

# Spring IN ACTION

FIFTH EDITION

Craig Walls



 MANNING



*Spring in Action*  
*Fifth Edition*

by Craig Walls

**Chapter 2**

Copyright 2019 Manning Publications

# *brief contents*

---

<b>PART 1</b>	<b>FOUNDATIONAL SPRING .....</b>	<b>1</b>
	1 ■ Getting started with Spring	3
	2 ■ Developing web applications	29
	3 ■ Working with data	56
	4 ■ Securing Spring	84
	5 ■ Working with configuration properties	114
<b>PART 2</b>	<b>INTEGRATED SPRING .....</b>	<b>135</b>
	6 ■ Creating REST services	137
	7 ■ Consuming REST services	169
	8 ■ Sending messages asynchronously	178
	9 ■ Integrating Spring	209
<b>PART 3</b>	<b>REACTIVE SPRING .....</b>	<b>239</b>
	10 ■ Introducing Reactor	241
	11 ■ Developing reactive APIs	269
	12 ■ Persisting data reactively	296

<b>PART 4</b>	<b>CLOUD-NATIVE SPRING .....</b>	<b>321</b>
13	■ Discovering services	323
14	■ Managing configuration	343
15	■ Handling failure and latency	376
<b>PART 5</b>	<b>DEPLOYED SPRING .....</b>	<b>393</b>
16	■ Working with Spring Boot Actuator	395
17	■ Administering Spring	429
18	■ Monitoring Spring with JMX	446
19	■ Deploying Spring	454

# 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.

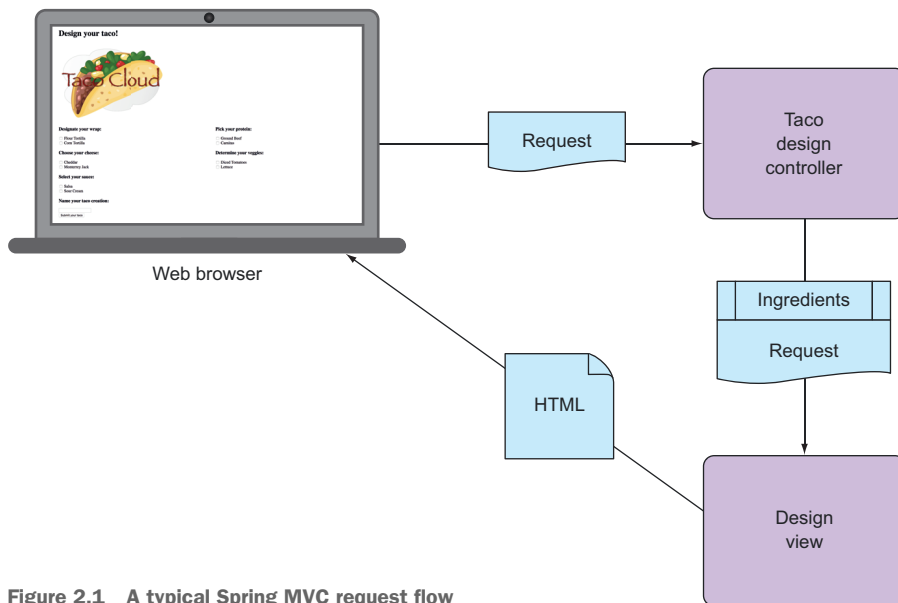


Figure 2.1 A typical Spring MVC request flow

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.<sup>1</sup> 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

---

<sup>1</sup> 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
<code>@RequestMapping</code>	General-purpose request handling
<code>@GetMapping</code>	Handles HTTP GET requests
<code>@PostMapping</code>	Handles HTTP POST requests
<code>@PutMapping</code>	Handles HTTP PUT requests
<code>@DeleteMapping</code>	Handles HTTP DELETE requests
<code>@PatchMapping</code>	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>
    

    <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.<sup>2</sup> 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.

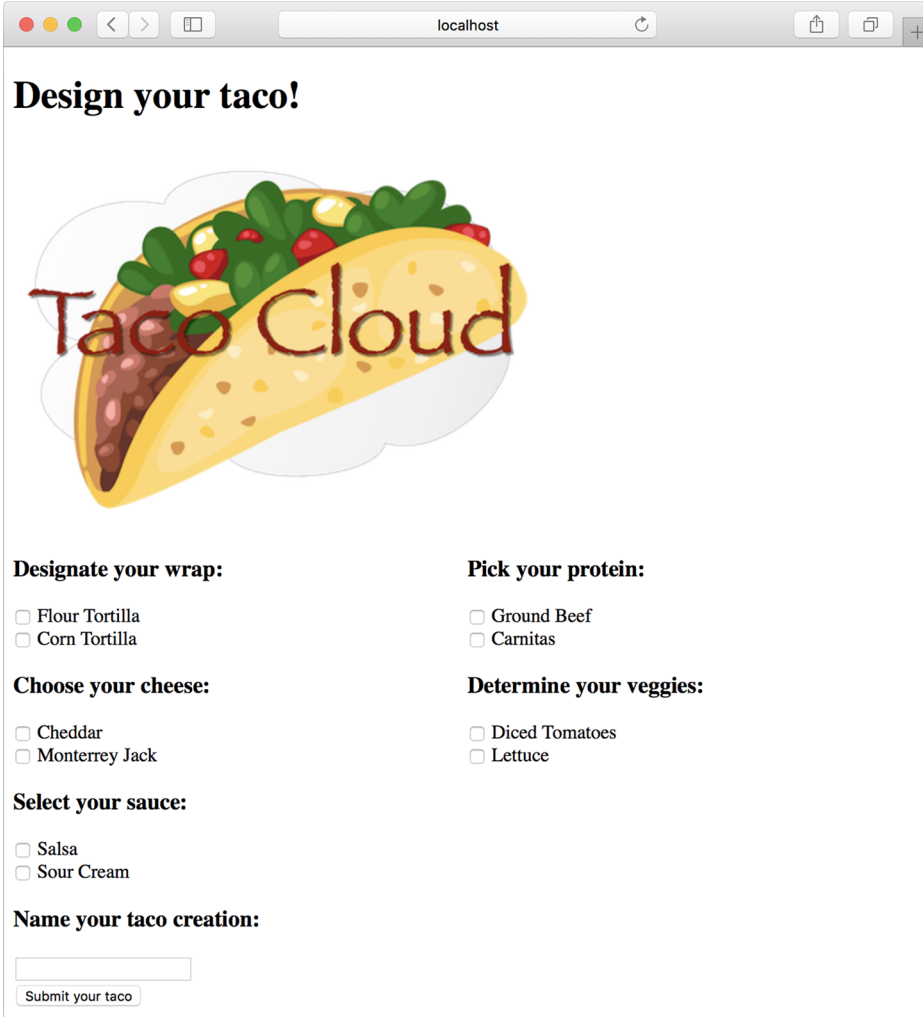
Now that your controller and view are complete, you can fire up the application to see the fruits of your labor. There are many ways to run a Spring Boot application. In chapter 1, I showed you how to run the application by first building it into an executable

---


<sup>2</sup> 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:**

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>

      
      <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>
    </form>

```

```

<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 {

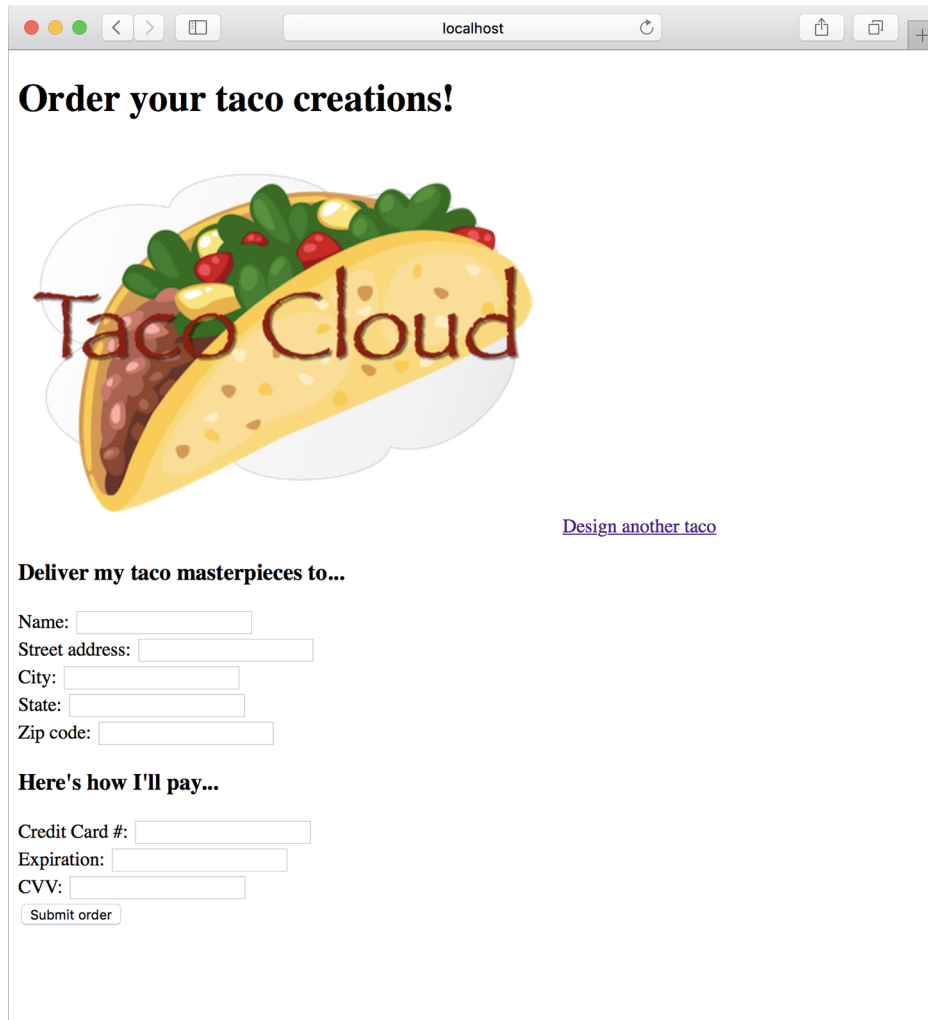
    private String name;
    private String street;
    private String city;
    private String state;
    private String zip;
    private String ccNumber;
    private String ccExpiration;
    private String ccCVV;
}
```

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](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

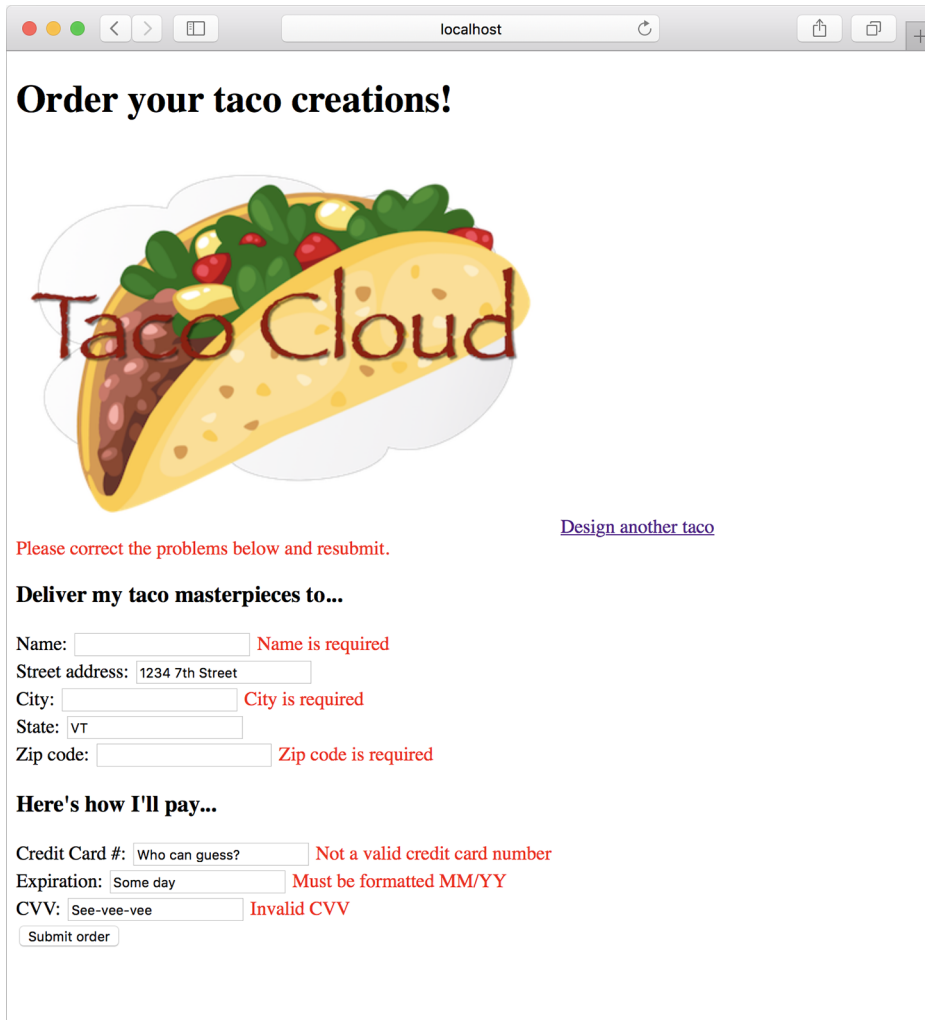
```
<label for="ccNumber">Credit Card #: </label>
<input type="text" th:field="*{ccNumber}"/>
<span class="validationError"
    th:if="${#fields.hasErrors('ccNumber')}"
    th:errors="*{ccNumber}">CC Num Error</span>
```

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.



The screenshot shows a web browser window with the address bar set to "localhost". The page has a title "Order your taco creations!" and a large illustration of a taco with the text "Taco Cloud" overlaid. Below the illustration, there is a link "Design another taco". The form is divided into two sections: "Deliver my taco masterpieces to..." and "Here's how I'll pay...". The first section contains fields for Name, Street address, City, State, and Zip code, each with a red error message. The second section contains fields for Credit Card #, Expiration, and CVV, each with a red error message. A "Submit order" button is at the bottom.

**Order your taco creations!**



[Design another taco](#)

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 `WebMvcConfigurer` interface. `WebMvcConfigurer` defines several methods for configuring Spring MVC. Even though it's an interface, it provides default implementations of all

the methods, so you only need to override the methods you need. In this case, you override `addViewControllers()`.

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

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

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

```
@SpringBootApplication
public class TacoCloudApplication implements WebMvcConfigurer {

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

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

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

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

## 2.5 *Choosing a view template library*

For the most part, your choice of a view template library is a matter of personal taste. Spring is very flexible and supports many common templating options. With only a

few small exceptions, the template library you choose will itself have no idea that it's even working with Spring.<sup>3</sup>

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>
```

<sup>3</sup> 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	<code>spring.freemarker.cache</code>
Groovy Templates	<code>spring.groovy.template.cache</code>
Mustache	<code>spring.mustache.cache</code>
Thymeleaf	<code>spring.thymeleaf.cache</code>

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 reenables 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.

# Spring IN ACTION Fifth Edition

Craig Walls

Free eBook  
See first page

**S**pring Framework makes life easier for Java developers. New features in Spring 5 bring its productivity-focused approach to microservices, reactive development, and other modern application designs. With Spring Boot now fully integrated, you can start even complex projects with minimal configuration code. And the upgraded WebFlux framework supports reactive apps right out of the box!

**Spring in Action, Fifth Edition** guides you through Spring's core features, explained in Craig Walls' famously clear style. You'll roll up your sleeves and build a secure database-backed web app step by step. Along the way, you'll explore reactive programming, microservices, service discovery, RESTful APIs, deployment, and expert best practices. Whether you're just discovering Spring or leveling up to Spring 5, this Manning classic is your ticket!

## What's Inside

- Building reactive applications
- Spring MVC for web apps and RESTful web services
- Securing applications with Spring Security
- Covers Spring 5.0

For intermediate Java developers.

**Craig Walls** is a principal software engineer at Pivotal, a popular author, an enthusiastic supporter of Spring Framework, and a frequent conference speaker.

“This new edition is a comprehensive update that strikes the balance between practical instruction and comprehensive theory.”

—Daniel Vaughan  
European Bioinformatics Institute

“The go-to book for learning the Spring Framework and an excellent reference guide.”

—Colin Joyce, Cisco

“Everything you need to know about Spring and how to build cloud-native applications.”

—David Witherspoon, Parsons

“This book is the Spring developer's Swiss Army knife!”

—Riccardo Noviello  
Nuvio Software Solutions

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit  
[manning.com/books/spring-in-action-fifth-edition](http://manning.com/books/spring-in-action-fifth-edition)

Over 100,000 copies sold!

 MANNING

\$49.99 / Can \$65.99 [INCLUDING eBook]

