TacoEntityModel` objects.

Speaking of `IngredientEntityModel`, it looks like this:

```
package tacos.web.api;
import org.springframework.hateoas.EntityModel;
import lombok.Getter;
import tacos.Ingredient;
import tacos.Ingredient.Type;

public class IngredientEntityModel extends EntityModel<IngredientEntityModel> {

  @Getter
  private String name;

  @Getter
  private Type type;

  public IngredientEntityModel(Ingredient ingredient) {
    this.name = ingredient.getName();
    this.type = ingredient.getType();
  }

}
```

As with `TacoEntityModel`, `IngredientEntityModel` extends `CollectionModelSupport` and copies pertinent properties from the domain type into its own set of properties (leaving out the `id` property).

All that's left is to make a slight change to `TacoEntityModel` so that it carries `IngredientEntityModel` objects instead of `Ingredient` objects:

```
package tacos.web.api;
import java.util.Date;

import org.springframework.hateoas.server.core.Relation;

import lombok.Getter;
import tacos.Taco;

@Relation(value="taco", collectionRelation="tacos")
public class TacoEntityModel extends EntityModel<TacoEntityModel> {

  private static final IngredientEntityModelAssembler
            ingredientAssembler = new IngredientEntityModelAssembler();

  @Getter
  private final String name;

  @Getter
  private final Date createdAt;

  @Getter
    private final CollectionModel<IngredientEntityModel> ingredients;

  public TacoEntityModel(Taco taco) {
        this.name = taco.getName();
        this.createdAt = taco.getCreatedAt();
        this.ingredients =
            ingredientAssembler.toCollectionModel(taco.getIngredients());
  }
```

```
}
```

This new version of `TacoEntityModel` creates a static final instance of `IngredientEntityModelAssembler` and uses its `toModel()` method to convert a given `Taco` object's list of `Ingredient` objects into a `CollectionModel<IngredientEntityModel>`.

Your recent tacos list is now completely outfitted with hyperlinks, not only for itself (the `recents` link), but also for all of its taco entries and the ingredients of those tacos. The response should look a lot like the JSON in listing 7.3.

You could stop here and move on to the next subject. But first I'll address something that's been bugging me about listing 7.3.

7.2.3 Naming embedded relationships

If you take a closer look at listing 7.3, you'll notice that the top-level elements look like this:

```
{
  "_embedded": {
    "tacoResourceList": [
      ...
    ]
  }
}
```

Most notably, let me draw your attention to the name `tacoResourceList` under `embedded`. That name was derived from the fact that the `CollectionModel` object was created from a `List<TacoEntityModel>`. Not that it's likely, but if you were to refactor the name of the `TacoEntityModel` class to something else, the field name in the resulting JSON would change to match it. This would likely break any clients coded to count on that name.

The `@Relation` annotation can help break the coupling between the JSON field name and the resource type class names as defined in Java. By annotating `TacoEntityModel` with `@Relation`, you can specify how Spring HATEOAS should name the field in the resulting JSON:

```
@Relation(value="taco", collectionRelation="tacos")
public class TacoEntityModel extends ResourceSupport {
  ...
}
```

Here you've specified that when a list of `TacoEntityModel` objects is used in a `CollectionModel` object, it should be named `tacos`. And although you're not making use of it in our API, a single `TacoEntityModel` object should be referred to in JSON as `taco`.

As a result, the JSON returned from /design/recent will now look like this (no matter what refactoring you may or may not perform on `TacoEntityModel`):

```
{
```

```
  "_embedded": {
    "tacos": [
      ...
    ]
  }
}
```

Spring HATEOAS makes adding links to your API rather straightforward and simple. Nonetheless, it did add several lines of code that you wouldn't otherwise need. Because of that, some developers may choose to not bother with HATEOAS in their APIs, even if it means that the client code is subject to breakage if the API's URL scheme changes. I encourage you to take HATEOAS seriously and not to take the lazy way out by not adding hyperlinks in your resources.

But if you insist on being lazy, then maybe there's a win-win scenario for you if you're using Spring Data for your repositories. Let's see how Spring Data REST can help you automatically create APIs based on the data repositories you created with Spring Data in chapter 3.

7.3 Enabling data-backed services

As you saw in chapter 3, Spring Data performs a special kind of magic by automatically creating repository implementations based on interfaces you define in your code. But Spring Data has another trick up its sleeve that can help you define APIs for your application.

Spring Data REST is another member of the Spring Data family that automatically creates REST APIs for repositories created by Spring Data. By doing little more than adding Spring Data REST to your build, you get an API with operations for each repository interface you've defined.

To start using Spring Data REST, you add the following dependency to your build:

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

Believe it or not, that's all that's required to expose a REST API in a project that's already using Spring Data for automatic repositories. By simply having the Spring Data REST starter in the build, the application gets auto-configuration that enables automatic creation of a REST API for any repositories that were created by Spring Data (including Spring Data JPA, Spring Data Mongo, and so on).

The REST endpoints that Spring Data REST creates are at least as good as (and possibly even better than) the ones you've created yourself. So at this point, feel free to do a little demolition work and remove any `@RestController`-annotated classes you've created up to this point before moving on.

To try out the endpoints provided by Spring Data REST, you can fire up the application and start

poking at some of the URLs. Based on the set of repositories you've already defined for Taco Cloud, you should be able to perform `GET` requests for tacos, ingredients, orders, and users.

For example, you can get a list of all ingredients by making a `GET` request for /ingredients. Using curl, you might get something that looks like this (abridged to only show the first ingredient):

```
$ curl localhost:8080/ingredients
{
  "_embedded" : {
    "ingredients" : [ {
      "name" : "Flour Tortilla",
      "type" : "WRAP",
      "_links" : {
        "self" : {
          "href" : "http://localhost:8080/ingredients/FLTO"
        },
        "ingredient" : {
          "href" : "http://localhost:8080/ingredients/FLTO"
        }
      }
    },
    ...
    ]
  },
  "_links" : {
    "self" : {
      "href" : "http://localhost:8080/ingredients"
    },
    "profile" : {
      "href" : "http://localhost:8080/profile/ingredients"
    }
  }
}
```

Wow! By doing nothing more than adding a dependency to your build, you're not only getting an endpoint for ingredients, but the resources that come back also contain hyperlinks! Pretending to be a client of this API, you can also use curl to follow the `self` link for the flour tortilla entry:

```
$ curl http://localhost:8080/ingredients/FLTO
{
  "name" : "Flour Tortilla",
  "type" : "WRAP",
  "_links" : {
    "self" : {
      "href" : "http://localhost:8080/ingredients/FLTO"
    },
    "ingredient" : {
      "href" : "http://localhost:8080/ingredients/FLTO"
    }
  }
}
```

To avoid getting too distracted, we won't waste much more time in this book digging into each and every endpoint and option that Spring Data REST has created. But you should know that it also supports `POST`, `PUT`, and `DELETE` methods for the endpoints it creates. That's right: you can `POST` to /ingredients to create a new ingredient and `DELETE` /ingredients/FLTO to remove flour tortillas from the menu.

One thing you might want to do is set a base path for the API so that its endpoints are distinct and don't collide with any controllers you write. (In fact, if you don't remove the `IngredientsController` you created earlier, it will interfere with the /ingredients endpoint provided by Spring Data REST.) To adjust the base path for the API, set the `spring.data.rest.base-path` property:

```
spring:
  data:
    rest:
      base-path: /api
```

This sets the base path for Spring Data REST endpoints to /api. Consequently, the ingredients endpoint is now /api/ingredients. Now give this new base path a spin by requesting a list of tacos:

```
$ curl http://localhost:8080/api/tacos
{
  "timestamp": "2018-02-11T16:22:12.381+0000",
  "status": 404,
  "error": "Not Found",
  "message": "No message available",
  "path": "/api/tacos"
}
```

Oh dear! That didn't work quite as expected. You have an `Ingredient` entity and an `IngredientRepository` interface, which Spring Data REST exposed with an /api/ ingredients endpoint. So if you have a `Taco` entity and a `TacoRepository` interface, why doesn't Spring Data REST give you an /api/tacos endpoint?

7.3.1 Adjusting resource paths and relation names

Actually, Spring Data REST does give you an endpoint for working with tacos. But as clever as Spring Data REST can be, it shows itself to be a tiny bit less awesome in how it exposes the tacos endpoint.

When creating endpoints for Spring Data repositories, Spring Data REST tries to pluralize the associated entity class. For the `Ingredient` entity, the endpoint is /ingredients. For the `TacoOrder` and `User` entities it's /orders and /users. So far, so good.

But sometimes, such as with "taco", it trips up on a word and the pluralized version isn't quite right. As it turns out, Spring Data REST pluralized "taco" as "tacoes", so to make a request for tacos, you must play along and request /api/tacoes:

```
% curl localhost:8080/api/tacoes
{
  "_embedded" : {
    "tacoes" : [ {
      "name" : "Carnivore",
      "createdAt" : "2018-02-11T17:01:32.999+0000",
      "_links" : {
        "self" : {
          "href" : "http://localhost:8080/api/tacoes/2"
```

```
        },
        "taco" : {
          "href" : "http://localhost:8080/api/tacoes/2"
        },
        "ingredients" : {
          "href" : "http://localhost:8080/api/tacoes/2/ingredients"
        }
      }
    }]
  },
  "page" : {
    "size" : 20,
    "totalElements" : 3,
    "totalPages" : 1,
    "number" : 0
  }
}
```

You may be wondering how I knew that “taco” would be mispluralized as “tacoes”. As it turns out, Spring Data REST also exposes a home resource that has links for all exposed endpoints. Just make a `GET` request to the API base path to get the goods:

```
$ curl localhost:8080/api
{
  "_links" : {
    "orders" : {
      "href" : "http://localhost:8080/api/orders"
    },
    "ingredients" : {
      "href" : "http://localhost:8080/api/ingredients"
    },
    "tacoes" : {
      "href" : "http://localhost:8080/api/tacoes{?page,size,sort}",
      "templated" : true
    },
    "users" : {
      "href" : "http://localhost:8080/api/users"
    },
    "profile" : {
      "href" : "http://localhost:8080/api/profile"
    }
  }
}
```

As you can see, the home resource shows the links for all of your entities. Everything looks good, except for the `tacoes` link, where both the relation name and the URL have the odd pluralization of “taco”.

The good news is that you don’t have to accept this little quirk of Spring Data REST. By adding a simple annotation to the `Taco` class, you can tweak both the relation name and that path:

```
@Data
@Entity
@RestResource(rel="tacos", path="tacos")
public class Taco {
  ...
}
```

The `@RestResource` annotation lets you give the entity any relation name and path you want. In this case, you’re setting them both to “tacos”. Now when you request the home resource, you see

the `tacos` link with correct pluralization:

```
"tacos" : {
  "href" : "http://localhost:8080/api/tacos{?page,size,sort}",
  "templated" : true
},
```

This also sorts out the path for the endpoint so that you can issue requests against /api/tacos to work with taco resources.

Speaking of sorting things out, let's look at how you can sort the results from Spring Data REST endpoints.

7.3.2 Paging and sorting

You may have noticed that the links in the home resource all offer optional `page`, `size`, and `sort` parameters. By default, requests to a collection resource such as /api/tacos will return up to 20 items per page from the first page. But you can adjust the page size and the page displayed by specifying the `page` and `size` parameters in your request.

For example, to request the first page of tacos where the page size is 5, you can issue the following `GET` request (using curl):

```
$ curl "localhost:8080/api/tacos?size=5"
```

Assuming that there are more than five tacos to be seen, you can request the second page of tacos by adding the `page` parameter:

```
$ curl "localhost:8080/api/tacos?size=5&page=1"
```

Notice that the `page` parameter is zero-based, which means that asking for page 1 is actually asking for the second page. (You'll also note that many command-line shells trip up over the ampersand in the request, which is why I quoted the whole URL in the preceding curl command.)

You could use string manipulation to add those parameters to the URL, but HATEOAS comes to the rescue by offering links for the first, last, next, and previous pages in the response:

```
"_links" : {
  "first" : {
    "href" : "http://localhost:8080/api/tacos?page=0&size=5"
  },
  "self" : {
    "href" : "http://localhost:8080/api/tacos"
  },
  "next" : {
    "href" : "http://localhost:8080/api/tacos?page=1&size=5"
  },
  "last" : {
    "href" : "http://localhost:8080/api/tacos?page=2&size=5"
  },
```

```
  "profile" : {
    "href" : "http://localhost:8080/api/profile/tacos"
  },
  "recents" : {
    "href" : "http://localhost:8080/api/tacos/recent"
  }
}
```

With these links, a client of the API need not keep track of what page it's on and concatenate the parameters to the URL. Instead, it must simply know to look for one of these page navigation links by its name and follow it.

The `sort` parameter lets you sort the resulting list by any property of the entity. For example, you need a way to fetch the 12 most recently created tacos for the UI to display. You can do that by specifying the following mix of paging and sorting parameters:

```
$ curl "localhost:8080/api/tacos?sort=createdAt,desc&page=0&size=12"
```

Here the `sort` parameter specifies that you should sort by the `createdDate` property and that it should be sorted in descending order (so that the newest tacos are first). The `page` and `size` parameters specify that you should see the first page of 12 tacos.

This is precisely what the UI needs in order to show the most recently created tacos. It's approximately the same as the /design/recent endpoint you defined in `DesignTacoController` earlier in this chapter.

There's a small problem, though. The UI code will need to be hardcoded to request the list of tacos with those parameters. Sure, it will work. But you're adding some brittleness to the client by making it too knowledgeable regarding how to construct an API request. It would be great if the client could look up the URL from a list of links. And it would be even more awesome if the URL were more succinct, like the /design/recent endpoint you had before.

7.3.3 Adding custom endpoints

Spring Data REST is great at creating endpoints for performing CRUD operations against Spring Data repositories. But sometimes you need to break away from the default CRUD API and create an endpoint that gets to the core of the problem.

There's absolutely nothing stopping you from implementing any endpoint you want in a `@RestController`-annotated bean to supplement what Spring Data REST automatically generates. In fact, you could resurrect the `DesignTacoController` from earlier in the chapter, and it would still work alongside the endpoints provided by Spring Data REST.

But when you write your own API controllers, their endpoints seem somewhat detached from the Spring Data REST endpoints in a couple of ways:

- Your own controller endpoints aren't mapped under Spring Data REST's base path. You

could force their mappings to be prefixed with whatever base path you want, including the Spring Data REST base path, but if the base path were to change, you'd need to edit the controller's mappings to match.

- Any endpoints you define in your own controllers won't be automatically included as hyperlinks in the resources returned by Spring Data REST endpoints. This means that clients won't be able to discover your custom endpoints with a relation name.

Let's address the concern about the base path first. Spring Data REST includes `@RepositoryRestController`, a new annotation for annotating controller classes whose mappings should assume a base path that's the same as the one configured for Spring Data REST endpoints. Put simply, all mappings in a `@RepositoryRestController`-annotated controller will have their path prefixed with the value of the `spring.data .rest.base-path` property (which you've configured as /api).

Rather than resurrect the `DesignTacoController`, which had several handler methods you won't need, you'll create a new controller that only contains the `recentTacos()` method. `RecentTacosController` in the next listing is annotated with `@RepositoryRestController` to adopt Spring Data REST's base path for its request mappings.

Listing 7.7 Applying Spring Data REST's base path to a controller

```
package tacos.web.api;
import static org.springframework.hateoas.server.mvc.WebMvcLinkBuilder.*;
import java.util.List;
import org.springframework.data.domain.PageRequest;
import org.springframework.data.domain.Sort;
import org.springframework.data.rest.webmvc.RepositoryRestController;
import org.springframework.hateoas.CollectionModel;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import tacos.Taco;
import tacos.data.TacoRepository;

@RepositoryRestController
public class RecentTacosController {

  private TacoRepository tacoRepo;

  public RecentTacosController(TacoRepository tacoRepo) {
    this.tacoRepo = tacoRepo;
  }

  @GetMapping(path="/tacos/recent", produces="application/hal+json")
  public ResponseEntity<CollectionModel<TacoEntityModel>> recentTacos() {
    PageRequest page = PageRequest.of(
                          0, 12, Sort.by("createdAt").descending());
    List<Taco> tacos = tacoRepo.findAll(page).getContent();

    CollectionModel<TacoEntityModel> tacoResources =
            new TacoEntityModelAssembler().toCollectionModel(tacos);
    CollectionModel<TacoEntityModel> recentResources =
            new CollectionModel<TacoEntityModel>(tacoResources);

    recentResources.add(
        linkTo(methodOn(RecentTacosController.class).recentTacos())
            .withRel("recents"));
    return new ResponseEntity<>(recentResources, HttpStatus.OK);
  }

}
```

Even though `@GetMapping` is mapped to the path /tacos/recent, the `@RepositoryRestController` annotation at the class level will ensure that it will be prefixed with Spring Data REST's base path. As you've configured it, the `recentTacos()` method will handle `GET` requests for /api/tacos/recent.

One important thing to notice is that although `@RepositoryRestController` is named similarly to `@RestController`, it doesn't carry the same semantics as `@RestController`. Specifically, it doesn't ensure that values returned from handler methods are automatically written to the body of the response. Therefore you need to either annotate the method with `@ResponseBody` or return a `ResponseEntity` that wraps the response data. Here you chose to return a `ResponseEntity`.

With `RecentTacosController` in play, requests for /api/tacos/recent will return up to 15 of the most recently created tacos, without the need for paging and sorting parameters in the URL. But it still doesn't appear in the hyperlinks list when requesting /api/tacos. Let's fix that.

7.3.4 Adding custom hyperlinks to Spring Data endpoints

If the recent tacos endpoint isn't among the hyperlinks returned from /api/tacos, how will a client know how to fetch the most recent tacos? It'll either have to guess or use the paging and sorting parameters. Either way, it'll be hardcoded in the client code, which isn't ideal.

By declaring a resource processor bean, however, you can add links to the list of links that Spring Data REST automatically includes. Spring Data HATEOAS offers `RepresentationModelProcessor`, an interface for manipulating resources before they're returned through the API. For your purposes, you need an implementation of `RepresentationModelProcessor` that adds a `recents` link to any resource of type `PagedModels<EntityModel<Taco>>` (the type returned for the /api/tacos endpoint). The next listing shows a bean declaration method that defines such a `RepresentationModelProcessor`.

Listing 7.8 Adding custom links to a Spring Data REST endpoint

```
@Bean
public RepresentationModelProcessor<PagedModel<EntityModel<Taco>>>
  tacoProcessor(EntityLinks links) {

  return new RepresentationModelProcessor<PagedModel<EntityModel<Taco>>>() {
    @Override
    public PagedModel<EntityModel<Taco>> process(
            PagedModel<EntityModel<Taco>> resource) {
      resource.add(
          links.linkFor(Taco.class)
               .slash("recent")
               .withRel("recents"));
      return resource;
    }
  };
}
```

The `RepresentationModelProcessor` shown in listing 7.8 is defined as an anonymous inner class and declared as a bean to be created in the Spring application context. Spring HATEOAS will discover this bean (as well as any other beans of type `RepresentationModelProcessor`) automatically and will apply them to the appropriate resources. In this case, if a `PagedModels<EntityModel<Taco>>` is returned from a controller, it will receive a link for the most recently created tacos. This includes the response for requests for /api/tacos.

7.4 Summary

- REST endpoints can be created with Spring MVC, with controllers that follow the same programming model as browser-targeted controllers.
- Controller handler methods can either be annotated with `@ResponseBody` or return `ResponseEntity` objects to bypass the model and view and write data directly to the response body.
- The `@RestController` annotation simplifies REST controllers, eliminating the need to use `@ResponseBody` on handler methods.
- Spring HATEOAS enables hyperlinking of resources returned from Spring MVC controllers.
- Spring Data repositories can automatically be exposed as REST APIs using Spring Data REST.

8 Consuming REST services

This chapter covers

- Using `RestTemplate` to consume REST APIs
- Navigating hypermedia APIs with Traverson

Have you ever gone to a movie and, as the movie starts, discovered that you were the only person in the theater? It certainly is a wonderful experience to have what is essentially a private viewing of a movie. You can pick whatever seat you want, talk back to the characters onscreen, and maybe even open your phone and tweet about it without anyone getting angry for disrupting their movie-watching experience. And the best part is that nobody else is there ruining the movie for you, either!

This hasn't happened to me often. But when it has, I have wondered what would have happened if I hadn't shown up. Would they still have shown the film? Would the hero still have saved the day? Would the theater staff still have cleaned the theater after the movie was over?

A movie without an audience is kind of like an API without a client. It's ready to accept and provide data, but if the API is never invoked, is it really an API? Like Schrödinger's cat, we can't know if the API is active or returning HTTP 404 responses until we issue a request to it.

In the previous chapter, we focused on defining REST endpoints that can be consumed by some client external to your application. Although the driving force for developing such an API was a single-page Angular application that served as the Taco Cloud website, the reality is that the client could be any application, in any language—even another Java application.

It's not uncommon for Spring applications to both provide an API and make requests to another application's API. In fact, this is becoming prevalent in the world of microservices. Therefore,

it's worthwhile to spend a moment looking at how to use Spring to interact with REST APIs.

A Spring application can consume a REST API with

- `RestTemplate` — A straightforward, synchronous REST client provided by the core Spring Framework.
- *Traverson*— A hyperlink-aware, synchronous REST client provided by Spring HATEOAS. Inspired from a JavaScript library of the same name.
- `WebClient` — A reactive, asynchronous REST client.

I'll defer discussion of `WebClient` until we cover Spring's reactive web framework in chapter 11. For now, we'll focus on the other two REST clients, starting with `RestTemplate`.

8.1 Consuming REST endpoints with RestTemplate

There's a lot that goes into interacting with a REST resource from the client's perspective—mostly tedium and boilerplate. Working with low-level HTTP libraries, the client needs to create a client instance and a request object, execute the request, interpret the response, map the response to domain objects, and handle any exceptions that may be thrown along the way. And all of this boilerplate is repeated, regardless of what HTTP request is sent.

To avoid such boilerplate code, Spring provides `RestTemplate`. Just as `JdbcTemplate` handles the ugly parts of working with JDBC, `RestTemplate` frees you from dealing with the tedium of consuming REST resources.

`RestTemplate` provides 41 methods for interacting with REST resources. Rather than examine all of the methods that it offers, it's easier to consider only a dozen unique operations, each overloaded to equal the complete set of 41 methods. The 12 operations are described in table 8.1.

Table 8.1 RestTemplate defines 12 unique operations, each of which is overloaded, providing a total of 41 methods. (continued)

Method	Description
`delete(…)`	Performs an HTTP `DELETE` request on a resource at a specified URL
`exchange(…)`	Executes a specified HTTP method against a URL, returning a `ResponseEntity` containing an object mapped from the response body
`execute(…)`	Executes a specified HTTP method against a URL, returning an object mapped from the response body
`getForEntity(…)`	Sends an HTTP `GET` request, returning a `ResponseEntity` containing an object mapped from the response body
`getForObject(…)`	Sends an HTTP `GET` request, returning an object mapped from a response body
`headForHeaders(…)`	Sends an HTTP `HEAD` request, returning the HTTP headers for the specified resource URL
`optionsForAllow(…)`	Sends an HTTP `OPTIONS` request, returning the Allow header for the specified URL
`patchForObject(…)`	Sends an HTTP `PATCH` request, returning the resulting object mapped from the response body
`postForEntity(…)`	`POST`s data to a URL, returning a ResponseEntity containing an object mapped from the response body
`postForLocation(…)`	`POST`s data to a URL, returning the URL of the newly created resource
`postForObject(…)`	`POST`s data to a URL, returning an object mapped from the response body
`put(…)`	`PUT`s resource data to the specified URL

With the exception of `TRACE`, `RestTemplate` has at least one method for each of the standard HTTP methods. In addition, `execute()` and `exchange()` provide lower-level, general-purpose methods for sending requests with any HTTP method.

Most of the methods in table 8.1 are overloaded into three method forms:

- One accepts a `String` URL specification with URL parameters specified in a variable argument list.
- One accepts a `String` URL specification with URL parameters specified in a `Map<String,String>`.
- One accepts a `java.net.URI` as the URL specification, with no support for parameterized URLs.

Once you get to know the 12 operations provided by `RestTemplate` and how each of the variant forms works, you'll be well on your way to writing resource-consuming REST clients.

To use `RestTemplate`, you'll either need to create an instance at the point you need it

```
RestTemplate rest = new RestTemplate();
```

or you can declare it as a bean and inject it where you need it:

```
@Bean
public RestTemplate restTemplate() {
  return new RestTemplate();
}
```

Let's survey `RestTemplate`'s operations by looking at those that support the four primary HTTP methods: `GET`, `PUT`, `DELETE`, and `POST`. We'll start with `getForObject()` and `getForEntity()`—the `GET` methods.

8.1.1 GETting resources

Suppose that you want to fetch an ingredient from the Taco Cloud API. Assuming that the API isn't HATEOAS-enabled, you can use `getForObject()` to fetch the ingredient. For example, the following code uses `RestTemplate` to fetch an `Ingredient` object by its ID:

```
public Ingredient getIngredientById(String ingredientId) {
  return rest.getForObject("http://localhost:8080/ingredients/{id}",
                           Ingredient.class, ingredientId);
}
```

Here you're using the `getForObject()` variant that accepts a `String` URL and uses a variable list for URL variables. The `ingredientId` parameter passed into `getForObject()` is used to fill in the `{id}` placeholder in the given URL. Although there's only one URL variable in this example, it's important to know that the variable parameters are assigned to the placeholders in the order that they're given.

The second parameter to `getForObject()` is the type that the response should be bound to. In this case, the response data (that's likely in JSON format) should be deserialized into an `Ingredient` object that will be returned.

Alternatively, you can use a `Map` to specify the URL variables:

```
public Ingredient getIngredientById(String ingredientId) {
  Map<String, String> urlVariables = new HashMap<>();
  urlVariables.put("id", ingredientId);
  return rest.getForObject("http://localhost:8080/ingredients/{id}",
      Ingredient.class, urlVariables);
}
```

In this case, the value of `ingredientId` is mapped to a key of `id`. When the request is made, the `{id}` placeholder is replaced by the map entry whose key is `id`.

Using a `URI` parameter is a bit more involved, requiring that you construct a `URI` object before calling `getForObject()`. Otherwise, it's similar to both of the other variants:

```
public Ingredient getIngredientById(String ingredientId) {
  Map<String, String> urlVariables = new HashMap<>();
  urlVariables.put("id", ingredientId);
  URI url = UriComponentsBuilder
            .fromHttpUrl("http://localhost:8080/ingredients/{id}")
            .build(urlVariables);
  return rest.getForObject(url, Ingredient.class);
}
```

Here the `URI` object is defined from a `String` specification, and its placeholders filled in from

entries in a `Map`, much like the previous variant of `getForObject()`. The `getForObject()` method is a no-nonsense way of fetching a resource. But if the client needs more than the payload body, you may want to consider using `getForEntity()`.

`getForEntity()` works in much the same way as `getForObject()`, but instead of returning a domain object that represents the response's payload, it returns a `ResponseEntity` object that wraps that domain object. The `ResponseEntity` gives access to additional response details, such as the response headers.

For example, suppose that in addition to the ingredient data, you want to inspect the Date header from the response. With `getForEntity()` that becomes straightforward:

```
public Ingredient getIngredientById(String ingredientId) {
  ResponseEntity<Ingredient> responseEntity =
      rest.getForEntity("http://localhost:8080/ingredients/{id}",
          Ingredient.class, ingredientId);
  log.info("Fetched time: " +
          responseEntity.getHeaders().getDate());
  return responseEntity.getBody();
}
```

The `getForEntity()` method is overloaded with the same parameters as `getForObject()`, so you can provide the URL variables as a variable list parameter or call `getForEntity()` with a `URI` object.

8.1.2 PUTting resources

For sending HTTP PUT requests, `RestTemplate` offers the `put()` method. All three overloaded variants of `put()` accept an `Object` that is to be serialized and sent to the given URL. As for the URL itself, it can be specified as a `URI` object or as a `String`. And like `getForObject()` and `getForEntity()`, the URL variables can be provided as either a variable argument list or as a `Map`.

Suppose that you want to replace an ingredient resource with the data from a new `Ingredient` object. The following code should do the trick:

```
public void updateIngredient(Ingredient ingredient) {
  rest.put("http://localhost:8080/ingredients/{id}",
        ingredient, ingredient.getId());
}
```

Here the URL is given as a `String` and has a placeholder that's substituted by the given `Ingredient` object's `id` property. The data to be sent is the `Ingredient` object itself. The `put()` method returns `void`, so there's nothing you need to do to handle a return value.

8.1.3 DELETEing resources

Suppose that Taco Cloud no longer offers an ingredient and wants it completely removed as an option. To make that happen, you can call the `delete()` method from `RestTemplate`:

```
public void deleteIngredient(Ingredient ingredient) {
  rest.delete("http://localhost:8080/ingredients/{id}",
      ingredient.getId());
}
```

In this example, only the URL (specified as a `String`) and a URL variable value are given to `delete()`. But as with the other `RestTemplate` methods, the URL could be specified as a `URI` object or the URL parameters given as a `Map`.

8.1.4 POSTing resource data

Now let's say that you add a new ingredient to the Taco Cloud menu. An HTTP `POST` request to the …/ingredients endpoint with ingredient data in the request body will make that happen. `RestTemplate` has three ways of sending a `POST` request, each of which has the same overloaded variants for specifying the URL. If you wanted to receive the newly created `Ingredient` resource after the `POST` request, you'd use `postForObject()` like this:

```
public Ingredient createIngredient(Ingredient ingredient) {
  return rest.postForObject("http://localhost:8080/ingredients",
      ingredient, Ingredient.class);
}
```

This variant of the `postForObject()` method takes a `String` URL specification, the object to be posted to the server, and the domain type that the response body should be bound to. Although you aren't taking advantage of it in this case, a fourth parameter could be a `Map` of the URL variable value or a variable list of parameters to substitute into the URL.

If your client has more need for the location of the newly created resource, then you can call `postForLocation()` instead:

```
public java.net.URI createIngredient(Ingredient ingredient) {
  return rest.postForLocation("http://localhost:8080/ingredients",
      ingredient);
}
```

Notice that `postForLocation()` works much like `postForObject()` with the exception that it returns a `URI` of the newly created resource instead of the resource object itself. The `URI` returned is derived from the response's `Location` header. In the off chance that you need both the location and response payload, you can call `postForEntity()`:

```
public Ingredient createIngredient(Ingredient ingredient) {
  ResponseEntity<Ingredient> responseEntity =
         rest.postForEntity("http://localhost:8080/ingredients",
                            ingredient,
                            Ingredient.class);
```

```
  log.info("New resource created at " +
           responseEntity.getHeaders().getLocation());
  return responseEntity.getBody();
}
```

Although the methods of `RestTemplate` differ in their purpose, they're quite similar in how they're used. This makes it easy to become proficient with `RestTemplate` and use it in your client code.

On the other hand, if the API you're consuming includes hyperlinks in its response, `RestTemplate` isn't as helpful. It's certainly possible to fetch the more detailed resource data with `RestTemplate` and work with the content and links contained therein, but it's not trivial to do so. Rather than struggle while consuming hypermedia APIs with `RestTemplate`, let's turn our attention to a client library that's made for such things—Traverson.

8.2 Navigating REST APIs with Traverson

Traverson comes with Spring Data HATEOAS as the out-of-the-box solution for consuming hypermedia APIs in Spring applications. This Java-based library is inspired by a similar JavaScript library of the same name (https://github.com/traverson/traverson).

You might have noticed that Traverson's name kind of sounds like "traverse on", which is a good way to describe how it's used. In this section, you'll consume an API by traversing the API on relation names.

Working with Traverson starts with instantiating a `Traverson` object with an API's base URI:

```
Traverson traverson = new Traverson(
    URI.create("http://localhost:8080/api"), MediaTypes.HAL_JSON);
```

Here I've pointed Traverson to the Taco Cloud's base URL (running locally). This is the only URL you'll need to give to Traverson. From here on out, you'll navigate the API by link relation names. You'll also specify that the API will produce JSON responses with HAL-style hyperlinks so that Traverson knows how to parse the incoming resource data. Like `RestTemplate`, you can choose to instantiate a `Traverson` object prior to its use or declare it as a bean to be injected wherever it's needed.

With a `Traverson` object in hand, you can start consuming an API by following links. For example, suppose that you're interested in retrieving a list of all ingredients. You know from section 6.3.1 that the ingredients link has an `href` property that links to the ingredients resource. You'll need to follow that link:

```
ParameterizedTypeReference<CollectionModel<Ingredient>> ingredientType =
    new ParameterizedTypeReference<CollectionModel<Ingredient>>() {};

CollectionModel<Ingredient> ingredientRes =
    traverson
      .follow("ingredients")
```

```
      .toObject(ingredientType);

Collection<Ingredient> ingredients = ingredientRes.getContent();
```

By calling the `follow()` method on the `Traverson` object, you can navigate to the resource whose link's relation name is ingredients. Now that the client has navigated to ingredients, you need to ingest the contents of that resource by calling `toObject()`.

The `toObject()` method requires that you tell it what kind of object to read the data into. This can get a little tricky, considering that you need to read it in as a `CollectionModel<Ingredient>` object, and Java type erasure makes it difficult to provide type information for a generic type. But creating a `ParameterizedTypeReference` helps with that.

As an analogy, imagine that instead of a REST API, this were a homepage on a website. And instead of REST client code, imagine that it's you viewing that homepage in a browser. You see a link on the page that says Ingredients and you follow that link by clicking it. Upon arriving at the next page, you read the page, which is analogous to Traverson ingesting the content as a `CollectionModel<Ingredient>` object.

Now let's consider a slightly more interesting use case. Let's say that you want to fetch the most recently created tacos. Starting at the home resource, you can navigate to the recent tacos resource like this:

```
ParameterizedTypeReference<CollectionModel<Taco>> tacoType =
    new ParameterizedTypeReference<CollectionModel<Taco>>() {};

CollectionModel<Taco> tacoRes =
    traverson
      .follow("tacos")
      .follow("recents")
      .toObject(tacoType);

Collection<Taco> tacos = tacoRes.getContent();
```

Here you follow the Tacos link and then, from there, follow the Recents link. That brings you to the resource you're interested in, so a call to `toObject()` with an appropriate `ParameterizedTypeReference` gets you what you want. The `.follow()` method can be simplified by listing a trail of relation names to follow:

```
CollectionModel<Taco> tacoRes =
   traverson
     .follow("tacos", "recents")
     .toObject(tacoType);
```

As you can see, Traverson makes easy work of navigating a HATEOAS-enabled API and consuming its resources. But one thing it doesn't do is offer any methods for writing to or deleting from those APIs. In contrast, `RestTemplate` can write and delete resources, but doesn't make it easy to navigate an API.

When you need to both navigate an API and update or delete resources, you'll need to use `RestTemplate` and Traverson together. Traverson can still be used to navigate to the link where a new resource will be created. Then `RestTemplate` can be given that link to do a `POST`, `PUT`, `DELETE`, or any other HTTP request you need.

For example, suppose you want to add a new `Ingredient` to the Taco Cloud menu. The following `addIngredient()` method teams up Traverson and `RestTemplate` to post a new `Ingredient` to the API:

```
public Ingredient addIngredient(Ingredient ingredient) {
  String ingredientsUrl = traverson
      .follow("ingredients")
      .asLink()
      .getHref();

  return rest.postForObject(ingredientsUrl,
                            ingredient,
                            Ingredient.class);
}
```

After following the Ingredients link, you ask for the link itself by calling `asLink()`. From that link, you ask for the link's URL by calling `getHref()`. With a URL in hand, you have everything you need to call `postForObject()` on the `RestTemplate` instance and save the new ingredient.

8.3 Summary

- Clients can use RestTemplate to make HTTP requests against REST APIs.
- Traverson enables clients to navigate an API using hyperlinks embedded in the responses.

Notes

1. For a much more in-depth discussion of application domains, I suggest Eric Evans' *Domain-Driven Design* (Addison-Wesley Professional, 2003).

2. The contents of the stylesheet aren't relevant to our discussion; it only contains styling to present the ingredients in two columns instead of one long list of ingredients.

3. One such exception is Thymeleaf's Spring Security dialect, which we'll talk about in chapter 4.

4. https://en.wikipedia.org/wiki/NoSQL

5. https://docs.datastax.com/en/developer/java-driver/latest/manual/core/load_balancing/

6. https://cassandra.apache.org/doc/latest/data_modeling/intro.html

7. I chose to use Angular, but the choice of frontend framework should have little to no bearing on how the backend Spring code is written. Feel free to choose Angular, React, Vue.js, or whatever frontend technology suits you best.

8. Mapping HTTP methods to create, read, update, and delete (CRUD) operations isn't a perfect match, but in practice, that's how they're often used and how you'll use them in Taco Cloud.

Index Terms

writing Spring applications (Spring Boot DevTools use) [built in H2 console]
writing Spring applications (Spring Boot DevTools use) [built in H2 console]
writing Spring applications (Spring Boot DevTools use)
writing Spring applications