Part 1

Foundational Spring

Part 1 of this book will get you started writing a Spring application, learning the foundations of Spring along the way.

In chapter 1, I'll give you a quick overview of Spring and Spring Boot essentials and show you how to initialize a Spring project as you work on building Taco Cloud, your first Spring application. In chapter 2, you'll dig deeper into the Spring MCV and learn how to present model data in the browser and how to process and validate form input. You'll also get some tips on choosing a view template library. You'll add data persistence to the Taco Cloud application in chapter 3. There, we'll cover using Spring's JDBC template, how to insert data, and how to declare JPA repositories with Spring Data. Chapter 4 covers security for your Spring application, including autoconfiguring Spring Security, defining custom user storage, customizing the login page, and securing against cross-site request forgery (CSRF) attacks. To close out part 1, we'll look at configuration properties in chapter 5. You'll learn how to fine-tune autoconfigured beans, apply configuration properties to application components, and work with Spring profiles.

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 15 years ago, when 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.

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

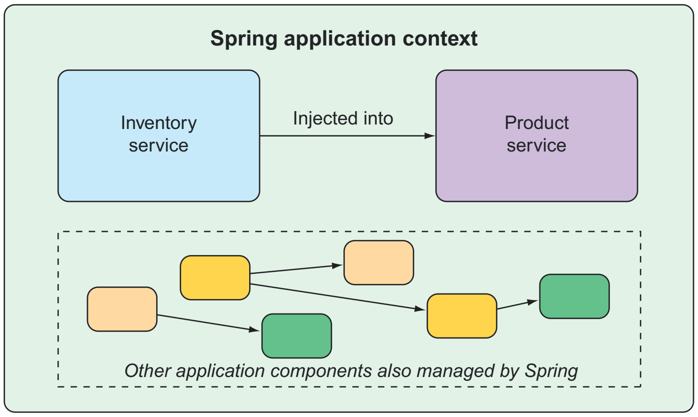

Spring application context
Inventory service
Injected into
Product service
Other application components also managed by Spring

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

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 (at least at the time I write this) isn't available in any of the other IDE options.

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.

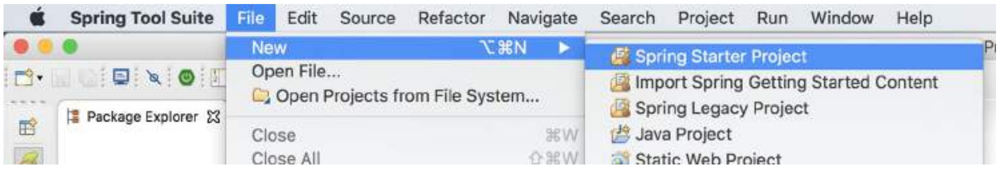
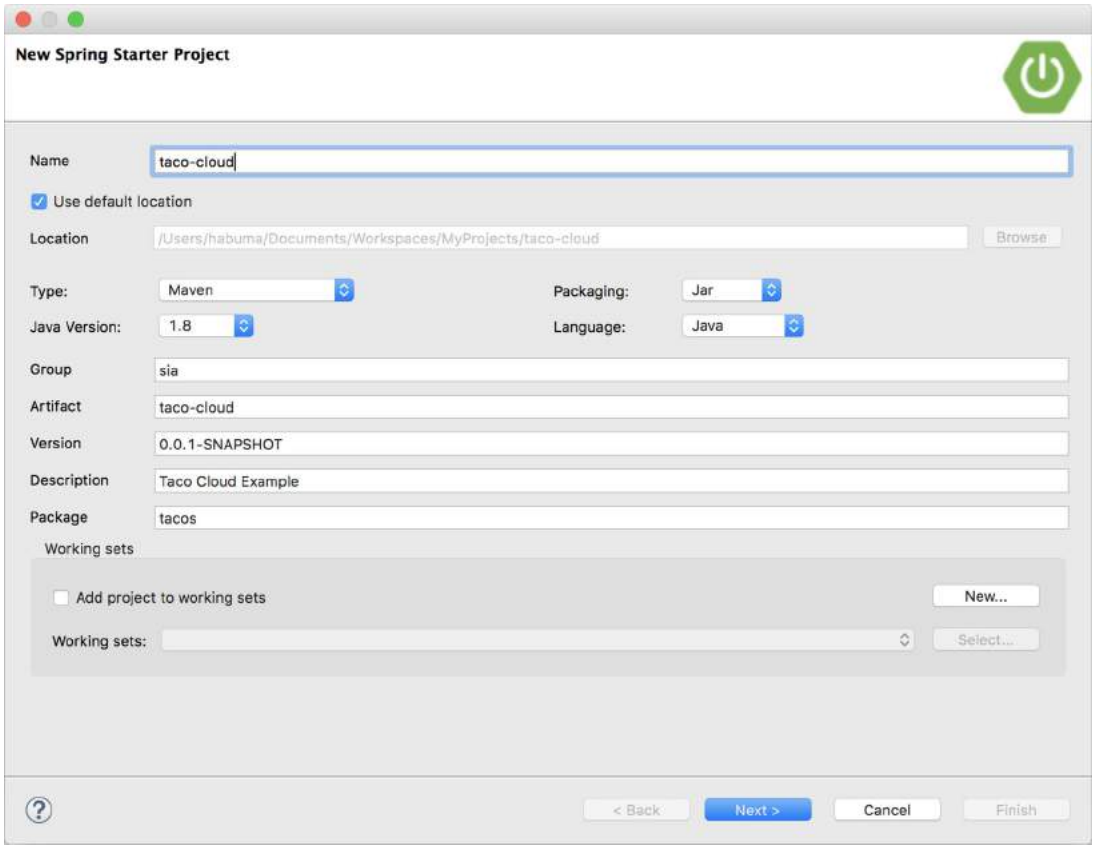


Figure 1.2 Starting a new project with the Initializr in Spring Tool Suite

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.


New Spring Starter Project

Name: taco-cloud

☑ Use default location

Location: /Users/habuma/Documents/Workspaces/MyProjects/taco-cloud

Type: Maven
Packaging: Jar
Java Version: 1.8
Language: Java

Group: sia
Artifact: taco-cloud
Version: 0.0.1-SNAPSHOT
Description: Taco Cloud Example
Package: tacos

Working sets

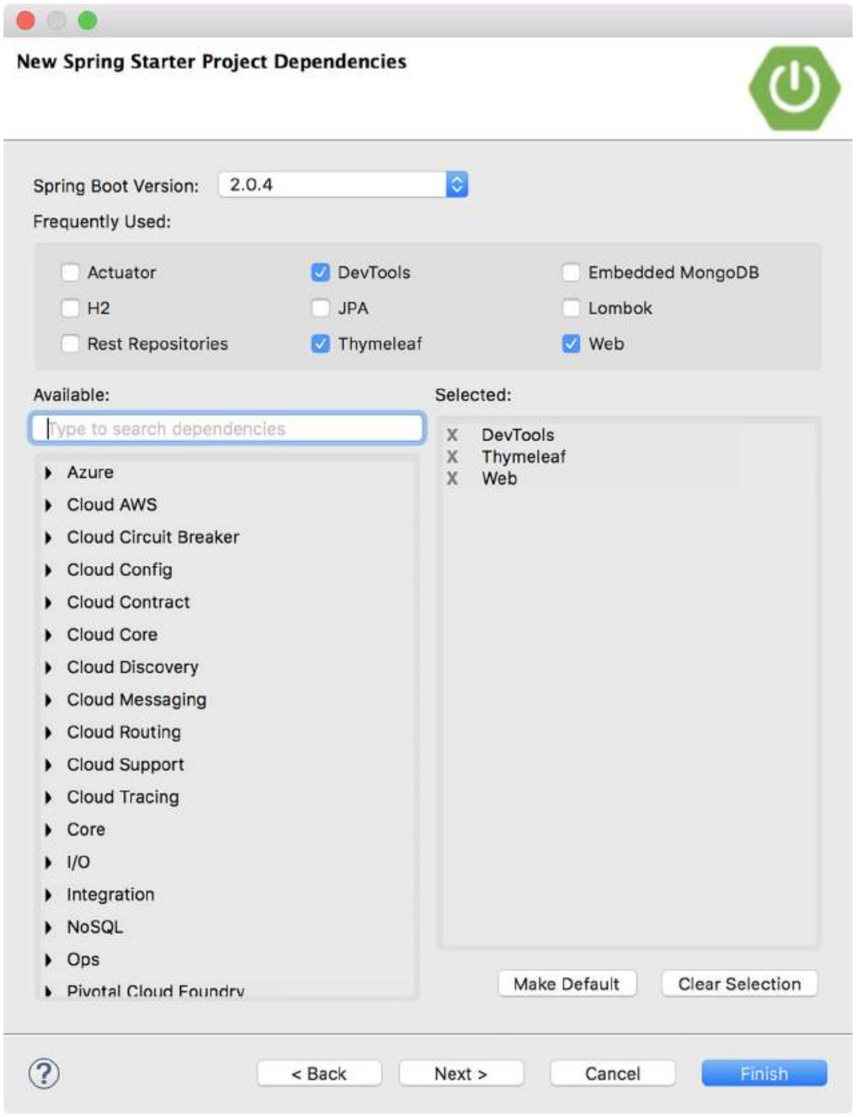
☐ Add project to working sets

Working sets:

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.


New Spring Starter Project Dependencies

Spring Boot Version: 2.0.4

Frequently Used:

☐ Actuator	✓ DevTools	☐ Embedded MongoDB
☐ H2	☐ JPA	☐ Lombok
☐ Rest Repositories	✓ Thymeleaf	✓ Web

Available:

Type to search dependencies

- Azure
- Cloud AWS
- Cloud Circuit Breaker
- Cloud Config
- Cloud Contract
- Cloud Core
- Cloud Discovery
- Cloud Messaging
- Cloud Routing
- Cloud Support
- Cloud Tracing
- Core
- I/O
- Integration
- NoSQL
- Ops
- Pivotal Cloud Foundry

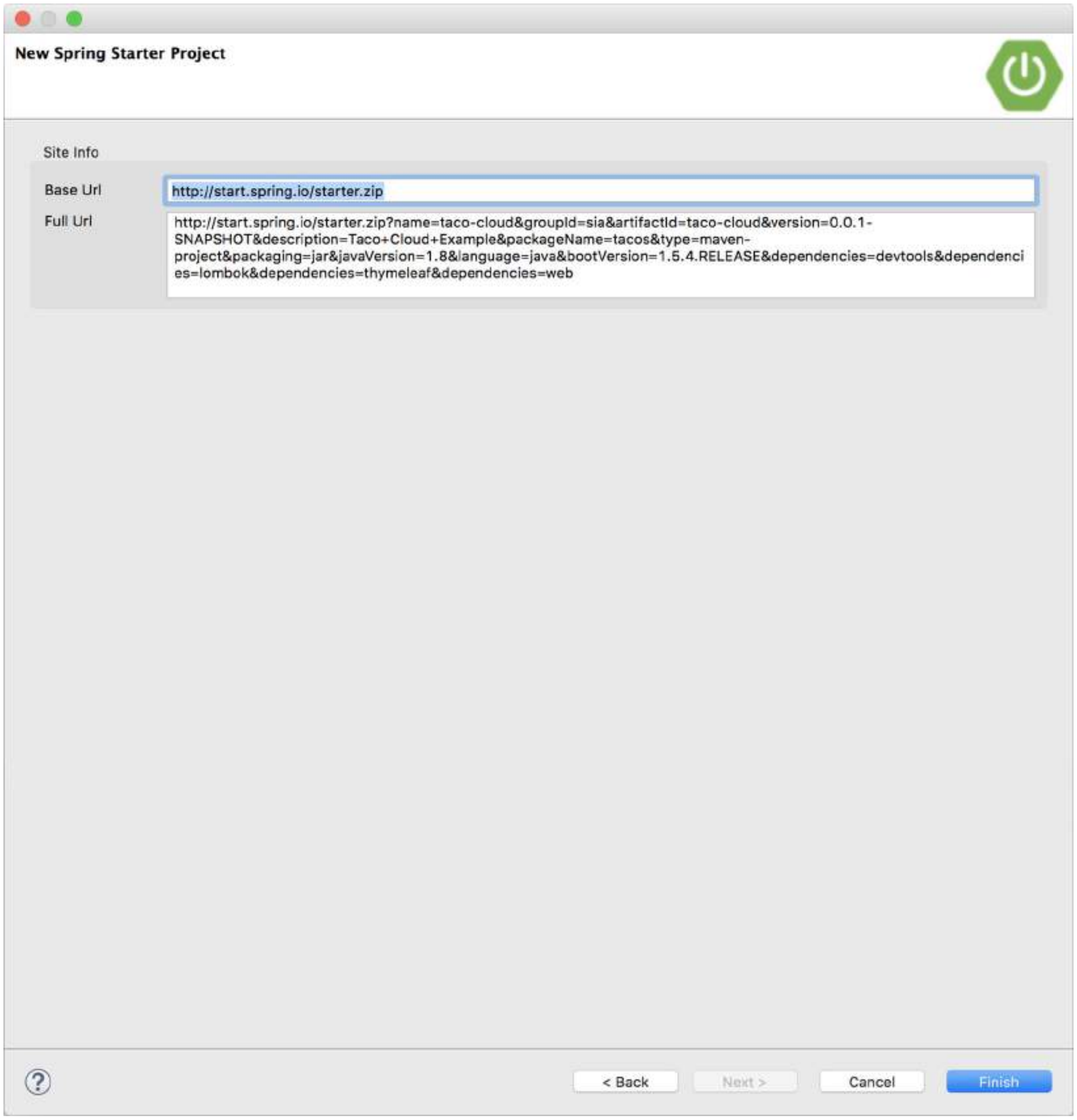
Selected:

X DevTools
X Thymeleaf
X Web

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.

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



Base Url: http://start.spring.io/starter.zip

Full Url: http://start.spring.io/starter.zip?name=taco-cloud&groupId=sia&artifactId=taco-cloud&version=0.0.1-SNAPSHOT&description=Taco+Cloud+Example&packageName=tacos&type=maven-project&packaging=jar&javaVersion=1.8&language=java&bootVersion=1.5.4.RELEASE&dependencies=devtools&dependencies=lombok&dependencies=thymeleaf&dependencies=web

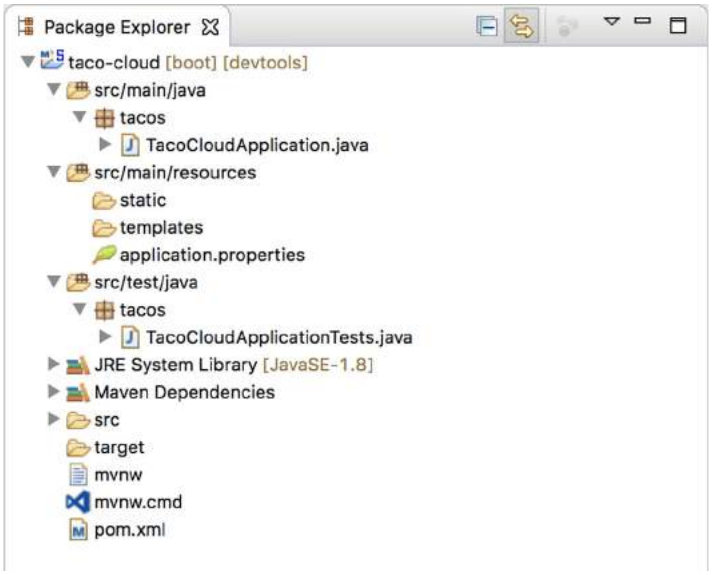
Figure 1.5 Optionally specifying an alternate Initializr address

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.



```
Package Explorer
▼ taco-cloud [boot] [devtools]
  ▼ src/main/java
    ▼ tacos
      ▶ TacoCloudApplication.java
  ▼ src/main/resources
      static
      templates
      application.properties
  ▼ src/test/java
    ▼ tacos
      ▶ TacoCloudApplicationTests.java
  ▶ JRE System Library [JavaSE-1.8]
  ▶ Maven Dependencies
  ▶ src
    target
    mvnw
    mvnw.cmd
    pom.xml
```

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
          http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>sia</groupId>
  <artifactId>taco-cloud</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>                <—— JAR packaging

  <name>taco-cloud</name>
  <description>Taco Cloud Example</description>

  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.0.4.RELEASE</version>                <—— Spring Boot version
    <relativePath/> <!-- lookup parent from repository -->
  </parent>

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

  <dependencies>                                                    Starter
    <dependency>                                                    dependencies
      <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>
    </dependency>

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

    <dependency>
      <groupId>org.seleniumhq.selenium</groupId>
      <artifactId>selenium-java</artifactId>
        <scope>test</scope>
    </dependency>

     <dependency>
        <groupId>org.seleniumhq.selenium</groupId>
        <artifactId>htmlunit-driver</artifactId>
        <scope>test</scope>
      </dependency>
  </dependencies>

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

</project>
```

The first noteworthy item in the pom.xml file is the `<packaging>` element. You chose to build your application as an executable JAR file, as opposed to a WAR file. This is probably one of the most curious choices you'll make, especially for a web application. After all, traditional Java web applications are packaged as WAR files, leaving JAR files the packaging of choice for libraries and the occasional desktop UI application.

The choice of JAR packaging is a cloud-minded choice. Whereas WAR files are perfectly suitable for deploying to a traditional Java application server, they're not a natural fit for most cloud platforms. Although some cloud platforms (such as Cloud Foundry) are capable of deploying and running WAR files, all Java cloud platforms are capable of running an executable JAR file. Therefore, the Spring Initializr defaults to JAR packaging unless you tell it to do otherwise.

If you intend to deploy your application to a traditional Java application server, then you'll need to choose WAR packaging and include a web initializer class. We'll look at how to build WAR files in more detail in chapter 2.

Next, take note of 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, `2.0.4.RELEASE`, indicates that you're using Spring Boot 2.0.4 and, thus, will inherit dependency management as defined by that version of Spring Boot.

While we're on the subject of dependencies, note that there are three dependencies declared under the `<dependencies>` element. The first two should look somewhat familiar to you. They correspond directly to the `Web` and `Thymeleaf` dependencies that you selected before clicking the Finish button in the Spring Tool Suite new project wizard. The third 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 three dependencies 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 worry 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                                  <-- Spring Boot
public class TacoCloudApplication {                         application

  public static void main(String[] args) {
    SpringApplication.run(TacoCloudApplication.class, args);   <-- 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 application 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.Test;
import org.junit.runner.RunWith;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)                                    Uses the
@SpringBootTest                                                 Spring runner
public class TacoCloudApplicationTests {
                                                          A Spring
                                                          Boot test
  @Test
  public void contextLoads() {                  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.

Also notice the class annotated with `@RunWith(SpringRunner.class)`. `@RunWith` is a JUnit annotation, providing a test runner that guides JUnit in running a test. Think of it as applying a plugin to JUnit to provide custom testing behavior. In this case, JUnit is given `SpringRunner`, a Spring-provided test runner that provides for the creation of a Spring application context that the test will run against.

A test runner by any other name…

If you're already familiar with writing Spring tests or are maybe looking at some existing Spring-based test classes, you may have seen a test runner named `SpringJUnit4ClassRunner`. `SpringRunner` is an alias for `SpringJUnit4ClassRunner`, and was introduced in Spring 4.3 to remove the association with a specific version of JUnit (for example, JUnit 4). And there's no denying that the alias is easier to read and type.

`@SpringBootTest` tells JUnit to bootstrap the test with Spring Boot capabilities. For now, 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 `@RunWith(SpringRunner.class)` and `@SpringBootTest` are tasked to load the Spring application context for the test, they 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                                  The controller
public class HomeController {

  @GetMapping("/")                           Handles requests
  public String home() {                     for the root path /
    return "home";                           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.

WHY THYMELEAF?

You may be wondering why you 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.

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

@RunWith(SpringRunner.class)
@WebMvcTest(HomeController.class)                Web test for
public class HomeControllerTest {                HomeController

  @Autowired
  private MockMvc mockMvc;                  Injects MockMvc

  @Test
  public void testHomePage() throws Exception {
    mockMvc.perform(get("/"))                           Performs GET /

      .andExpect(status().isOk())             Expects HTTP 200

      .andExpect(view().name("home"))              Expects home view

      .andExpect(content().string(                      Expects Welcome to...
          containsString("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.

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.

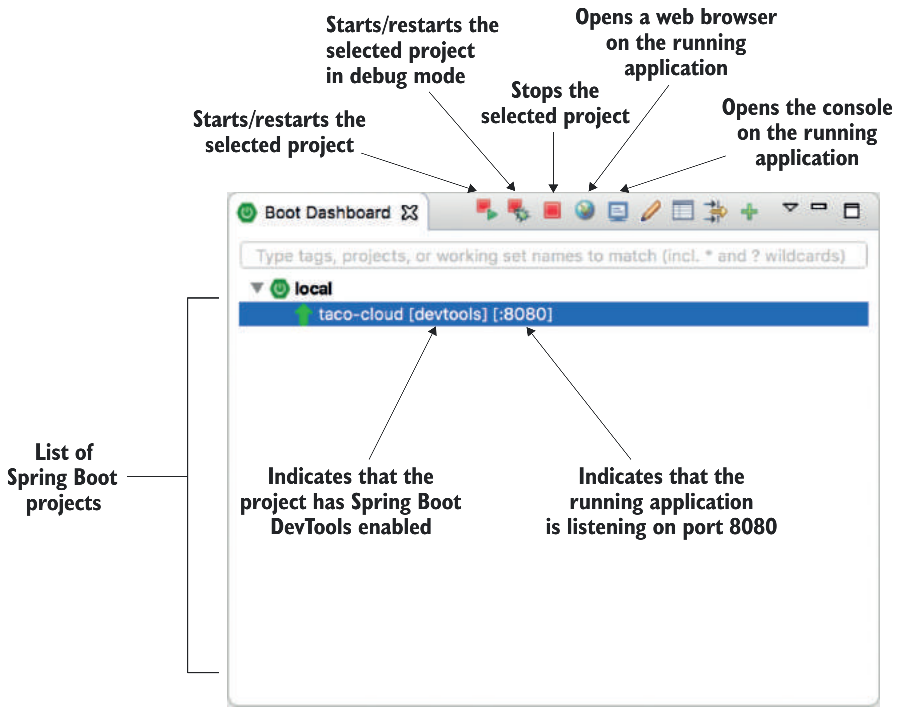

Starts/restarts the selected project in debug mode
Opens a web browser on the running application
Starts/restarts the selected project
Stops the selected project
Opens the console on the running application
Boot Dashboard
Type tags, projects, or working set names to match (incl. * and ? wildcards)
local
taco-cloud [devtools] [:8080]
List of Spring Boot projects
Indicates that the project has Spring Boot DevTools enabled
Indicates that the running application is listening on port 8080

Figure 1.7 Highlights of the Spring Boot Dashboard

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.

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

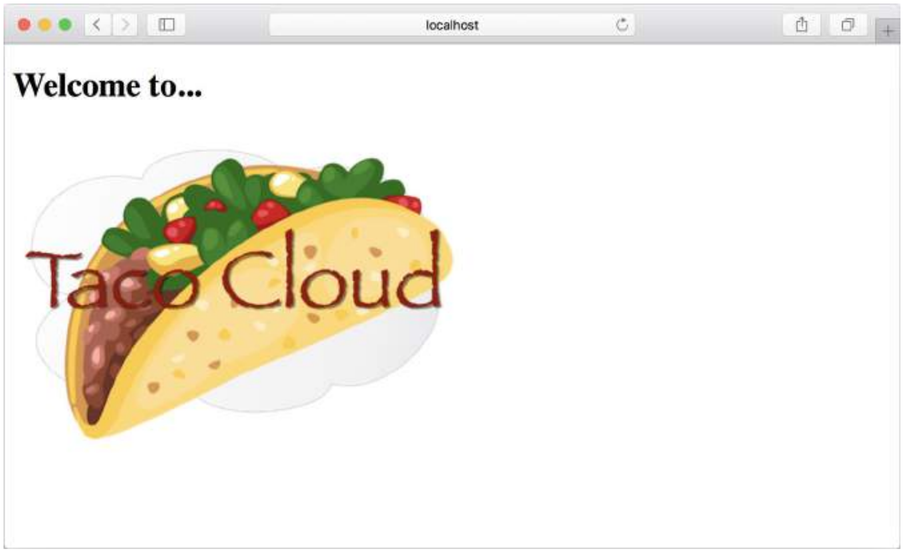


Figure 1.8 The Taco Cloud homepage

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.

In the most recent version of Spring (5.0.8), support was added 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 a several different kinds of databases, including relational (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 (Manning, 2017, www.manning.com/books/spring-microservices-in-action).

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.

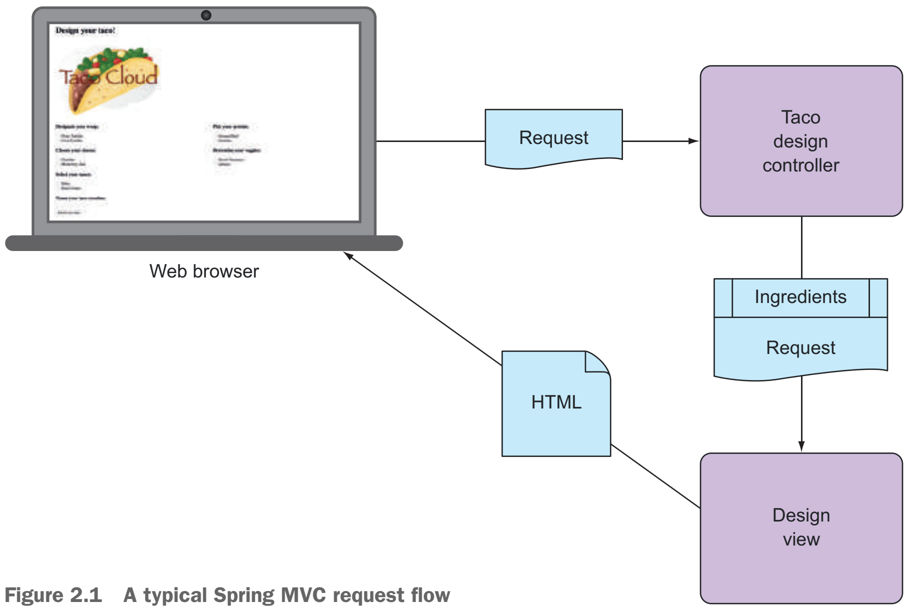


Figure 2.1 A typical Spring MVC request flow

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;
import lombok.RequiredArgsConstructor;

@Data
@RequiredArgsConstructor
public class Ingredient {

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

  public static 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 runtime. In fact, the `@Data` annotation at the class level is provided by Lombok and tells

[1] For a much more in-depth discussion of application domains, I suggest Eric Evans' *Domain-Driven Design* (Addison-Wesley Professional, 2003).

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>
  <optional>true</optional>
</dependency>
```

This dependency will provide you with Lombok annotations (such as `@Data`) at development time and with automatic method generation at runtime. 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.

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. When you finish, you'll add some controllers to handle web requests in your 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 2.2 The beginnings of a Spring controller class

```
package tacos.web;

import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

import javax.validation.Valid;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
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 lombok.extern.slf4j.Slf4j;
import tacos.Taco;
import tacos.Ingredient;
import tacos.Ingredient.Type;

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

  @GetMapping
  public String showDesignForm(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));
    }

    model.addAttribute("design", new Taco());

    return "design";
  }

}
```

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 runtime, will automatically generate an SLF4J (Simple Logging Facade for Java, https://www.slf4j.org/) `Logger` 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 a relatively new annotation, having been introduced in Spring 4.3. Prior to Spring 4.3, you might have used a method-level `@RequestMapping` annotation instead:

```
@RequestMapping(method=RequestMethod.GET)
```

Clearly, `@GetMapping` is more succinct and specific to the HTTP method that it targets. `@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

Making the right thing the easy thing

It's always a good idea to be as specific as possible when declaring request mappings on your controller methods. At the very least, this means declaring both a path (or inheriting a path from the class-level `@RequestMapping`) and which HTTP method it will handle.

The lengthier `@RequestMapping(method=RequestMethod.GET)` made it tempting to take the lazy way out and leave off the `method` attribute. Thanks to Spring 4.3's new mapping annotations, the right thing to do is also the easy thing to do—with less typing.

The new request-mapping annotations have all of the same attributes as `@RequestMapping`, so you can use them anywhere you'd otherwise use `@RequestMapping`.

Generally, I prefer to only use `@RequestMapping` at the class level to specify the base path. I use the more specific `@GetMapping`, `@PostMapping`, and so on, on each of the handler methods.

Now that you know that the `showDesignForm()` method will handle the request, let's look at the method body to see how it ticks. The bulk of the method constructs a list of `Ingredient` objects. 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 `showDesignForm()` filters the list by ingredient type. A list of ingredient types is then added as an attribute to the `Model` object that's 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 response attributes, where the view can find them. The `showDesignForm()` method concludes by returning `"design"`, which is the logical name of the view that will be used to render the model to the 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()` would be engaged, fetching data from the repository and placing it in 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 404 (Not Found) 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.

In order to use Thymeleaf, you need to add another dependency to your project build. The following `<dependency>` entry uses Spring Boot's Thymeleaf starter to make Thymeleaf available for rendering the view you're about to create:

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

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 : ${wrap}">
  <input name="ingredients" type="checkbox" th:value="${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 `<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 2.3 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="${design}">
    <div class="grid">
      <div class="ingredient-group" id="wraps">
      <h3>Designate your wrap:</h3>
      <div th:each="ingredient : ${wrap}">
        <input name="ingredients" type="checkbox" th:value="${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 : ${protein}">
        <input name="ingredients" type="checkbox" th:value="${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 name="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 : ${veggies}">
          <input name="ingredients" type="checkbox" th:value="${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 : ${sauce}">
          <input name="ingredients" type="checkbox" th:value="${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.

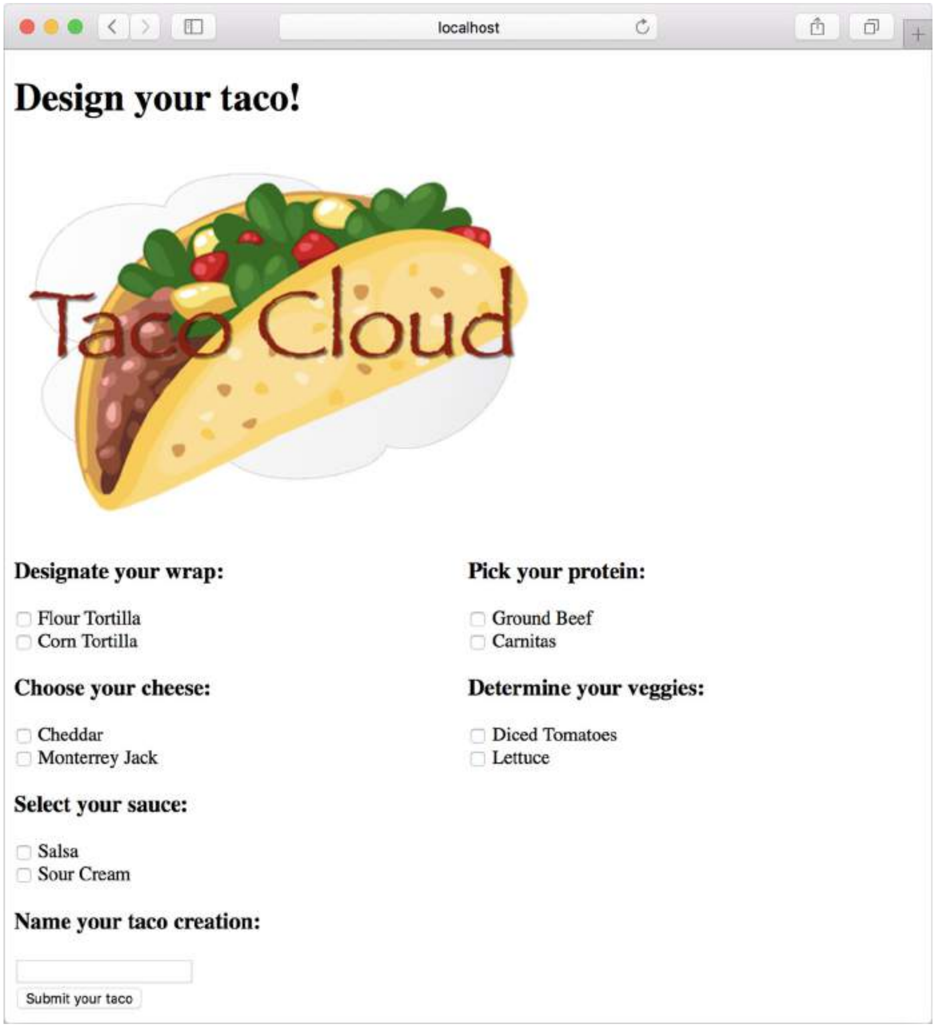
It's also worth noting that the complete template includes the Taco Cloud logo image and a `<link>` reference to a stylesheet.[2] In both cases, Thymeleaf's `@{}` operator is used to produce a context-relative path to the static artifacts that they're referencing. As you learned in chapter 1, static 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 first building it into an executable

[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.

JAR file and then running the JAR with `java -jar`. I also showed how you can run the application directly from the build with `mvn spring-boot:run`.

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.



Design your taco!

Designate your wrap:

- Flour Tortilla
- Corn Tortilla

Pick your protein:

- Ground Beef
- Carnitas

Choose your cheese:

- Cheddar
- Monterrey Jack

Determine your veggies:

- Diced Tomatoes
- Lettuce

Select your sauce:

- Salsa
- Sour Cream

Name your taco creation:

Submit your taco

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 2.2, 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 processDesign() method in the following listing to DesignTacoController.

Listing 2.4 Handling POST requests with @PostMapping

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

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

As applied to the processDesign() method, @PostMapping coordinates with the class-level @RequestMapping to indicate that processDesign() 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 processDesign(). From there, the processDesign() method can do whatever it wants with the Taco object.

Listing 2.5 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<String> 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 automatically generate essential JavaBean methods for you at runtime.

If you look back at the form in listing 2.3, 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<String> that will capture each of the chosen ingredients.

For now, the processDesign() 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, processDesign() 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 processDesign() is prefixed with "redirect:", indicating that this is a redirect view. More specifically, it indicates that after processDesign() completes, the user's browser should be redirected to the relative path /order/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 @GetMapping, you can easily create such a controller. It might look something like the following listing.

Listing 2.6 A controller to present a taco order form

```
package tacos.web;
import javax.validation.Valid;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.Errors;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import lombok.extern.slf4j.Slf4j;
import tacos.Order;
```

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

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

}
```

Once again, you use Lombok's `@Slf4j` annotation to create a free SLF4J `Logger` object at runtime. 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 2.7 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="${order}">
      <h1>Order your taco creations!</h1>

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

      <div th:if="${#fields.hasErrors()}">
        <span class="validationError">
        Please correct the problems below and resubmit.
        </span>
      </div>
      <h3>Deliver my taco masterpieces to...</h3>
      <label for="name">Name: </label>
      <input type="text" th:field="*{name}"/>
      <br/>

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

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

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

      <label for="zip">Zip code: </label>
      <input type="text" th:field="*{zip}"/>
      <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 2.3 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 2.8 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 `Order` object whose properties are bound to the submitted form fields. `Order`, much like `Taco`, is a fairly straightforward class that carries order information.

Listing 2.9 A domain object for taco orders

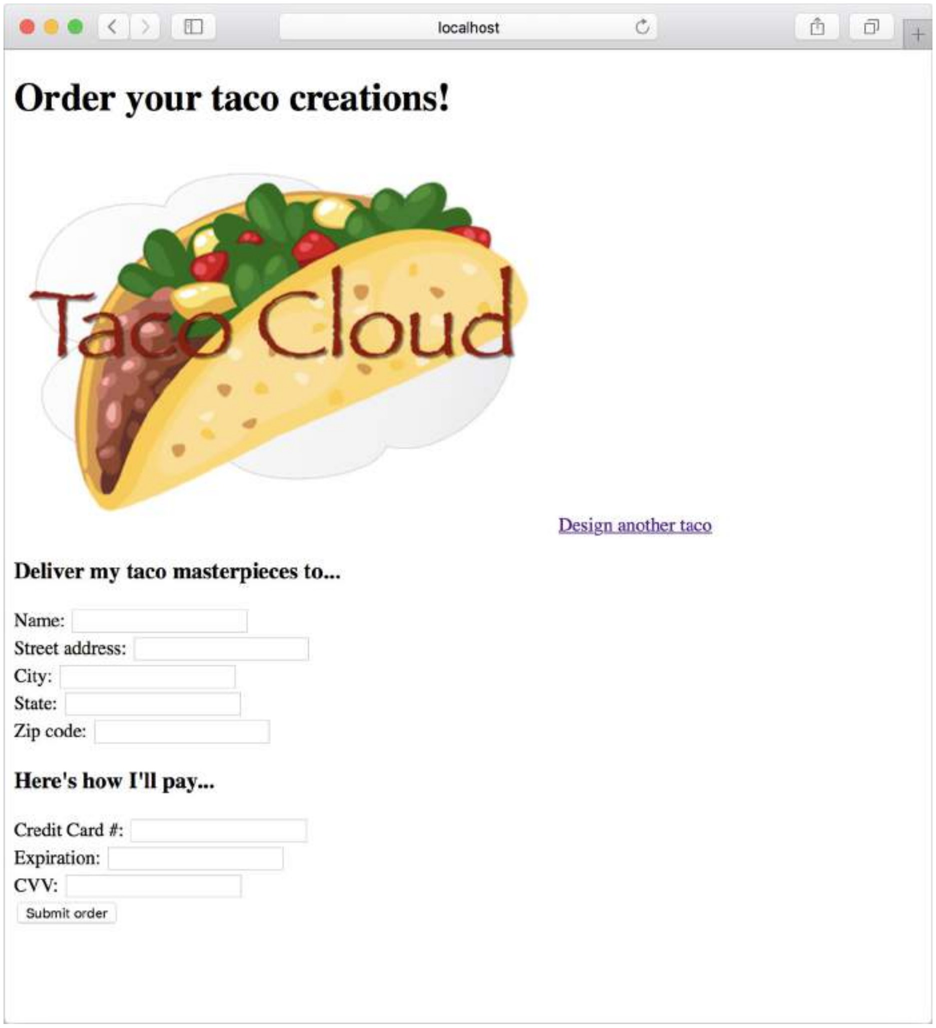
```
package tacos;
import javax.validation.constraints.Digits;
import javax.validation.constraints.Pattern;
import org.hibernate.validator.constraints.CreditCardNumber;
import org.hibernate.validator.constraints.NotBlank;
import lombok.Data;

@Data
public class Order {

  private String name;
  private String street;
  private String city;
  private String state;
  private String zip;
  private String ccNumber;
  private String ccExpiration;
  private String ccCVV;

}
```

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.

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: Order(name=Craig Walls,street1=1234 7th Street,
     city=Somewhere, state=Who knows?, zip=zipzap, ccNumber=Who can guess?,
ccExpiration=Some day, ccCVV=See-vee-vee)
```

If you look carefully at the log entry from my test order, you can see that although the `processOrder()` method did its job and handled the form submission, it let a little bit of bad information get in. Most of the fields in the form contained data that couldn't



Order your taco creations!

Design another taco

Deliver my taco masterpieces to...

Name:
Street address:
City:
State:
Zip code:

Here's how I'll pay...

Credit Card #:
Expiration:
CVV:

Figure 2.3 The taco order form

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 `processDesign()` 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. And with Spring Boot, you don't need to do anything special to add validation libraries to your project, because the Validation API and the Hibernate implementation of the Validation API are automatically added to the project as transient dependencies of Spring Boot's web starter.

To apply validation in Spring MVC, you need to

- 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 `processDesign()` 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. Let's see how you can apply a few of these annotations to validate a submitted `Taco` or `Order`.

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 2.10 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;
  @Size(min=1, message="You must choose at least 1 ingredient")
  private List<String> 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 `Order` 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 Hibernate Validator's `@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 `Order` class.

Listing 2.11 Validating order fields

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

@Data
public class Order {

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

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

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

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

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

  @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;

}
```

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 `Order` 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 `processDesign()` method.

Listing 2.12 Validating a POSTed `Taco`

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

  // Save the taco design...
  // We'll do this in chapter 3
  log.info("Processing design: " + design);
  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 `processDesign()` 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 `processDesign()`. The first few lines of `processDesign()` 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 `Order` objects, similar changes are also required in the `processOrder()` method of `OrderController`.

Listing 2.13 Validating a POSTed `Order`

```
@PostMapping
public String processOrder(@Valid Order 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.

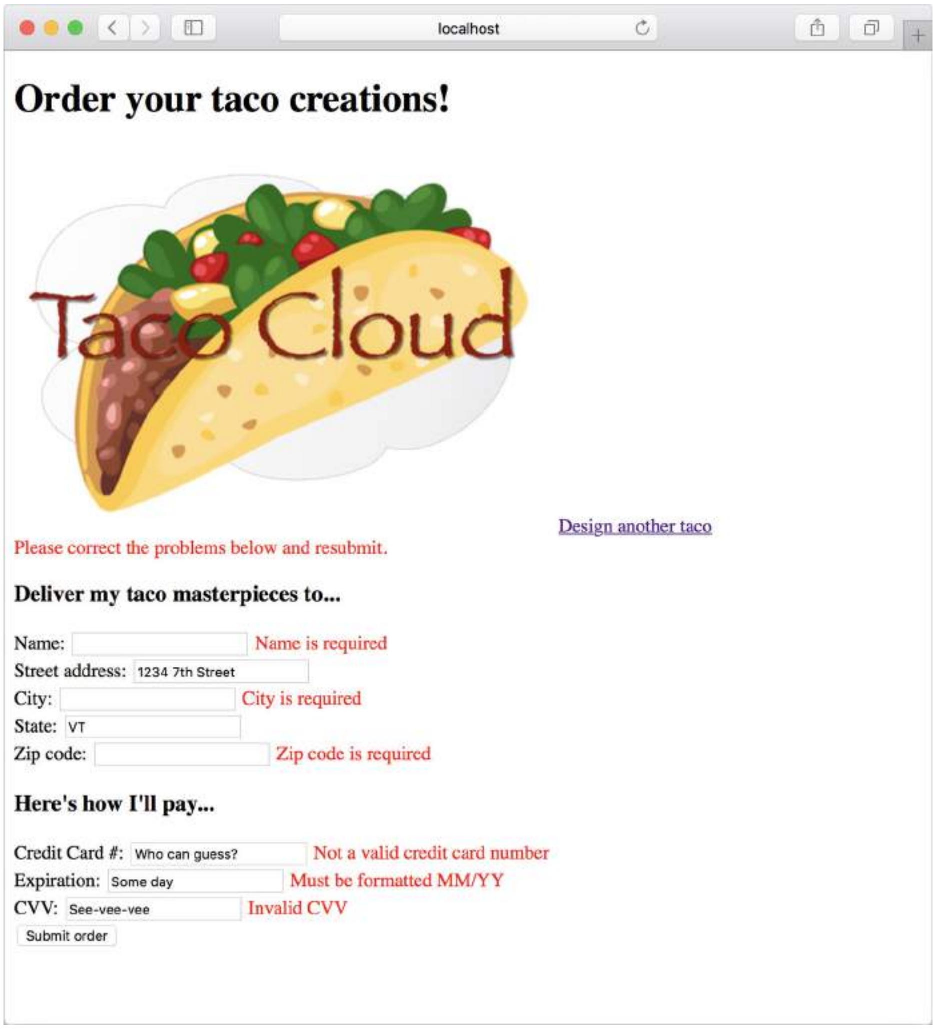
Listing 2.14 Displaying validation errors

```
<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.



Order your taco creations!

Please correct the problems below and resubmit.

Deliver my taco masterpieces to...

Name: Name is required
Street address: 1234 7th Street
City: City is required
State: VT
Zip code: Zip code is required

Here's how I'll pay...

Credit Card #: Who can guess? Not a valid credit card number
Expiration: Some day Must be formatted MM/YY
CVV: See-vee-vee Invalid CVV

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 2.15 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 `Web-MvcConfigurer` 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>
```

[3] One such exception is Thymeleaf's Spring Security dialect, which we'll talk about in chapter 4.

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.

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`
- Inserting data with `SimpleJdbcInsert`
- 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 Ingredient findOne(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 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 methods that use `JdbcTemplate`.

Listing 3.2 Querying a database with `JdbcTemplate`

```
private JdbcTemplate jdbc;

@Override
public Ingredient findOne(String id) {
  return jdbc.queryForObject(
      "select id, name, type from Ingredient where id=?",
      this::mapRowToIngredient, id);
}

private Ingredient mapRowToIngredient(ResultSet rs, int rowNum)
    throws SQLException {
  return new Ingredient(
      rs.getString("id"),
      rs.getString("name"),
      Ingredient.Type.valueOf(rs.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 `queryForObject()` method) and mapping the results to an `Ingredient` object (in 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 `Order`.

Moreover, it might be useful to know when a `Taco` is created and when an `Order` 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;

    ...

}
```

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 `Order` class, as shown here:

```
@Data
public class Order {

  private Long id;

  private Date placedAt;

  ...

}
```

Again, Lombok automatically generates the accessor methods, so these are the only changes required in `Order`. (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 `Jdbc-Template` 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>
```

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 tacos.Ingredient;

public interface IngredientRepository {

  Iterable<Ingredient> findAll();

  Ingredient findOne(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 org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.stereotype.Repository;

import tacos.Ingredient;

@Repository
public class JdbcIngredientRepository
    implements IngredientRepository {

  private JdbcTemplate jdbc;


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

  ...

}
```

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` via the `@Autowired` annotated construction. 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 jdbc.query("select id, name, type from Ingredient",
      this::mapRowToIngredient);
}

@Override
public Ingredient findOne(String id) {
  return jdbc.queryForObject(
      "select id, name, type from Ingredient where id=?",
      this::mapRowToIngredient, id);
}
```

```
private Ingredient mapRowToIngredient(ResultSet rs, int rowNum)
    throws SQLException {
  return new Ingredient(
      rs.getString("id"),
      rs.getString("name"),
      Ingredient.Type.valueOf(rs.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. `findAll()` 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.

The `findById()` method only expects to return a single `Ingredient` object, so it uses the `queryForObject()` method of `JdbcTemplate` instead of `query()`. `queryForObject()` works much like `query()` except that it returns a single object instead of a `List` of objects. In this case, it's given the query to perform, a `RowMapper`, and the `id` of `Ingredient` to fetch, which is used in place of the `?` in the query.

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 8'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 `findAll()` shows how to do that:

```
@Override
public Ingredient findOne(String id) {
  return jdbc.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) {
  jdbc.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()` or `queryForObject()`. 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("order")
public class DesignTacoController {

  private final IngredientRepository ingredientRepo;

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

  @GetMapping
  public String showDesignForm(Model model) {
    List<Ingredient> ingredients = new ArrayList<>();
    ingredientRepo.findAll().forEach(i -> ingredients.add(i));

    Type[] types = Ingredient.Type.values();
    for (Type type : types) {
      model.addAttribute(type.toString().toLowerCase(),
          filterByType(ingredients, type));
    }

    return "design";
  }

  ...

}
```

Notice that the second line of the `showDesignForm()` method now makes a call to the injected `IngredientRepository`'s `findAll()` method. The `findAll()` method fetches all the ingredients from the database before filtering them into distinct types in the model.

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're 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.

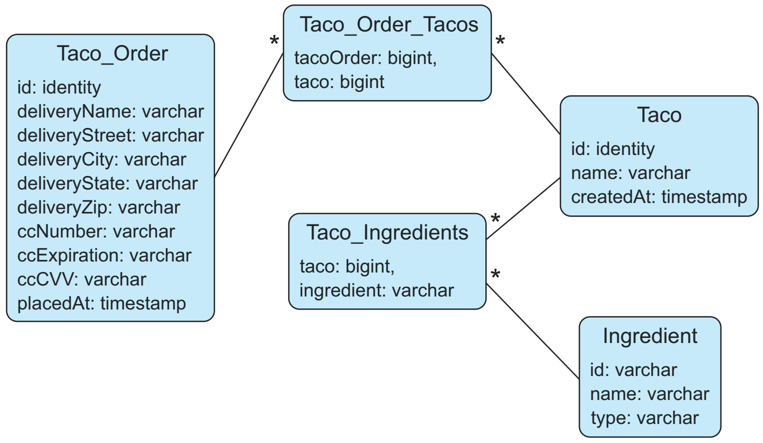


Figure 3.1 The tables for the Taco Cloud schema

The tables in figure 3.1 serve the following purposes:

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

The next listing shows the SQL that creates the tables.

Listing 3.8 Defining the Taco Cloud schema

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

create table if not exists Taco (
  id identity,
  name varchar(50) not null,
  createdAt timestamp not null
);

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

alter table Taco_Ingredients
    add foreign key (taco) references Taco(id);
alter table Taco_Ingredients
    add foreign key (ingredient) references Ingredient(id);

create table if not exists Taco_Order (
  id identity,
    deliveryName varchar(50) not null,
    deliveryStreet varchar(50) not null,
    deliveryCity varchar(50) not null,
    deliveryState varchar(2) not null,
    deliveryZip varchar(10) not null,
    ccNumber varchar(16) not null,
    ccExpiration varchar(5) not null,
    ccCVV varchar(3) not null,
    placedAt timestamp not null
);

create table if not exists Taco_Order_Tacos (
  tacoOrder bigint not null,
  taco bigint not null
);

alter table Taco_Order_Tacos
    add foreign key (tacoOrder) references Taco_Order(id);
alter table Taco_Order_Tacos
    add foreign key (taco) references Taco(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 3.9 Preloading the database

```
delete from Taco_Order_Tacos;
delete from Taco_Ingredients;
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`, `Order`, and 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. Two ways to save data with `JdbcTemplate` include the following:

- Directly, using the `update()` method
- Using the `SimpleJdbcInsert` wrapper class

Let's first see how to use the `update()` method when the persistence needs are more complex than what was required to save an `Ingredient`.

Saving data with JdbcTemplate

For now, the only thing that the taco and order repositories need to do is to save their respective objects. To save `Taco` objects, the `TacoRepository` declares a `save()` method like this:

```
package tacos.data;

import tacos.Taco;

public interface TacoRepository  {

  Taco save(Taco design);

}
```

Similarly, `OrderRepository` also declares a `save()` method:

```
package tacos.data;

import tacos.Order;

public interface OrderRepository {

  Order save(Order order);

}
```

Seems simple enough, right? Not so quick. Saving a taco design requires that you also save the ingredients associated with that taco to the `Taco_Ingredients` table. Likewise, saving an order requires that you also save the tacos associated with the order to the `Taco_Order_Tacos` table. This makes saving tacos and orders a bit more challenging than what was required to save an ingredient.

To implement `TacoRepository`, you need a `save()` method that starts by saving the essential taco design details (for example, the name and time of creation), and then inserts one row into `Taco_Ingredients` for each ingredient in the `Taco` object. The following listing shows the complete `JdbcTacoRepository` class.

Listing 3.10 Implementing `TacoRepository` with `JdbcTemplate`

```
package tacos.data;

import java.sql.Timestamp;
import java.sql.Types;
import java.util.Arrays;
import java.util.Date;

import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.PreparedStatementCreator;
import org.springframework.jdbc.core.PreparedStatementCreatorFactory;
import org.springframework.jdbc.support.GeneratedKeyHolder;
import org.springframework.jdbc.support.KeyHolder;
import org.springframework.stereotype.Repository;

import tacos.Ingredient;
import tacos.Taco;

@Repository
public class JdbcTacoRepository implements TacoRepository {

  private JdbcTemplate jdbc;

  public JdbcTacoRepository(JdbcTemplate jdbc) {
    this.jdbc = jdbc;
  }

  @Override
  public Taco save(Taco taco) {
    long tacoId = saveTacoInfo(taco);
    taco.setId(tacoId);
    for (Ingredient ingredient : taco.getIngredients()) {
      saveIngredientToTaco(ingredient, tacoId);
    }

    return taco;
  }

  private long saveTacoInfo(Taco taco) {
    taco.setCreatedAt(new Date());
    PreparedStatementCreator psc =
        new PreparedStatementCreatorFactory(
            "insert into Taco (name, createdAt) values (?, ?)",
            Types.VARCHAR, Types.TIMESTAMP
        ).newPreparedStatementCreator(
            Arrays.asList(
                taco.getName(),
                new Timestamp(taco.getCreatedAt().getTime())));

    KeyHolder keyHolder = new GeneratedKeyHolder();
    jdbc.update(psc, keyHolder);

    return keyHolder.getKey().longValue();
  }

  private void saveIngredientToTaco(
          Ingredient ingredient, long tacoId) {
    jdbc.update(
        "insert into Taco_Ingredients (taco, ingredient) " +
        "values (?, ?)",
        tacoId, ingredient.getId());
  }

}
```

As you can see, the `save()` method starts by calling the private `saveTacoInfo()` method, and then uses the taco ID returned from that method to call `saveIngredientToTaco()`, which saves each ingredient. The devil is in the details of `saveTacoInfo()`.

When you insert a row into `Taco`, you need to know the ID generated by the database so that you can reference it in each of the ingredients. The `update()` method, used when saving ingredient data, doesn't help you get at the generated ID, so you need a different `update()` method here.

The `update()` method you need accepts a `PreparedStatementCreator` and a `KeyHolder`. It's the `KeyHolder` that will provide the generated taco ID. But in order to use it, you must also create a `PreparedStatementCreator`.

As you can see from listing 3.10, creating a `PreparedStatementCreator` is non-trivial. Start by creating a `PreparedStatementCreatorFactory`, giving it the SQL you want to execute, as well as the types of each query parameter. Then call `newPreparedStatementCreator()` on that factory, passing in the values needed in the query parameters to produce the `PreparedStatementCreator`.

With a `PreparedStatementCreator` in hand, you can call `update()`, passing in `PreparedStatementCreator` and `KeyHolder` (in this case, a `GeneratedKeyHolder` instance). Once the `update()` is finished, you can return the taco ID by returning `keyHolder.getKey().longValue()`.

Back in `save()`, cycle through each `Ingredient` in `Taco`, calling `saveIngredientToTaco()`. The `saveIngredientToTaco()` method uses the simpler form of `update()` to save ingredient references to the `Taco_Ingredients` table.

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

Listing 3.11 Injecting and using `TacoRepository`

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

  private final IngredientRepository ingredientRepo;

  private TacoRepository designRepo;

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

  ...

}
```

As you can see, the constructor takes both an `IngredientRepository` and a `TacoRepository`. It assigns both to instance variables so that they can be used in the `showDesignForm()` and `processDesign()` methods.

Speaking of the `processDesign()` method, its changes are a bit more extensive than the changes you made to `showDesignForm()`. The next listing shows the new `processDesign()` method.

Listing 3.12 Saving taco designs and linking them to orders

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

  @ModelAttribute(name = "order")
  public Order order() {
    return new Order();
  }

  @ModelAttribute(name = "taco")
  public Taco taco() {
    return new Taco();
  }

  @PostMapping
  public String processDesign(
      @Valid Taco design, Errors errors,
      @ModelAttribute Order order) {

    if (errors.hasErrors()) {
      return "design";
    }

    Taco saved = designRepo.save(design);
    order.addDesign(saved);

    return "redirect:/orders/current";
  }

  ...

}
```

The first thing you'll notice about the code in listing 3.12 is that `DesignTacoController` is now annotated with `@SessionAttributes("order")` and that it has a new `@ModelAttribute` annotated method, `order()`. As with the `taco()` method, the `@ModelAttribute` annotation on `order()` ensures that an `Order` object will be created in the model. But unlike the `Taco` object in the session, you need the order to be present across multiple requests so that you can create multiple tacos and add them to the order. The class-level `@SessionAttributes` annotation specifies any model objects like the order attribute that should be kept in session and available across multiple requests.

The real processing of a taco design happens in the `processDesign()` method, which now accepts an `Order` object as a parameter, in addition to `Taco` and `Errors` objects. The `Order` parameter is annotated with `@ModelAttribute` to indicate that its value should come from the model and that Spring MVC shouldn't attempt to bind request parameters to it.

After checking for validation errors, `processDesign()` uses the injected `Taco-Repository` to save the taco. It then adds the `Taco` object to the `Order` that's kept in the session.

In fact, the `Order` object remains in the session and isn't saved to the database until the user completes and submits the order form. At that point, `OrderController` needs to call out to an implementation of `OrderRepository` to save the order. Let's write that implementation.

Inserting data with SimpleJdbcInsert

You'll recall that saving a taco involved not only saving the taco's name and creation time to the `Taco` table, but also saving a reference to the ingredients associated with the taco to the `Taco_Ingredients` table. And you'll also recall that this required you to know the `Taco`'s ID, which you obtained using `KeyHolder` and `PreparedStatementCreator`.

When it comes to saving orders, a similar circumstance exists. You must not only save the order data to the `Taco_Order` table, but also references to each taco in the order to the `Taco_Order_Tacos` table. But rather than use the cumbersome `Prepared-StatementCreator`, allow me to introduce you to `SimpleJdbcInsert`, an object that wraps `JdbcTemplate` to make it easier to insert data into a table.

You'll start by creating `JdbcOrderRepository`, an implementation of `OrderRepos-itory`. But before you write the `save()` method implementation, let's focus on the constructor, where you'll create a couple of instances of `SimpleJdbcInsert` for inserting values into the `Taco_Order` and `Taco_Order_Tacos` tables. The following listing shows `JdbcOrderRepository` (without the `save()` method).

Listing 3.13 Creating a `SimpleJdbcInsert` from a `JdbcTemplate`

```
package tacos.data;

import java.util.Date;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.simple.SimpleJdbcInsert;
import org.springframework.stereotype.Repository;

import com.fasterxml.jackson.databind.ObjectMapper;
import tacos.Taco;
import tacos.Order;

@Repository
public class JdbcOrderRepository implements OrderRepository {

  private SimpleJdbcInsert orderInserter;
  private SimpleJdbcInsert orderTacoInserter;
  private ObjectMapper objectMapper;

  @Autowired
  public JdbcOrderRepository(JdbcTemplate jdbc) {
    this.orderInserter = new SimpleJdbcInsert(jdbc)
        .withTableName("Taco_Order")
        .usingGeneratedKeyColumns("id");

    this.orderTacoInserter = new SimpleJdbcInsert(jdbc)
        .withTableName("Taco_Order_Tacos");

    this.objectMapper = new ObjectMapper();
  }

...

}
```

Like `JdbcTacoRepository`, `JdbcOrderRepository` is injected with `JdbcTemplate` through its constructor. But instead of assigning `JdbcTemplate` directly to an instance variable, the constructor uses it to construct a couple of `SimpleJdbcInsert` instances.

The first instance, which is assigned to the `orderInserter` instance variable, is configured to work with the `Taco_Order` table and to assume that the `id` property will be provided or generated by the database. The second instance, assigned to `order-TacoInserter`, is configured to work with the `Taco_Order_Tacos` table but makes no claims about how any IDs will be generated in that table.

The constructor also creates an instance of Jackson's `ObjectMapper` and assigns it to an instance variable. Although Jackson is intended for JSON processing, you'll see in a moment how you'll repurpose it to help you as you save orders and their associated tacos.

Now let's take a look at how the `save()` method uses the `SimpleJdbcInsert` instances. The next listing shows the `save()` method, as well as a couple of private methods that `save()` delegates for the real work.

Listing 3.14 Using `SimpleJdbcInsert` to insert data

```
  @Override
  public Order save(Order order) {
    order.setPlacedAt(new Date());
    long orderId = saveOrderDetails(order);
    order.setId(orderId);
    List<Taco> tacos = order.getTacos();
    for (Taco taco : tacos) {
      saveTacoToOrder(taco, orderId);
    }

    return order;
  }

  private long saveOrderDetails(Order order) {
    @SuppressWarnings("unchecked")
    Map<String, Object> values =
        objectMapper.convertValue(order, Map.class);
    values.put("placedAt", order.getPlacedAt());

    long orderId =
        orderInserter
            .executeAndReturnKey(values)
            .longValue();
    return orderId;
  }

  private void saveTacoToOrder(Taco taco, long orderId) {
    Map<String, Object> values = new HashMap<>();
    values.put("tacoOrder", orderId);
    values.put("taco", taco.getId());
    orderTacoInserter.execute(values);
  }
```

The `save()` method doesn't actually save anything. It defines the flow for saving an `Order` and its associated `Taco` objects, and delegates the persistence work to `save-OrderDetails()` and `saveTacoToOrder()`.

`SimpleJdbcInsert` has a couple of useful methods for executing the insert: `execute()` and `executeAndReturnKey()`. Both accept a `Map<String, Object>`, where the map keys correspond to the column names in the table the data is inserted into. The map values are inserted into those columns.

It's easy to create such a `Map` by copying the values from `Order` into entries of the `Map`. But `Order` has several properties, and those properties all share the same name with the columns that they're going into. Because of that, in `saveOrderDetails()`, I've decided to use Jackson's `ObjectMapper` and its `convertValue()` method to convert an `Order` into a `Map`.[1] Once the `Map` is created, you'll set the `placedAt` entry to the value of the `Order` object's `placedAt` property. This is necessary because `Object-Mapper` would otherwise convert the `Date` property into a `long`, which is incompatible with the `placedAt` field in the `Taco_Order` table.

With a `Map` full of order data ready, you can now call `executeAndReturnKey()` on `orderInserter`. This saves the order information to the `Taco_Order` table and returns

[1] I'll admit that this is a hackish use of `ObjectMapper`, but you already have Jackson in the classpath; Spring Boot's web starter brings it in. Also, using `ObjectMapper` to map an object into a `Map` is much easier than copying each property from the object into the `Map`. Feel free to replace the use of `ObjectMapper` with any code you prefer that builds the `Map` you'll give to the inserter objects.

the database-generated ID as a `Number` object, which a call to `longValue()` converts to a `long` returned from the method.

The `saveTacoToOrder()` method is significantly simpler. Rather than use the `ObjectMapper` to convert an object to a `Map`, you create the `Map` and set the appropriate values. Once again, the map keys correspond to column names in the table. A simple call to the `orderTacoInserter`'s `execute()` method performs the insert.

Now you can inject `OrderRepository` into `OrderController` and start using it. The following listing shows the complete `OrderController`, including the changes to use an injected `OrderRepository`.

Listing 3.15 Using an `OrderRepository` in `OrderController`

```
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.Order;
import tacos.data.OrderRepository;

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

  private OrderRepository orderRepo;

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

  @GetMapping("/current")
  public String orderForm() {
    return "orderForm";
  }

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

    orderRepo.save(order);
    sessionStatus.setComplete();
    return "redirect:/";
  }

}
```

Aside from injecting `OrderRepository` into the controller, the only significant changes in `OrderController` are in the `processOrder()` method. Here, the `Order` object submitted in the form (which also happens to be the same `Order` object maintained in session) is saved via the `save()` method on the injected `OrderRepository`.

Once the order is saved, you don't need it hanging around in a session anymore. In fact, if you don't clean it out, the order remains in session, including its associated tacos, and the next order will start with whatever tacos the old order contained. Therefore, the `processOrder()` method asks for a `SessionStatus` parameter and calls its `setComplete()` method to reset the session.

All of the JDBC persistence code is in place. Now you can fire up the Taco Cloud application and try it out. Feel free to create as many tacos and as many orders as you'd like.

You might also find it helpful to dig around in the database. Because you're using H2 as your embedded database, and because you have Spring Boot DevTools in place, you should be able to point your browser to http://localhost:8080/h2-console to see the H2 Console. The default credentials should get you in, although you'll need to be sure that the JDBC URL field is set to `jdbc:h2:mem:testdb`. Once logged in, you should be able to issue any query you like against the tables in the Taco Cloud schema.

Spring's `JdbcTemplate`, along with `SimpleJdbcInsert`, makes working with relational databases significantly simpler than plain vanilla JDBC. But you may find that JPA makes it even easier. Let's rewind your work and see how to use Spring Data to make data persistence even easier.

3.2 Persisting data with Spring Data JPA

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 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 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.

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.

3.2.1 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>
      <artifactId>hibernate-entitymanager</artifactId>
      <groupId>org.hibernate</groupId>
    </exclusion>
  </exclusions>
</dependency>
<dependency>
  <groupId>org.eclipse.persistence</groupId>
  <artifactId>eclipselink</artifactId>
  <version>2.5.2</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.

3.2.2 Annotating the domain as entities

As you'll soon see, 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 `Order` classes and throw in a few annotations. First up is the `Ingredient` class.

Listing 3.16 Annotating `Ingredient` for JPA persistence

```
package tacos;

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

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

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

  @Id
  private final String id;
  private final String name;
  private final 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 `null`.

You also add a `@RequiredArgsConstructor`. The `@Data` implicitly adds a required arguments constructor, but when a `@NoArgsConstructor` is used, that constructor gets removed. An explicit `@RequiredArgsConstructor` ensures that you'll still have a required arguments constructor in addition to the `private` no-arguments constructor.

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

Listing 3.17 Annotating `Taco` as an entity

```
package tacos;
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.persistence.OneToMany;
import javax.persistence.PrePersist;
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;

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

  @PrePersist
  void createdAt() {
    this.createdAt = new Date();
  }
}
```

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 `Tacos`.

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 `Order` object as an entity. The next listing shows the new `Order` class.

Listing 3.18 Annotating `Order` 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.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.ManyToMany;
import javax.persistence.OneToMany;
import javax.persistence.PrePersist;
import javax.persistence.Table;
import javax.validation.constraints.Digits;
import javax.validation.constraints.Pattern;
import org.hibernate.validator.constraints.CreditCardNumber;
import org.hibernate.validator.constraints.NotBlank;
import lombok.Data;

@Data
@Entity
@Table(name="Taco_Order")
public class Order implements Serializable {

  private static final long serialVersionUID = 1L;

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

  private Date placedAt;

  ...

  @ManyToMany(targetEntity=Taco.class)
  private List<Taco> tacos = new ArrayList<>();

  public void addDesign(Taco design) {
    this.tacos.add(design);
  }

  @PrePersist
  void placedAt() {
    this.placedAt = new Date();
  }

}
```

As you can see, the changes to `Order` closely mirror the changes to `Taco`. But there's one new annotation at the class level: `@Table`. This specifies that `Order` entities should be persisted to a table named `Taco_Order` in the database.

Although you could have used this annotation on any of the entities, it's necessary with `Order`. Without it, JPA would default to persisting the entities to a table named `Order`, but *order* is a reserved word in SQL and would cause problems. Now that the entities are properly annotated, it's time to write your repositories.

3.2.3 Declaring JPA repositories

In the JDBC versions of the repositories, you explicitly declared the methods you wanted the repository to provide. But with Spring Data, you can extend the `CrudRepository` interface instead. 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> {

}
```

`CrudRepository` declares about a dozen methods for CRUD (create, read, update, delete) operations. Notice that it's parameterized, with the first parameter being the entity type the repository is to persist, and the second parameter being the type of the entity ID property. For `IngredientRepository`, the parameters should be `Ingredient` and `String`.

You can similarly define the `TacoRepository` like this:

```
package tacos.data;

import org.springframework.data.repository.CrudRepository;

import tacos.Taco;

public interface TacoRepository
         extends CrudRepository<Taco, Long> {

}
```

The only significant differences between `IngredientRepository` and `TacoRepository` are the parameters to `CrudRepository`. Here, they're set to `Taco` and `Long` to specify the `Taco` entity (and its ID type) as the unit of persistence for this repository interface. Finally, the same changes can be applied to `OrderRepository`:

```
package tacos.data;

import org.springframework.data.repository.CrudRepository;

import tacos.Order;
```

```
public interface OrderRepository
         extends CrudRepository<Order, Long> {

}
```

And now you have your three repositories. You might be thinking that you need to write the implementations for all three, including the dozen methods for each implementation. But that's the good news about Spring Data JPA—there's no need to write an implementation! When the application starts, Spring Data JPA automatically generates an implementation on the fly. This means the repositories are ready to use from the get-go. Just inject them into the controllers like you did for the JDBC-based implementations, and you're done.

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.

3.2.4 Customizing JPA 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<Order> 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 (an `Order`, 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 `Order`s, because you've parameterized `CrudRepository` with `Order`. The method name, `findByDeliveryZip()`, makes it clear that this method should find all `Order` 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 an `Order`.

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<Order> 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.

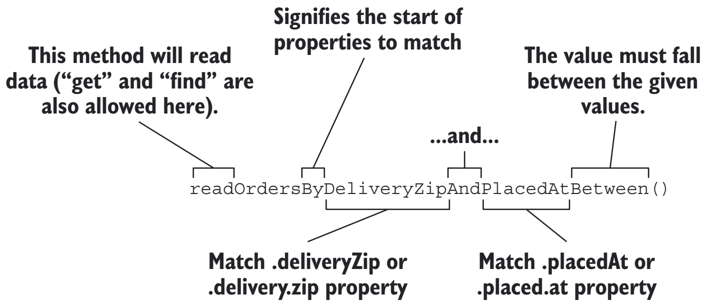

Signifies the start of properties to match
This method will read data ("get" and "find" are also allowed here).
The value must fall between the given values.
...and...
readOrdersByDeliveryZipAndPlacedAtBetween()
Match .deliveryZip or .delivery.zip property
Match .placedAt or .placed.at property

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 `Order` 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 `Order` 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<Order> 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<Order> 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<Order> readOrdersDeliveredInSeattle();
```

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

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.
- For easy execution of data inserts, use `SimpleJdbcInsert`.
- Spring Data JPA makes JPA persistence as easy as writing a repository interface.

4 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.

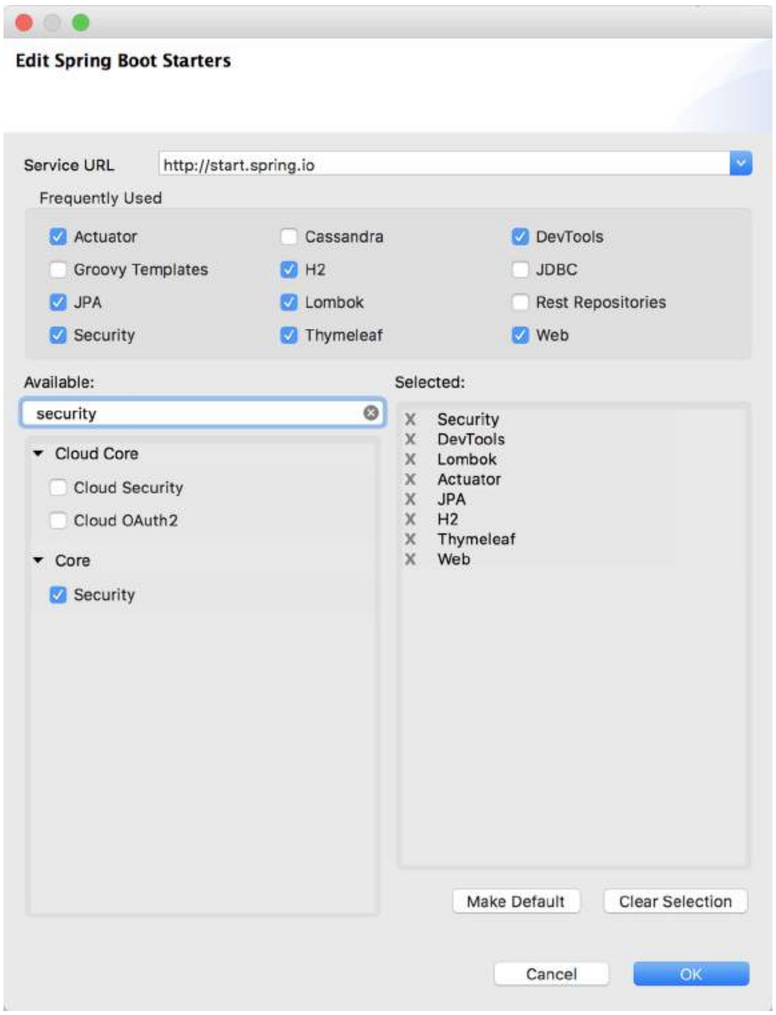
Information is probably the most valuable item we now have; crooks are looking for ways to steal our data and identities by sneaking into unsecured applications. As software developers, we must take steps to protect the information that resides in our applications. Whether it's an email account protected with a username-password pair or a brokerage account protected with a trading PIN, security is a crucial aspect of most applications.

4.1 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 Security entry under the Core category, as shown in figure 4.1.



Edit Spring Boot Starters

Service URL: http://start.spring.io

Frequently Used

✓ Actuator	Cassandra	✓ DevTools
Groovy Templates	✓ H2	JDBC
✓ JPA	✓ Lombok	Rest Repositories
✓ Security	✓ Thymeleaf	✓ Web

Available: security

- Cloud Core
  - Cloud Security
  - Cloud OAuth2
- Core
  - ✓ Security

Selected:

X Security
X DevTools
X Lombok
X Actuator
X JPA
X H2
X Thymeleaf
X Web

Figure 4.1 **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 an HTTP basic authentication dialog box. 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 default 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.
- There's no login page.
- Authentication is prompted with HTTP basic authentication.
- 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:

- Prompt for authentication with a login page, instead of an HTTP basic dialog box.
- 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.

4.2 Configuring Spring Security

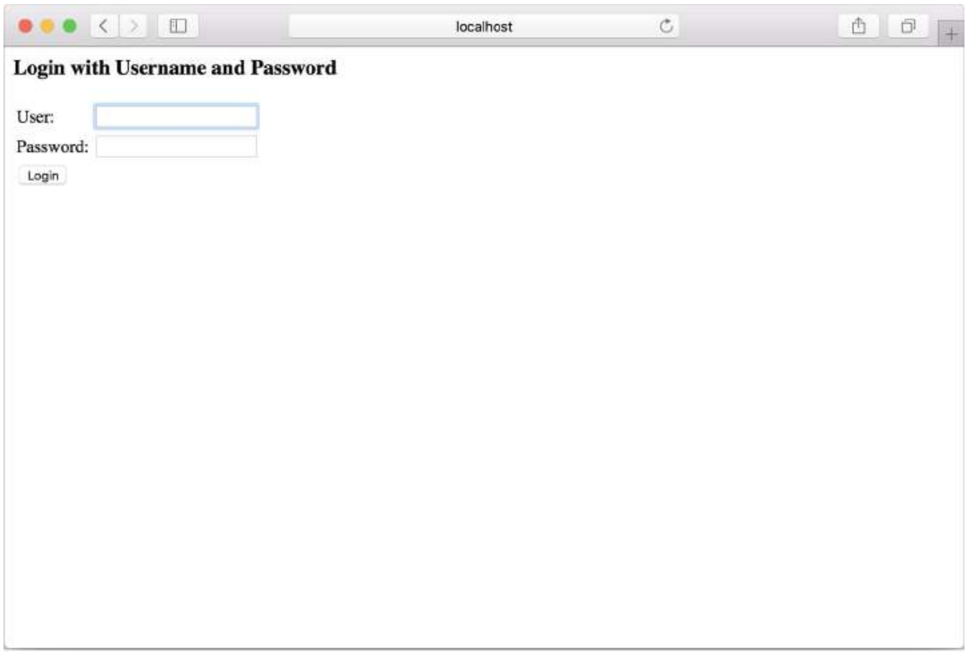
Over the years there have been several ways of configuring Spring Security, including lengthy XML-based configuration. Fortunately, several recent versions of Spring Security have supported Java-based configuration, which is much easier to read and write.

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 the barebones configuration class shown in the following listing.

Listing 4.1 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.config.annotation.web
                        .configuration.EnableWebSecurity;
import org.springframework.security.config.annotation.web
                    .configuration.WebSecurityConfigurerAdapter;

@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

}
```

What does this barebones security configuration do for you? Well, not much, but it does move you a step closer to the security functionality you need. If you attempt to hit the Taco Cloud homepage again, you'll still be prompted to sign in. But instead of being prompted with an HTTP basic authentication dialog box, you'll be shown a login form like the one in figure 4.2.



Login with Username and Password

User:

Password:

Figure 4.2 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.

This is a small improvement—prompting for login with a web page (even if it is rather plain in appearance) is always more user-friendly than an HTTP basic dialog box. You'll customize the login page in section 4.3.2. The current task at hand, however, is to configure a user store that can handle more than one user.

As it turns out, Spring Security offers several options for configuring a user store, including these:

- An in-memory user store
- A JDBC-based user store
- An LDAP-backed user store
- A custom user details service

No matter which user store you choose, you can configure it by overriding a `configure()` method defined in the `WebSecurityConfigurerAdapter` configuration base class. To get started, you'll add the following method override to the `SecurityConfig` class:

```
@Override
protected void configure(AuthenticationManagerBuilder auth)
    throws Exception {

   ...

}
```

Now you just need to replace those ellipses with code that uses the given `AuthenticationManagerBuilder` to specify how users will be looked up during authentication. First up, you'll try the in-memory user store.

4.2.1 In-memory user store

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.

For example, the next listing shows how to configure two users, `"buzz"` and `"woody"`, in an in-memory user store.

Listing 4.2 Defining users in an in-memory user store

```
@Override
protected void configure(AuthenticationManagerBuilder auth)
    throws Exception {
```

```
  auth
    .inMemoryAuthentication()
      .withUser("buzz")
        .password("infinity")
        .authorities("ROLE_USER")
      .and()
      .withUser("woody")
        .password("bullseye")
        .authorities("ROLE_USER");

}
```

As you can see, `AuthenticationManagerBuilder` employs a builder-style API to configure authentication details. In this case, a call to the `inMemoryAuthentication()` method gives you an opportunity to specify user information directly in the security configuration itself.

Each call to `withUser()` starts the configuration for a user. The value given to `withUser()` is the username, whereas the password and granted authorities are specified with the `password()` and `authorities()` methods. As shown in listing 4.2, both users are granted `ROLE_USER` authority. User buzz is configured to have *infinity* as their password. Likewise, woody's password is *bullseye*.

The in-memory user store 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 store, so let's take a look at another option that allows for a database-backed user store.

4.2.2 JDBC-based user store

User information is often maintained in a relational database, and a JDBC-based user store seems appropriate. The following listing shows how to configure Spring Security to authenticate against user information kept in a relational database with JDBC.

Listing 4.3 Authenticating against a JDBC-based user store

```
@Autowired
DataSource dataSource;

@Override
protected void configure(AuthenticationManagerBuilder auth)
    throws Exception {

  auth
    .jdbcAuthentication()
      .dataSource(dataSource);

}
```

This implementation of `configure()` calls `jdbcAuthentication()` on the given `AuthenticationManagerBuilder`. From there, you must also set the `DataSource` so that it knows how to access the database. The `DataSource` used here is provided by the magic of autowiring.

OVERRIDING THE DEFAULT USER QUERIES

Although this minimal configuration will work, it makes some assumptions about your database schema. It expects that certain tables exist where user data will be kept. More specifically, the following snippet of code from Spring Security's internals shows the SQL queries that will be performed when looking up user details:

```
public static final String DEF_USERS_BY_USERNAME_QUERY =
        "select username,password,enabled " +
        "from users " +
        "where username = ?";
public static final String DEF_AUTHORITIES_BY_USERNAME_QUERY =
        "select username,authority " +
        "from authorities " +
        "where username = ?";
public static final String DEF_GROUP_AUTHORITIES_BY_USERNAME_QUERY =
        "select g.id, g.group_name, ga.authority " +
        "from groups g, group_members gm, group_authorities ga " +
        "where gm.username = ? " +
        "and g.id = ga.group_id " +
        "and g.id = gm.group_id";
```

The first query retrieves a user's username, password, and whether or not they're enabled. This information is used to authenticate the user. The next query looks up the user's granted authorities for authorization purposes, and the final query looks up authorities granted to a user as a member of a group.

If you're OK with defining and populating tables in your database that satisfy those queries, there's not much else for you to do. But chances are your database doesn't look anything like this, and you'll want more control over the queries. In that case, you can configure your own queries.

Listing 4.4 Customizing user detail queries

```
@Override
protected void configure(AuthenticationManagerBuilder auth)
    throws Exception {

  auth
    .jdbcAuthentication()
      .dataSource(dataSource)
      .usersByUsernameQuery(
          "select username, password, enabled from Users " +
          "where username=?")
      .authoritiesByUsernameQuery(
          "select username, authority from UserAuthorities " +
          "where username=?");

}
```

In this case, you only override the authentication and basic authorization queries. But you can also override the group authorities query by calling `groupAuthoritiesByUsername()` with a custom query.

When replacing the default SQL queries with those of your own design, it's important to adhere to the basic contract of the queries. All of them take the username as their only parameter. The authentication query selects the username, password, and enabled status. The authorities query selects zero or more rows containing the username and a granted authority. The group authorities query selects zero or more rows, each with a group ID, a group name, and an authority.

Working with encoded passwords

Focusing on the authentication query, you can see that user passwords are expected to be stored in the database. The only problem with this is that if the passwords are stored in plain text, they're subject to the prying eyes of a hacker. But if you encode the passwords in the database, authentication will fail because it won't match the plaintext password submitted by the user.

To remedy this problem, you need to specify a password encoder by calling the `passwordEncoder()` method:

```
@Override
protected void configure(AuthenticationManagerBuilder auth)
    throws Exception {

  auth
    .jdbcAuthentication()
      .dataSource(dataSource)
      .usersByUsernameQuery(
          "select username, password, enabled from Users " +
          "where username=?")
      .authoritiesByUsernameQuery(
          "select username, authority from UserAuthorities " +
          "where username=?")
      .passwordEncoder(new StandardPasswordEncoder("53cr3t");

}
```

The `passwordEncoder()` method accepts any implementation of Spring Security's `PasswordEncoder` interface. Spring Security's cryptography module includes several such implementations:

- `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

The preceding code uses `StandardPasswordEncoder`. But you can choose any of the other implementations or even provide your own custom implementation if none of the out-of-the-box implementations meet your needs. The PasswordEncoder interface is rather simple:

```
public interface PasswordEncoder {
  String encode(CharSequence rawPassword);
  boolean matches(CharSequence rawPassword, String encodedPassword);
}
```

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.

Ultimately, you'll maintain Taco Cloud user data in a database. Rather than use jdbcAuthentication(), however, I've got another authentication option in mind. But before we go there, let's look at how you can configure Spring Security to rely on another common source of user data: a user store accessed with LDAP (Lightweight Directory Access Protocol).

4.2.3 LDAP-backed user store

To configure Spring Security for LDAP-based authentication, you can use the ldapAuthentication() method. This method is the LDAP analog to jdbcAuthentication(). The following configure() method shows a simple configuration for LDAP authentication:

```
@Override
protected void configure(AuthenticationManagerBuilder auth)
    throws Exception {
  auth
    .ldapAuthentication()
      .userSearchFilter("(uid={0})")
      .groupSearchFilter("member={0}");
}
```

The userSearchFilter() and groupSearchFilter() methods are used to provide filters for the base LDAP queries, which are used to search for users and groups. By default, the base queries for both users and groups are empty, indicating that the search will be done from the root of the LDAP hierarchy. But you can change that by specifying a query base:

```
@Override
protected void configure(AuthenticationManagerBuilder auth)
    throws Exception {
  auth
    .ldapAuthentication()
      .userSearchBase("ou=people")
      .userSearchFilter("(uid={0})")
      .groupSearchBase("ou=groups")
      .groupSearchFilter("member={0}");
}
```

The `userSearchBase()` method provides a base query for finding users. Likewise, the `groupSearchBase()` method specifies the base query for finding groups. Rather than search from the root, this example specifies that users be searched for where the organizational unit is `people`. Groups should be searched for where the organizational unit is `groups`.

Configuring password comparison

The default strategy for authenticating against LDAP is to perform a bind operation, authenticating the user directly to the LDAP server. Another option is to perform a comparison operation. This involves sending the entered password to the LDAP directory and asking the server to compare the password against a user's password attribute. Because the comparison is done within the LDAP server, the actual password remains secret.

If you'd rather authenticate by doing a password comparison, you can declare so with the `passwordCompare()` method:

```
@Override
protected void configure(AuthenticationManagerBuilder auth)
    throws Exception {
  auth
    .ldapAuthentication()
      .userSearchBase("ou=people")
      .userSearchFilter("(uid={0})")
      .groupSearchBase("ou=groups")
      .groupSearchFilter("member={0}")
      .passwordCompare();
}
```

By default, the password given in the login form will be compared with the value of the `userPassword` attribute in the user's LDAP entry. If the password is kept in a different attribute, you can specify the password attribute's name with `passwordAttribute()`:

```
@Override
protected void configure(AuthenticationManagerBuilder auth)
    throws Exception {
  auth
    .ldapAuthentication()
      .userSearchBase("ou=people")
      .userSearchFilter("(uid={0})")
      .groupSearchBase("ou=groups")
      .groupSearchFilter("member={0}")
      .passwordCompare()
      .passwordEncoder(new BCryptPasswordEncoder())
      .passwordAttribute("passcode");
}
```

In this example, you specify that the `passcode` attribute is what should be compared with the given password. Moreover, you also specify a password encoder. It's nice that the actual password is kept secret on the server when doing server-side password comparison. But the attempted password is still passed across the wire to the LDAP server and could be intercepted by a hacker. To prevent that, you can specify an encryption strategy by calling the `passwordEncoder()` method.

In the preceding example, passwords are encrypted using the `bcrypt` password hashing function. This assumes that the passwords are also encrypted using `bcrypt` in the LDAP server.

Referring to a remote LDAP server

The one thing I've left out until now is where the LDAP server and data actually reside. You've been happily configuring Spring to authenticate against an LDAP server, but where is that server?

By default, Spring Security's LDAP authentication assumes that the LDAP server is listening on port 33389 on localhost. But if your LDAP server is on another machine, you can use the `contextSource()` method to configure the location:

```
@Override
protected void configure(AuthenticationManagerBuilder auth)
    throws Exception {
  auth
    .ldapAuthentication()
      .userSearchBase("ou=people")
      .userSearchFilter("(uid={0})")
      .groupSearchBase("ou=groups")
      .groupSearchFilter("member={0}")
      .passwordCompare()
      .passwordEncoder(new BCryptPasswordEncoder())
      .passwordAttribute("passcode")
      .contextSource()
        .url("ldap://tacocloud.com:389/dc=tacocloud,dc=com");
}
```

The `contextSource()` method returns a `ContextSourceBuilder`, which, among other things, offers the `url()` method, which lets you specify the location of the LDAP server.

Configuring an embedded LDAP server

If you don't happen to have an LDAP server lying around waiting to be authenticated against, Spring Security can provide an embedded LDAP server for you. Instead of setting the URL to a remote LDAP server, you can specify the root suffix for the embedded server via the `root()` method:

```
@Override
protected void configure(AuthenticationManagerBuilder auth)
    throws Exception {
  auth
    .ldapAuthentication()
      .userSearchBase("ou=people")
      .userSearchFilter("(uid={0})")
      .groupSearchBase("ou=groups")
      .groupSearchFilter("member={0}")
      .passwordCompare()
      .passwordEncoder(new BCryptPasswordEncoder())
      .passwordAttribute("passcode")
      .contextSource()
        .root("dc=tacocloud,dc=com");
}
```

When the LDAP server starts, it will attempt to load data from any LDIF files that it can find in the classpath. LDIF (LDAP Data Interchange Format) is a standard way of representing LDAP data in a plain text file. Each record is composed of one or more lines, each containing a `name:value` pair. Records are separated from each other by blank lines.

If you'd rather that Spring not rummage through your classpath looking for any LDIF files it can find, you can be more explicit about which LDIF file gets loaded by calling the `ldif()` method:

```
@Override
protected void configure(AuthenticationManagerBuilder auth)
    throws Exception {
  auth
    .ldapAuthentication()
      .userSearchBase("ou=people")
      .userSearchFilter("(uid={0})")
      .groupSearchBase("ou=groups")
      .groupSearchFilter("member={0}")
      .passwordCompare()
      .passwordEncoder(new BCryptPasswordEncoder())
      .passwordAttribute("passcode")
      .contextSource()
        .root("dc=tacocloud,dc=com")
        .ldif("classpath:users.ldif");
}
```

Here, you specifically ask the LDAP server to load its content from the users.ldif file at the root of the classpath. In case you're curious, here's an LDIF file that you could use to load the embedded LDAP server with user data:

```
dn: ou=groups,dc=tacocloud,dc=com
objectclass: top
objectclass: organizationalUnit
ou: groups
dn: ou=people,dc=tacocloud,dc=com
objectclass: top
objectclass: organizationalUnit
ou: people
dn: uid=buzz,ou=people,dc=tacocloud,dc=com
objectclass: top
objectclass: person
objectclass: organizationalPerson
objectclass: inetOrgPerson
cn: Buzz Lightyear
sn: Lightyear
uid: buzz
userPassword: password
dn: cn=tacocloud,ou=groups,dc=tacocloud,dc=com
```

```
objectclass: top
objectclass: groupOfNames
cn: tacocloud
member: uid=buzz,ou=people,dc=tacocloud,dc=com
```

Spring Security's built-in user stores are convenient and cover some common use cases. But the Taco Cloud application needs something a bit special. When the out-of-the-box user stores don't meet your needs, you'll need to create and configure a custom user details service.

4.2.4 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 4.5 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;
  }

}
```

You've 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

Spring Security's UserDetailsService is a rather straightforward interface:

```
public interface UserDetailsService {
  UserDetails loadUserByUsername(String username)
                     throws UsernameNotFoundException;
}
```

As you can see, implementations of this interface are given a user's username and are expected to either return a UserDetails object or throw a UsernameNotFoundException if the given username doesn't turn up any results.

Because your User class implements UserDetails, and because UserRepository provides a findByUsername() method, they're perfectly suitable for use in a custom UserDetailsService implementation. The following listing shows the user details service you'll use in the Taco Cloud application.

Listing 4.6 Defining a custom user details service

```
package tacos.security;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.
                                              UserDetailsService;
import org.springframework.security.core.userdetails.
                                       UsernameNotFoundException;
import org.springframework.stereotype.Service;

import tacos.User;
import tacos.data.UserRepository;
```

```
@Service
public class UserRepositoryUserDetailsService
        implements UserDetailsService {

  private UserRepository userRepo;

  @Autowired
  public UserRepositoryUserDetailsService(UserRepository userRepo) {
    this.userRepo = userRepo;
  }

  @Override
  public UserDetails loadUserByUsername(String username)
      throws UsernameNotFoundException {
    User user = userRepo.findByUsername(username);
    if (user != null) {
      return user;
    }
    throw new UsernameNotFoundException(
                    "User '" + username + "' not found");
  }

}
```

`UserRepositoryUserDetailsService` is injected with an instance of `UserRepository` through its constructor. Then, in its `loadByUsername()` method, it calls `findByUsername()` on the `UserRepository` to look up a `User`.

The `loadByUsername()` method has one simple rule: it must never return `null`. Therefore, if the call to `findByUsername()` returns `null`, `loadByUsername()` will throw a `UsernameNotFoundException`. Otherwise, the `User` that was found will be returned.

You'll notice that `UserRepositoryUserDetailsService` is annotated with `@Service`. This is another one of Spring's stereotype annotations that flag it for inclusion in Spring's component scanning, so there's no need to explicitly declare this class as a bean. Spring will automatically discover it and instantiate it as a bean.

You do, however, still need to configure your custom user details service with Spring Security. Therefore, you'll return to the `configure()` method once more:

```
@Autowired
private UserDetailsService userDetailsService;

@Override
protected void configure(AuthenticationManagerBuilder auth)
    throws Exception {

  auth
    .userDetailsService(userDetailsService);

}
```

This time, you simply make a call to the `userDetailsService()` method, passing in the `UserDetailsService` instance that has been autowired into `SecurityConfig`.

As with JDBC-based authentication, you can (and should) also configure a password encoder so that the password can be encoded in the database. You'll do this by first declaring a bean of type `PasswordEncoder` and then injecting it into your user details service configuration by calling `passwordEncoder()`:

```
@Bean
public PasswordEncoder encoder() {
  return new StandardPasswordEncoder("53cr3t");
}

@Override
protected void configure(AuthenticationManagerBuilder auth)
    throws Exception {

  auth
    .userDetailsService(userDetailsService)
    .passwordEncoder(encoder());

}
```

It's important that we discuss the last line in the `configure()` method. It would appear that you call the `encoder()` method and pass its return value to `passwordEncoder()`. In reality, however, because the `encoder()` method is annotated with `@Bean`, it will be used to declare a `PasswordEncoder` bean in the Spring application context. Any calls to `encoder()` will then be intercepted to return the bean instance from the application context.

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 4.7 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 4.8 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.

4.3 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, let me introduce you to `WebSecurityConfigurerAdapter`'s other `configure()` method:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
  ...
}
```

This `configure()` method accepts an `HttpSecurity` object, which can be used to configure how security is handled at the web level. 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.

4.3.1 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 `configure()` implementation does exactly that:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
  http
    .authorizeRequests()
      .antMatchers("/design", "/orders")
        .hasRole("ROLE_USER")
      .antMatchers("/", "/**").permitAll()
    ;
}
```

The call to `authorizeRequests()` returns an object (`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`.
- 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 4.1 describes all the available methods.

Table 4.1 Configuration methods to define how a path is to be secured

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

Table 4.1 Configuration methods to define how a path is to be secured *(continued)*

Method	What it does
`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 4.1 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 4.2.

Table 4.2 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`
`hasAnyRole(list of roles)`	`true` if the user has any of the given roles
`hasRole(role)`	`true` if the user has the given role
`hasIpAddress(IP address)`	`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 4.2 correspond to similar methods in table 4.1. In fact, using the `access()` method along with the `has-Role()` and `permitAll` expressions, you can rewrite `configure()` as follows.

Listing 4.9 Using Spring expressions to define authorization rules

```
@Override
protected void configure(HttpSecurity http) throws Exception {
  http
    .authorizeRequests()
```

```
        .antMatchers("/design", "/orders")
          .access("hasRole('ROLE_USER')")
        .antMatchers("/", "/**").access("permitAll")
      ;
}
```

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 `configure()`:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
  http
    .authorizeRequests()
      .antMatchers("/design", "/orders")
        .access("hasRole('ROLE_USER') && " +
          "T(java.util.Calendar).getInstance().get("+
          "T(java.util.Calendar).DAY_OF_WEEK) == " +
          "T(java.util.Calendar).TUESDAY")
      .antMatchers("/", "/**").access("permitAll")
    ;
}
```

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 4.9. Now let's see about customizing the login page to fit the look of the Taco Cloud application.

4.3.2 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 passed into `configure()`:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
  http
    .authorizeRequests()
      .antMatchers("/design", "/orders")
        .access("hasRole('ROLE_USER')")
      .antMatchers("/", "/**").access("permitAll")

    .and()
      .formLogin()
```

```
        .loginPage("/login")
    ;
}
```

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>

    <!-- tag::thAction[] -->
    <form method="POST" th:action="@{/login}" id="loginForm">
    <!-- end::thAction[] -->
      <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)
```

Now that you've dealt with your custom login page, let's flip to the other side of the authentication coin and see how you can enable a user to log out.

4.3.3 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()
    .logoutSuccessUrl("/")
```

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 `logoutSucessFilter()` to specify a different post-logout landing page:

```
.and()
  .logout()
    .logoutSuccessUrl("/")
```

In this case, users will be sent to the homepage following logout.

4.3.4 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.

4.4 Knowing your user

Often, it's not enough to simply know that the user has logged in. 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 `Order` object that's bound to the order form, it'd be nice if you could prepopulate the `Order` 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 `Order` entity with the `User` that created the order.

To achieve the desired connection between an `Order` entity and a `User` entity, you need to add a new property to the `Order` class:

```
@Data
@Entity
```

```
@Table(name="Taco_Order")
public class Order 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 `Order` 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 Order 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 Order 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 `Order`.

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.

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`.

5 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 beans in the Spring