covers Spring 3.0

Spring IN ACTION

THIRD EDITION

Craig Walls

MANNING

Table of Contents

7 Building web applications with Spring MVC

This chapter covers

- Mapping requests to Spring controllers
- Transparently binding form parameters
- Validating form submissions
- Uploading files

As an enterprise Java developer, you've likely developed a web-based application or two. For many Java developers, web-based applications are their primary focus. If you do have this type of experience, you're well aware of the challenges that come with these systems. Specifically, state management, workflow, and validation are all important features that need to be addressed. None of these is made any easier given the HTTP protocol's stateless nature.

Spring's web framework is designed to help you address these concerns. Based on the Model-View-Controller (MVC) pattern, Spring MVC helps you build web-based applications that are as flexible and as loosely coupled as the Spring Framework itself.

In this chapter we'll explore the Spring MVC web framework. We'll build controllers using the new Spring MVC annotations to handle web requests. As we do, we'll strive to design our web layer in a RESTful way. Finally, we'll wrap up by looking at how to use Spring's JSP tags in views to send a response back to the user.

Before we go too deep with the specifics of Spring MVC controllers and handler mappings, let's start with a high-level view of Spring MVC and set up the basic plumbing needed to make Spring MVC work.

7.1 Getting started with Spring MVC

Have you ever seen the children's game Mousetrap? It's crazy. The goal is to send a small steel ball over a series of wacky contraptions in order to trigger a mousetrap. The ball goes over all kinds of intricate gadgets, from rolling down a curvy ramp to springing off a teeter-totter to spinning on a miniature Ferris wheel to being kicked out of a bucket by a rubber boot. It goes through all of this to spring a trap on a poor, unsuspecting plastic mouse.

At first glance, you may think that Spring's MVC framework is a lot like Mousetrap. Instead of moving a ball around through various ramps, teeter-totters, and wheels, Spring moves requests around between a dispatcher servlet, handler mappings, controllers, and view resolvers.

But don't draw too strong of a comparison between Spring MVC and the Rube Goldbergesque game of Mousetrap. Each of the components in Spring MVC performs a specific purpose. Let's start our exploration of Spring MVC by examining the lifecycle of a typical request.

7.1.1 Following a request through Spring MVC

Every time a user clicks a link or submits a form in their web browser, a request goes to work. A request's job description is that of a courier. Just like a postal carrier or a FedEx delivery person, a request lives to carry information from one place to another.

The request is a busy fellow. From the time it leaves the browser until it returns with a response, it'll make several stops, each time dropping off a bit of information and picking up some more. Figure 7.1 shows all the stops that the request makes.

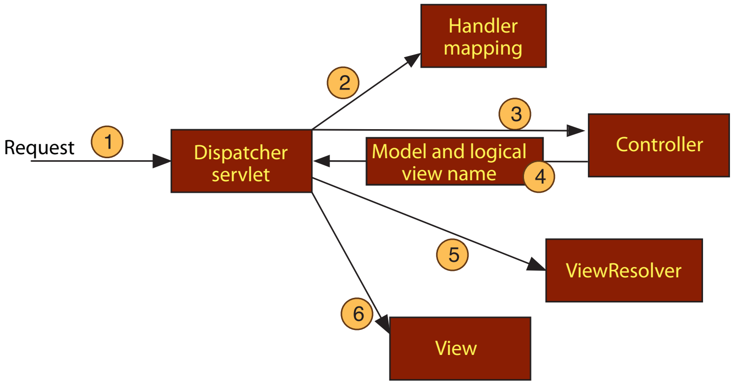


Figure 7.1 The web layer of the Spitter application includes two resource-oriented controllers along with a couple of utility controllers.

When the request leaves the browser, it carries information about what the user is asking for. At least, the request will be carrying the requested URL. But it may also carry additional data such as the information submitted in a form by the user.

The first stop in the request's travels is at Spring's `DispatcherServlet`. Like most Java-based web frameworks, Spring MVC funnels requests through a single front controller servlet. A front controller is a common web application pattern where a single servlet delegates responsibility for a request to other components of an application to perform actual processing. In the case of Spring MVC, `DispatcherServlet` is the front controller.

The `DispatcherServlet`'s job is to send the request on to a Spring MVC controller. A controller is a Spring component that processes the request. But a typical application may have several controllers and `DispatcherServlet` needs some help deciding which controller to send the request to. So the `DispatcherServlet` consults one or more handler mappings to figure out where the request's next stop will be. The handler mapping will pay particular attention to the URL carried by the request when making its decision.

Once an appropriate controller has been chosen, `DispatcherServlet` sends the request on its merry way to the chosen controller. At the controller, the request will drop off its payload (the information submitted by the user) and patiently wait while the controller processes that information. (Actually, a well-designed controller performs little or no processing itself and instead delegates responsibility for the business logic to one or more service objects.)

The logic performed by a controller often results in some information that needs to be carried back to the user and displayed in the browser. This information is referred to as the *model*. But sending raw information back to the user isn't sufficient—it needs to be formatted in a user-friendly format, typically HTML. For that the information needs to be given to a *view*, typically a JSP.

One of the last things that a controller does is package up the model data and identify the name of a view that should render the output. It then sends the request, along with the model and view name, back to the `DispatcherServlet`.

So that the controller doesn't get coupled to a particular view, the view name passed back to `DispatcherServlet` doesn't directly identify a specific JSP. In fact, it doesn't even necessarily suggest that the view is a JSP at all. Instead, it only carries a logical name which will be used to look up the actual view that will produce the result. The `DispatcherServlet` will consult a view resolver to map the logical view name to a specific view implementation, which may or may not be a JSP.

Now that `DispatcherServlet` knows which view will render the result, the request's job is almost over. Its final stop is at the view implementation (probably a JSP) where it delivers the model data. The request's job is finally done. The view will use the model data to render output that will be carried back to the client by the (not-so-hardworking) response object.

We'll dive into each of these steps in more detail throughout this chapter. But first things first—we need to set up Spring MVC and `DispatcherServlet` in the Spitter application.

7.1.2 Setting up Spring MVC

At the heart of Spring MVC is `DispatcherServlet`, a servlet that functions as Spring MVC's front controller. Like any servlet, `DispatcherServlet` must be configured in the web application's web.xml file. So the first thing we must do to use Spring MVC in our application is to place the following `<servlet>` declaration in the web.xml file:

```
<servlet>
    <servlet-name>spitter</servlet-name>
    <servlet-class>
       org.springframework.web.servlet.DispatcherServlet
     </servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
```

The `<servlet-name>` given to the servlet is significant. By default, when `DispatcherServlet` is loaded, it'll load the Spring application context from an XML file whose name is based on the name of the servlet. In this case, because the servlet is named `spitter`, `DispatcherServlet` will try to load the application context from a file named spitter-servlet.xml (located in the application's WEB-INF directory).

Next we must indicate what URLs will be handled by the `DispatcherServlet`. It's common to find `DispatcherServlet` mapped to URL patterns such as `*.htm`, `/*`, or `/app`. But these URL patterns have a few problems:

- The `*.htm` pattern implies that the response will always be in HTML form (which, as we'll learn in chapter 11, isn't necessarily the case).
- Mapping it to `/*` doesn't imply any specify type of response, but indicates that `DispatcherServlet` will serve *all* requests. That makes serving static content such as images and stylesheets more difficult than necessary.
- The `/app` pattern (or something similar) helps us distinguish `DispatcherServlet`-served content from other types of content. But then we have an implementation detail (specifically, the /app path) exposed in our URLs. That leads to complicated URL rewriting tactics to hide the /app path.

Rather than use any of those flawed servlet-mapping schemes, I prefer mapping `DispatcherServlet` like this:

```
<servlet-mapping>
    <servlet-name>spitter</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

By mapping `DispatcherServlet` to `/`, I'm saying that it's the default servlet and that it'll be responsible for handling all requests, including requests for static content.

If it concerns you that DispatcherServlet will be handling those kinds of requests, then hold on for a bit. A handy configuration trick frees you, the developer, from having to worry about that detail much. Spring's mvc namespace includes a new <mvc:resources> element that handles requests for static content for you. All you must do is configure it in the Spring configuration.

That means that it's now time to create the spitter-servlet.xml file that DispatcherServlet will use to create an application context. The following listing shows the beginnings of the spitter-servlet.xml file.

Listing 7.1 <mvc:resources> sets up a handler for serving static resources.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       xsi:schemaLocation="http://www.springframework.org/schema/mvc
         http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd
         http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

  <mvc:resources mapping="/resources/**"
                 location="/resources/" />

</beans>
```

Handle requests for static resources

As I said earlier, all requests that go through DispatcherServlet must be handled in some way, commonly via controllers. Since requests for static content are also being handled by DispatcherServlet, we're going to need some way to tell DispatcherServlet how to serve those resources. But writing and maintaining a controller for that purpose seems too involved. Fortunately, the <mvc:resources> element is on the job.[1]

<mvc:resources> sets up a handler for serving static content. The mapping attribute is set to /resources/**, which includes an Ant-style wildcard to indicate that the path must begin with */resources*, but may include any subpath thereof. The location attribute indicates the location of the files to be served. As configured here, any requests whose paths begin with /resources will be automatically served from the /resources folder at the root of the application. Therefore, all of our images, stylesheets, JavaScript, and other static content needs to be kept in the application's /resources folder.

Now that we've settled the issue of how static content will be served, we can start thinking about how our application's functionality can be served. Since we're just getting started, we'll start simple by developing the Spitter application's home page.

[1] The <mvc:resources> element was added in Spring 3.0.4. If you're using an older version of Spring, this facility won't be available.

7.2 Writing a basic controller

As we develop the web functionality for the Spitter application, we're going to develop resource-oriented controllers. Rather than write one controller for each use case in our application, we're going to write a single controller for each kind of resource that our application serves.

The Spitter application, being rather simple, has only two primary resource types: Spitters who are the users of the application and the spittles that they use to communicate their thoughts. Therefore, you'll need to write a spitter-oriented controller and a spittle-oriented controller. Figure 7.2 shows where these controllers fit into the overall application.

In addition to controllers for each of the application's core concepts, we also have two other utility controllers in 7.2. These controllers handle a few requests that are necessary, but don't directly map to a specific concept.

One of those controllers, `HomeController`, performs the necessary job of displaying the home page—a page that isn't directly associated with either `Spitters` or `Spittles`. That will be the first controller we write. But first, since we're developing annotation-driven controllers, there's a bit more setup to do.

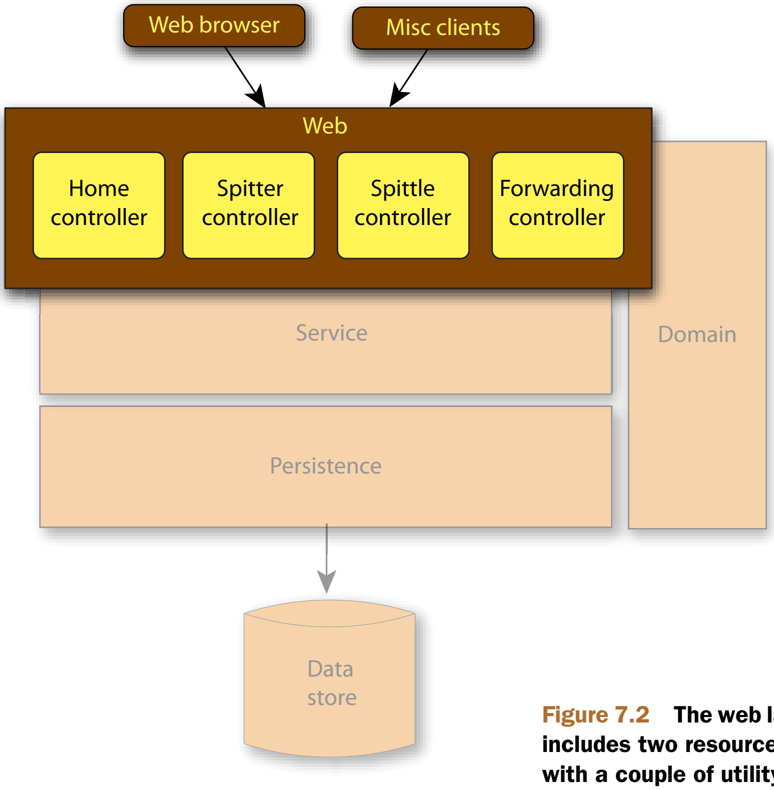


Figure 7.2 The web layer of the Spittle application includes two resource-oriented controllers along with a couple of utility controllers.

7.2.1 Configuring an annotation-driven Spring MVC

As I mentioned earlier, `DispatcherServlet` consults one or more handler mappings in order to know which controller to dispatch a request to. Spring comes with a handful of handler mapping implementations to choose from, including

- `BeanNameUrlHandlerMapping`—Maps controllers to URLs that are based on the controllers' bean names.
- `ControllerBeanNameHandlerMapping`—Similar to `BeanNameUrlHandlerMapping`, maps controllers to URLs that are based on the controllers' bean names. In this case, the bean names aren't required to follow URL conventions.
- `ControllerClassNameHandlerMapping`—Maps controllers to URLs by using the controllers' class names as the basis for their URLs.
- `DefaultAnnotationHandlerMapping`—Maps request to controller and controller methods that are annotated with `@RequestMapping`.
- `SimpleUrlHandlerMapping`—Maps controllers to URLs using a property collection defined in the Spring application context.

Using one of these handler mappings is usually just a matter of configuring it as a bean in Spring. But if no handler mapping beans are found, then `DispatcherServlet` creates and uses `BeanNameUrlHandlerMapping` and `DefaultAnnotationHandlerMapping`. Fortunately, we'll be working primarily with annotated controller classes, so the `DefaultAnnotationHandlerMapping` that `DispatcherServlet` gives us will do fine.

`DefaultAnnotationHandlerMapping` maps requests to controller methods that are annotated with `@RequestMapping` (which we'll see in the next section). But there's more to annotation-driven Spring MVC than just mapping requests to methods. As we build our controllers, we'll also use annotations to bind request parameters to handler method parameters, perform validation, and perform message conversion. Therefore, `DefaultAnnotationHandlerMapping` isn't enough.

Fortunately, you only need to add a single line of configuration to spitter-servlet.xml to flip on all of the annotation-driven features you'll need from Spring MVC:

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

Although small, the `<mvc:annotation-driven>` tag packs a punch. It registers several features, including JSR-303 validation support, message conversion, and support for field formatting.

We'll talk more about those features as we need them. For now, we have a home page controller to write.

7.2.2 Defining the home page controller

The home page is usually the first thing that visitors to a website will see. It's the front door to the rest of the site's functionality. In the case of the Spitter application, the home page's main job is to welcome visitors and to display a handful of recent spittles, hopefully enticing the visitors to join in on the conversation.

HomeController is a basic Spring MVC controller that handles requests for the home page.

Listing 7.2 HomeController welcomes the user to the Spitter application.

```
package com.habuma.spitter.mvc;
import javax.inject.Inject;
import java.util.Map;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import com.habuma.spitter.service.SpitterService;

@Controller
public class HomeController {
  public static final int DEFAULT_SPITTLES_PER_PAGE = 25;

  private SpitterService spitterService;

  @Inject
  public HomeController(SpitterService spitterService) {
    this.spitterService = spitterService;
  }

  @RequestMapping({"/","/home"})

  public String showHomePage(Map<String, Object> model) {

    model.put("spittles", spitterService.getRecentSpittles(
              DEFAULT_SPITTLES_PER_PAGE));

    return "home";
  }
}
```

Declare as controller

Inject SpitterService

Handle requests for home page

Place spittles into model

Return view name

Although HomeController is simple, there's a lot to talk about here. First, the @Controller annotation indicates that this class is a controller class. This annotation is a specialization of the @Component annotation, which means that <context:component-scan> will pick up and register @Controller-annotated classes as beans, just as if they were annotated with @Component.

That means that we need to configure a <context:component-scan> in spitter-servlet.xml so that the HomeController class (and all of the other controllers we'll write) will be automatically discovered and registered as beans. Here's the relevant snippet of XML:

```
<context:component-scan base-package="com.habuma.spitter.mvc" />
```

Going back to the HomeController class, we know that it'll need to retrieve a list of the most recent spittles via a SpitterService. Therefore, we've written the constructor to take a SpitterService as an argument and have annotated it with @Inject annotation so that it'll automatically be injected when the controller is instantiated.

The real work takes place in the showHomePage() method. As you can see, it's annotated with @RequestMapping. This annotation serves two purposes. First, it identifies showHomePage() as a request-handling method. And, more specifically, it specifies that this method should handle requests whose path is either / or /home.

As a request-handling method, showHomePage() takes a Map of String-to-Object as a parameter. This Map represents the model—the data that's passed between the controller and a view. After retrieving a list of recent Spittles from the SpitterService's getRecentSpittles() method, that list is placed into the model Map so that it can be displayed when the view is rendered.

As we write more controllers we'll see that the signature of a request-handling method can include almost anything as an argument. Even though showHomePage() only needed the model Map, we could've added HttpServletRequest, HttpServletResponse, String, or numeric parameters that correspond to query parameters in the request, cookie values, HTTP request header values, or a number of other possibilities. For now, though, the model Map is all we need.

The last thing that showHomePage() does is return a String value that's the logical name of the view that should render the results. A controller class shouldn't play a direct part in rendering the results to the client, but should only identify a view implementation that'll render the data to the client. After the controller has finished its work, DispatcherServlet will use this name to look up the actual view implementation by consulting a view resolver.

We'll configure a view resolver soon. But first let's write a quick unit test to assert that HomeController is doing what we expect it to do.

TESTING THE CONTROLLER

What's most remarkable about HomeController (and most Spring MVC controllers) is that there's little that's Spring-specific about it. In fact, if you were to strip away the three annotations, this would be a POJO.

From a unit testing perspective, this is significant because it means that HomeController can be tested easily without having to mock anything or create any Spring-specific objects. HomeControllerTest demonstrates how you might test HomeController.

Listing 7.3 A test to assert that the HomeController does its job correctly

```
package com.habuma.spitter.mvc;

import static com.habuma.spitter.mvc.HomeController.*;
import static java.util.Arrays.*;
import static org.junit.Assert.*;
import static org.mockito.Mockito.*;

import java.util.HashMap;
import java.util.List;

import org.junit.Test;

import com.habuma.spitter.domain.Spittle;
import com.habuma.spitter.service.SpitterService;

public class HomeControllerTest {
  @Test
  public void shouldDisplayRecentSpittles() {
    List<Spittle> expectedSpittles =
      asList(new Spittle(), new Spittle(), new Spittle());

    SpitterService spitterService = mock(SpitterService.class);

    when(spitterService.getRecentSpittles(DEFAULT_SPITTLES_PER_PAGE)).
        thenReturn(expectedSpittles);

    HomeController controller =
                   new HomeController(spitterService);

    HashMap<String, Object> model = new HashMap<String, Object>();
    String viewName = controller.showHomePage(model);

    assertEquals("home", viewName);

    assertSame(expectedSpittles, model.get("spittles"));
    verify(spitterService).getRecentSpittles(DEFAULT_SPITTLES_PER_PAGE);
  }
}
```

Mock SpitterService

Create controller

Call handler method

Assert results

The only thing that HomeController needs to do its job is an instance of SpitterService, which Mockito[2] graciously provides as a mock implementation. Once the mock SpitterService is ready, you just need to create a new instance of HomeController and then call the showHomePage() method. Finally, you assert that the list of spittles returned from the mock SpitterService ends up in the model Map under the spittles key and that the method returns a logical view name of home.

As you can see, testing a Spring MVC controller is like testing any other POJO in your Spring application. Even though it'll ultimately be used to serve a web page, we didn't have to do anything special or web-specific to test it.

At this point we've developed a controller to handle requests for the home page. And we've written a test to ensure that the controller does what we think it should. One question is still unanswered, though. The showHomePage() method returned a logical view name. But how does that view name end up being used to render output to the user?

7.2.3 *Resolving views*

The last thing that must be done in the course of handling a request is rendering output to the user. This job falls to some view implementation—typically JavaServer Pages (JSP), but other view technologies such as Velocity or FreeMarker may be used. In order to figure out which view should handle a given request, DispatcherServlet consults a view resolver to exchange the logical view name returned by a controller for an actual view that should render the results.

In reality, a view resolver's job is to map a logical view name to some implementation of org.springframework.web.servlet.View. But it's sufficient for now to think of a view resolver as something that maps a view name to a JSP, as that's effectively what it does.

[2] http://mockito.org

Spring comes with several view resolver implementations to choose from, as described in table 7.1.

Table 7.1 When it's time to present information to a user, Spring MVC can select an appropriate view using one of several view resolvers.

View resolver	Description
`BeanNameViewResolver`	Finds an implementation of `View` that's registered as a `<bean>` whose ID is the same as the logical view name.
`ContentNegotiatingViewResolver`	Delegates to one or more other view resolvers, the choice of which is based on the content type being requested. (We'll talk more about this view resolver in chapter 11.)
`FreeMarkerViewResolver`	Finds a FreeMarker-based template whose path is determined by prefixing and suffixing the logical view name.
`InternalResourceViewResolver`	Finds a view template contained within the web application's WAR file. The path to the view template is derived by prefixing and suffixing the logical view name.
`JasperReportsViewResolver`	Finds a view defined as a Jasper Reports report file whose path is derived by prefixing and suffixing the logical view name.
`ResourceBundleViewResolver`	Looks up `View` implementations from a properties file.
`TilesViewResolver`	Looks up a view that is defined as a Tiles template. The name of the template is the same as the logical view name.
`UrlBasedViewResolver`	This is the base class for some of the other view resolvers, such as `InternalResourceViewResolver`. It can be used on its own, but it's not as powerful as its subclasses. For example, `UrlBasedViewResolver` is unable to resolve views based on the current locale.
`VelocityLayoutViewResolver`	This is a subclass of `VelocityViewResolver` that supports page composition via Spring's `VelocityLayoutView` (a view implementation that emulates Velocity's `VelocityLayoutServlet`).
`VelocityViewResolver`	Resolves a Velocity-based view where the path of a Velocity template is derived by prefixing and suffixing the logical view name.
`XmlViewResolver`	Finds an implementation of `View` that's declared as a `<bean>` in an XML file (/WEB-INF/views.xml). This view resolver is a lot like `BeanNameViewResolver` except that the view `<bean>`s are declared separately from those for the application's Spring context.
`XsltViewResolver`	Resolves an XSLT-based view where the path of the XSLT stylesheet is derived by prefixing and suffixing the logical view name.

There's neither time nor space enough for me to cover all of these view resolvers. But a few of them are quite useful and worth a closer look. We'll start by looking at `InternalResolverViewResolver`.

RESOLVING INTERNAL VIEWS

A lot of Spring MVC embraces a convention-over-configuration approach to development. `InternalResourceViewResolver` is one such convention-oriented element. It resolves a logical view name into a `View` object that delegates rendering responsibility to a template (usually a JSP) located in the web application's context. As illustrated in figure 7.3, it does this by taking the logical view name and surrounding it with a prefix and a suffix to arrive at the path of a template that's a resource within the web application.

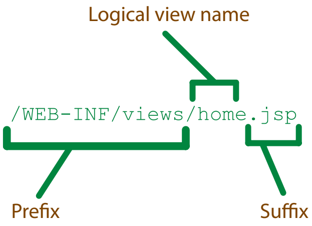


Figure 7.3 `InternalResourceViewResolver` resolves a view template's path by attaching a specified prefix and suffix to the logical view name.

Let's say that we've placed all of the JSPs for the Spitter application in the /WEB-INF/views/ directory. Given that arrangement, we'll need to configure an `InternalResourceViewResolver` bean in spitter-servlet.xml as follows:

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

When `DispatcherServlet` asks `InternalResourceViewResolver` to resolve a view, it takes the logical view name, prefixes it with /WEB-INF/views/ and suffixes it with .jsp. The result is the path of a JSP that will render the output. Internally, `InternalResourceViewResolver` then hands that path over to a `View` object that dispatches the request to the JSP. So, when `HomeController` returns *home* as the logical view name, it'll end up being resolved to the path /WEB-INF/views/home.jsp.

By default the `View` object that `InternalResourceViewResolver` creates is an instance of `InternalResourceView`, which simply dispatches the request to the JSP for rendering. But since home.jsp uses some JSTL tags, we may choose to replace `InternalResourceView` with `JstlView` by setting the `viewClass` property as follows:

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

`JstlView` dispatches the request to JSP, just like `InternalResourceView`. But it also exposes JSTL-specific request attributes so that you can take advantage of JSTL's internationalization support.

Although we won't delve into the details of FreeMarkerViewResolver, JasperReportsViewResolver, VelocityViewResolver, VelocityLayoutViewResolver, or XsltViewResolver, they're all similar to InternalResourceViewResolver in that they resolve views by adding a prefix and a suffix to the logical view name to find a view template. Once you know how to use InternalResourceViewResolver, working with those other view resolvers should feel natural.

Using InternalResourceViewResolver to resolve to JSP views is fine for a simple web application with an uncomplicated look and feel. But websites often have interesting user interfaces with some common elements shared between pages. For those kinds of sites, a layout manager such as Apache Tiles is in order. Let's see how to configure Spring MVC to resolve Tiles layout views.

RESOLVING TILES VIEWS

Apache Tiles[3] is a templating framework for laying out pieces of a page as fragments that are assembled into a full page at runtime. Although it was originally created as part of the Struts framework, Tiles proved to be useful with other web frameworks. In fact, we'll use it with Spring MVC to lay out the look and feel of the Spitter application.

To use Tiles views in Spring MVC, the first thing to do is to register Spring's TilesViewResolver as a <bean> in spitter-servlet.xml:

```
<bean class=
    "org.springframework.web.servlet.view.tiles2.TilesViewResolver"/>
```

This modest <bean> declaration sets up a view resolver that attempts to find views that are Tiles template definitions where the logical view name is the same as the Tiles definition name.

What's missing here is how Spring knows about Tiles definitions. By itself, TilesViewResolver doesn't know anything about any Tiles definitions, but instead relies on a TilesConfigurer to keep track of that information. So we'll need to add a TilesConfigurer bean to spitter-servlet.xml:

```
<bean class=
    "org.springframework.web.servlet.view.tiles2.TilesConfigurer">
  <property name="definitions">
    <list>
      <value>/WEB-INF/viewsviews.xml</value>
    </list>
  </property>
</bean>
```

TilesConfigurer loads one or more Tiles definition files and make them available for TilesViewResolver to resolve views from. For the Spitter application we're going to have a few Tiles definition files, all named views.xml, spread around under the /WEB-INF/views folder. So we wire /WEB-INF/views/**/views.xml into the definitions property. The Ant-style ** pattern indicates that the entire directory hierarchy under /WEB-INF/views should be searched for files named views.xml.

[3] http://tiles.apache.org

As for the contents of views.xml files, we'll build them up throughout this chapter, starting with just enough to render the home page. The following views.xml file defines the `home` tile definition as well as a common `template` definition to be used by other tile definitions.

Listing 7.4 Tiles defined

```
<!DOCTYPE tiles-definitions PUBLIC
      "-//Apache Software Foundation//DTD Tiles Configuration 2.1//EN"
      "http://tiles.apache.org/dtds/tiles-config_2_1.dtd">

<tiles-definitions>
   <definition name="template"
               template="/WEB-INF/views/main_template.jsp">
     <put-attribute name="top"
                    value="/WEB-INF/views/tiles/spittleForm.jsp" />
     <put-attribute name="side"
                    value="/WEB-INF/views/tiles/signinsignup.jsp" />
   </definition>

   <definition name="home" extends="template">

     <put-attribute name="content" value="/WEB-INF/views/home.jsp" />
   </definition>
</tiles-definitions>
```

Define common layout

Define home tile

The `home` definition extends the `template` definition, using home.jsp as the JSP that renders the main content of the page, but relying on `template` for all of the common features of the page.

It's the `home` template that `TilesViewResolver` will find when it tries to resolve the logical view name returned by `HomeController`'s `showHomePage()` methods. `DispatcherServlet` will send the request to Tiles to render the results using the `home` definition.

7.2.4 Defining the home page view

As you can see from listing 7.4, the home page is made up of several distinct pieces. The main_template.jsp file describes the common layout for all pages in the Spitter application, while home.jsp displays the main content for the home page. Plus, spittleForm.jsp and signinsignup.jsp provide some additional common elements.

For now we'll focus on home.jsp, as it's most pertinent to our discussion of displaying the home page. This JSP is where the home page request finishes its journey. It picks up the list of `Spittles` that `HomeController` placed into the model and renders them to be displayed in the user's browser. The following shows what home.jsp is made of.

Listing 7.5 The home page `<div>` element will be inserted into the template.

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="s" uri="http://www.springframework.org/tags"%>
<%@ taglib prefix="t" uri="http://tiles.apache.org/tags-tiles"%>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt"%>

<div>
  <h2>A global community of friends and strangers spitting out their
  inner-most and personal thoughts on the web for everyone else to
  see.</h2>
  <h3>Look at what these people are spitting right now...</h3>

  <ol class="spittle-list">
    <c:forEach var="spittle" items="${spittles}">

      <s:url value="/spitters/{spitterName}"
                  var="spitter_url" >
        <s:param name="spitterName"
                      value="${spittle.spitter.username}" />
      </s:url>

      <li>
        <span class="spittleListImage">
          <img src=
            "http://s3.amazonaws.com/spitterImages/${spittle.spitter.id}.jpg"
             width="48"
             border="0"
             align="middle"
             onError=
      "this.src='<s:url value="/resources/images"/>/spitter_avatar.png';"/>
      </span>
      <span class="spittleListText">
        <a href="${spitter_url}">
          <c:out value="${spittle.spitter.username}" /></a>
          - <c:out value="${spittle.text}" /><br/>
         <small><fmt:formatDate value="${spittle.when}"
                                pattern="hh:mma MMM d, yyyy" /></small>
      </span>
      </li>
    </c:forEach>
  </ol>
</div>
```

Iterate over list of Spittles

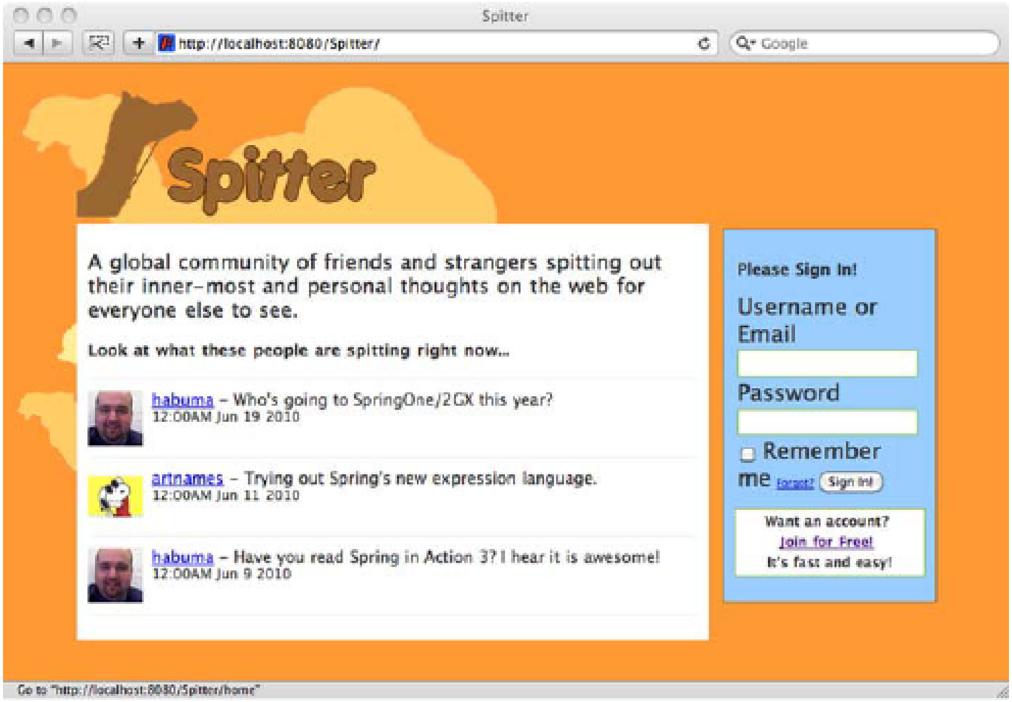
Construct context-relative Spitter URL

Display Spitter properties

Aside from a few friendly messages at the beginning, the crux of home.jsp is contained in the `<c:forEach>` tag, which cycles through the list of `Spittles`, rendering the details of each one as it goes. Since the `Spittles` were placed into the model with the key `spittles`, the list is referenced in the JSP using `${spittles}`.

> **THE MODEL AND REQUEST ATTRIBUTES: THE INSIDE STORY** It's not obvious, but `${spittles}` in home.jsp refers to a servlet request attribute named `spittles`. After `HomeController` finished its work and before home.jsp was called into action, `DispatcherServlet` copied all of the members of the model into request attributes with the same name.

Take notice of the `<s:url>` tag near the middle. We use this tag to create a servlet context–relative URL to the `Spitter` that authored each `Spittle`. The `<s:url>` tag is new to Spring 3.0 and works much like JSTL's `<c:url>` tag.



A global community of friends and strangers spitting out their inner-most and personal thoughts on the web for everyone else to see.

Look at what these people are spitting right now...

habuma – Who's going to SpringOne/2GX this year?
12:00AM Jun 19 2010

artnames – Trying out Spring's new expression language.
12:00AM Jun 11 2010

habuma – Have you read Spring in Action 3? I hear it is awesome!
12:00AM Jun 9 2010

Want an account?
Join for Free!
It's fast and easy!

Figure 7.4 The Spitter application's home page displays a welcome message along with a list of recent spittles.

The main difference between Spring's `<s:url>` and JSTL's `<c:url>` is that `<s:url>` supports parameterized URL paths. In this case, the path is parameterized with the `Spitter`'s username. For example, if the `Spitter`'s username is *habuma* and the servlet context name is *Spitter*, then the resulting path will be /Spitter/spitters/habuma.

When rendered, this JSP along with the other JSPs in the same Tiles definition will display the Spitter application's home page, as shown in figure 7.4.

At this point, we've written our first Spring MVC controller, configured a view resolver, and defined a basic JSP view to display the results of invoking the controller. There's one tiny problem, though. An exception is waiting to happen in `HomeController` because `DispatcherServlet`'s Spring application context won't know where to find a `SpitterService` bean. Fortunately, it's an easy fix.

7.2.5 Rounding out the Spring application context

As I mentioned earlier, `DispatcherServlet` loads its Spring application context from a single XML file whose name is based on its `<servlet-name>`. But what about the other beans we've declared in previous chapters, such as the `SpitterService` bean? If `DispatcherServlet` is going to load its beans from a file named spitter-servlet.xml, then won't we need to declare those other beans in spitter-servlet.xml?

In the earlier chapters we've split our Spring configuration across multiple XML files: one for the service layer, one for the persistence layer, and another for the data source configuration. Although not strictly required, it's a good idea to organize our

Spring configuration across multiple files. With that in mind, it makes sense to put all of the web layer configuration in spitter-servlet.xml, the file loaded by `DispatcherServlet`. But we still need a way to load the other configuration files.

That's where `ContextLoaderListener` comes into play. `ContextLoaderListener` is a servlet listener that loads additional configuration into a Spring application context alongside the application context created by `DispatcherServlet`. To use `ContextLoaderListener`, add the following `<listener>` declaration to the web.xml file:

```
<listener>
    <listener-class>
        org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
```

We also need to tell `ContextLoaderListener` which Spring configuration file(s) it should load. If not specified otherwise, the context loader will look for a Spring configuration file at /WEB-INF/applicationContext.xml. But this single file doesn't lend itself to breaking up the application context into several pieces. So we'll need to override this default.

To specify one or more Spring configuration files for `ContextLoaderListener` to load, set the `contextConfigLocation` parameter in the servlet context:

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        /WEB-INF/spitter-security.xml
        classpath:service-context.xml
        classpath:persistence-context.xml
        classpath:dataSource-context.xml
    </param-value>
</context-param>
```

The `contextConfigLocation` parameter is specified as a list of paths. Unless specified otherwise, the paths are relative to the application root. But since our Spring configuration is split across multiple XML files that are scattered across several JAR files in the web application, we've prefixed some of them with `classpath:` to load them as resources from the application classpath and others with a path local to the web application.

You'll recognize that we've included the Spring configuration files that we created in previous chapters. You may also notice a few extra configuration files that we've not covered yet. Don't worry...we'll get to those in later chapters.

Now we have our first controller written and ready to serve requests for the Spitter application's home page. If all we needed is a home page, we'd be done. But there's more to Spitter than just the home page, so let's continue building out the application. The next thing we'll try is to write a controller that can handle input.

7.3 Handling controller input

HomeController had it easy. It didn't have to deal with user input or any parameters. It just handled a basic request and populated the model for the view to render. It couldn't have been much simpler.

But not all controllers live such simple lives. Controllers are often asked to perform some logic against one or more pieces of information that are passed in as URL parameters or as form data. Such is the case for both SpitterController and SpittleController. These two controllers will handle several kinds of requests, many of which take input of some kind.

One example of how SpitterController will handle input is in how it supports displaying a list of Spittles for a given Spitter. Let's drive out that functionality now to see how to write controllers that process input.

7.3.1 Writing a controller that processes input

One way that we could implement SpitterController is to have it respond to a URL with the Spitter's username as a request query parameter. For example, http://localhost:8080/spitter/spitters/spittles?spitter=habuma could be the URL for displaying all of the Spittles for a Spitter whose username is habuma.

The following shows an implementation of SpitterController that can respond to this kind of request.

Listing 7.6 A conventional approach to handling requests for a Spitter's spittles

```
package com.habuma.spitter.mvc;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RequestMapping;
import com.habuma.spitter.domain.Spitter;
import com.habuma.spitter.service.SpitterService;
import static org.springframework.web.bind.annotation.RequestMethod.*;

@Controller
@RequestMapping("/spitter")                                    Root URL path
public class SpitterController {

  private final SpitterService spitterService;

  @Inject
  public SpitterController(SpitterService spitterService) {
    this.spitterService = spitterService;
  }                                                              Handle GET requests
                                                                 for /spitter/spittles
  @RequestMapping(value="/spittles", method=GET)
  public String listSpittlesForSpitter(
        @RequestParam("spitter") String username, Model model) {
    Spitter spitter = spitterService.getSpitter(username);
    model.addAttribute(spitter);                                    Fill model
    model.addAttribute(spitterService.getSpittlesForSpitter(username));
    return "spittles/list";
  }
}
```

As you can see, we've annotated SpitterController with @Controller and @RequestMapping at the class level. As we've already discussed, @Controller is a clue to <context:component-scan> that this class should be automatically discovered and registered as a bean in the Spring application context.

You'll also notice that SpitterController is annotated with @RequestMapping at the class level. In HomeController we used @RequestMapping on the showHomePage() handler method, but this class-level use of @RequestMapping is different.

As used here, the class-level @RequestMapping defines the root URL path that this controller will handle. We'll ultimately have several handler methods in SpitterController, each handling different types of requests. But here @RequestMapping is saying that all of those requests will have paths that start with /spitters.

Within SpitterController we currently have a single method: listSpittlesForSpitter(). Like any good handler method, this method is annotated with @RequestMapping. It's not dramatically different from the one we used in HomeController. But there's more to this @RequestMapping than meets the eye.

Method-level @RequestMappings narrow the mapping defined by any class-level @RequestMapping. Here, SpitterController is mapped to /spitters at the class level and to /spittles at the method level. Taken together, that means that listSpittlesForSpitter() handles requests for /spitters/spittles. Moreover, the method attribute is set to GET indicating that this method will only handle HTTP GET requests for /spitters/spittles.

The listSpittlesForSpitter() method takes a Stringusername and a Model object as parameters.

The username parameter is annotated with @RequestParam("spitter") to indicate that it should be given the value of the spitter query parameter in the request. listSpittlesForSpitter() will use that parameter to look up the Spitter object and its list of Spittles.

You're probably scratching your head over the second parameter to

Do I really need @RequestParam?

The @RequestParam annotation isn't strictly required. @RequestParam is useful for binding query parameters to method parameters where the names don't match. As a matter of convention, any parameters of a handler method that aren't annotated otherwise will be bound to the query parameter of the same name. In the case of listSpittlesForSpitter(), if the parameter were named *spitter* or if the query parameter were called *username*, then we could leave the @RequestParam annotation off.

@RequestParam also comes in handy when you compile your Java code without debugging information compiled in. In that circumstance, the name of the method parameter is lost and so there's no way to bind the query parameter to the method parameter by convention. For that reason, it's probably best to always use @RequestParam and not rely too heavily on the convention.

the `listSpittlesForSpitter()` method. When we wrote `HomeController`, we passed in a `Map<String, Object>` to represent the model. But here we're using a new `Model` parameter.

The truth be known, the object passed in as a `Model` likely is a `Map<String, Object>` under the covers. But `Model` provides a few convenient methods for populating the model, such as `addAttribute()`. The `addAttribute()` method does pretty much the same thing as `Map`'s `put()` method, except that it figures out the key portion of the map on its own.

When adding a `Spitter` object to the model, `addAttribute()` gives it the name `spitter`, a name it arrives at by applying JavaBeans property naming rules to the object's class name. When adding a `List` of `Spittles`, it tacks *List* to the end the member type of the `List`, naming the attribute `spittleList`.

We're almost ready to call `listSpittlesForSpitter()` done. We've written the `SpitterController` and a handler method. All that's left is to write the view that will display that list of `Spittles`.

7.3.2 Rendering the view

When the list of `Spittles` is displayed to the user, we don't need much different than what we did for the home page. We just need to show the name of the `Spitter` (so that it's clear whom the list of `Spittles` belongs to) and then list each `Spittle`

To enable that, we first need to create a new Tiles definition. `listSpittlesForSpitter()` returns `spittles/list` as its logical view name, so the following Tile definition should do the trick:

```
<definition name="spittles/list" extends="template">
 <put-attribute name="content"
                value="/WEB-INF/views/spittles/list.jsp" />
</definition>
```

Just like the `home` Tile, this one adds another JSP page to the `content` attribute to be rendered within main_template.jsp. The list.jsp file used to display the list of `Spittles` is shown next.

Listing 7.7 The list.jsp file is a JSP that's used to display a list of `Spittle` objects.

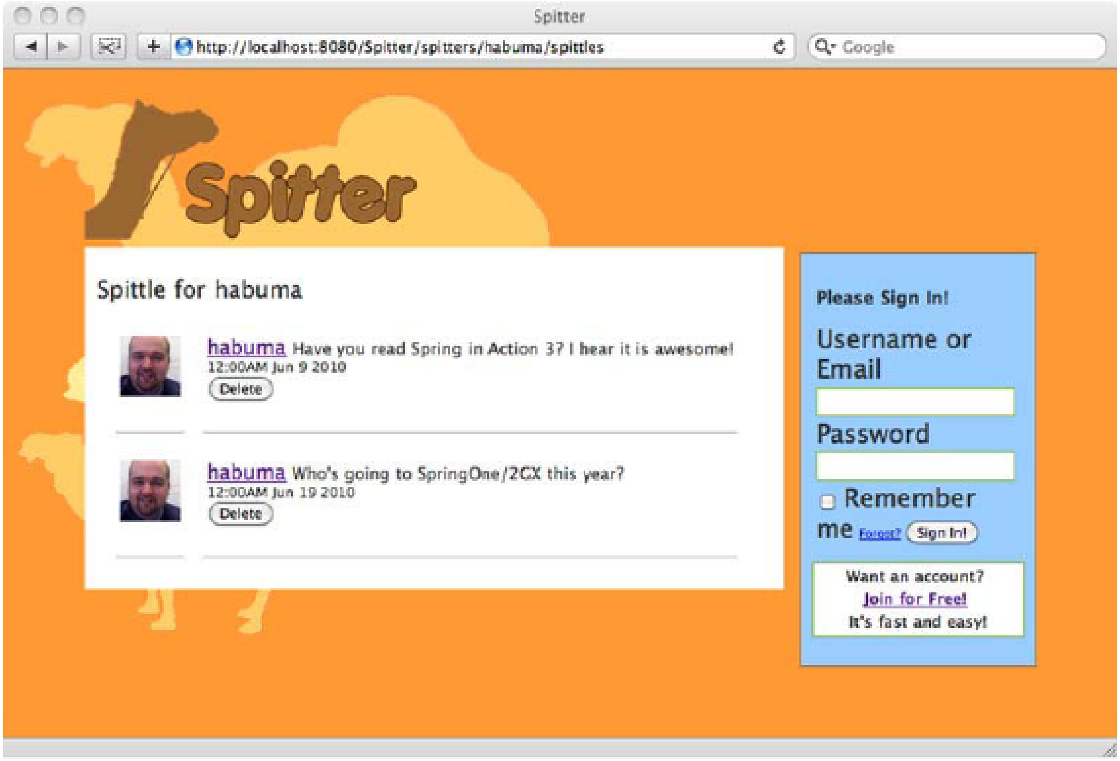
```
<%@ taglib prefix="s" uri="http://www.springframework.org/tags"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>

<div>
  <h2>Spittles for ${spitter.username}</h2>              ◁—— Display username

  <table cellspacing="15">
    <c:forEach items="${spittleList}" var="spittle">          ◁—— List Spittles

    <tr>
      <td>
          <img src="<s:url value="/resources/images/spitter_avatar.png"/>"
               width="48" height="48" /></td>
      <td>
          <a href="<s:url value="/spitters/${spittle.spitter.username}"/>">
                       ${spittle.spitter.username}</a>
          <c:out value="${spittle.text}" /><br/>
          <c:out value="${spittle.when}" />
      </td>
    </tr>
    </c:forEach>
  </table>
</div>
```

Aesthetics aside, this JSP does what we need. Near the top, it displays a header indicating who the list of `Spittles` belongs to. This header references the `username` property of the `Spitter` object that `listSpittlesForSpitter()` placed into the model with `${spitter.username}`.

The better part of this JSP iterates through the list of `Spittles`, displaying their details. The JSTL `<c:forEach>` tag's `items` attribute references the list with `${spittleList}`—the name that `Model`'s `addAttribute()` gave it.

One minor thing to take note of is that we're using a hardcoded reference to spitter_avatar.png as the user's profile image. In section 7.5 we'll see how to let the user upload an image to their profile.

The result of list.jsp, as rendered in the context of the spittles/list view, is shown in figure 7.5.



Spittle for habuma

habuma Have you read Spring in Action 3? I hear it is awesome!
12:00AM Jun 9 2010

habuma Who's going to SpringOne/2GX this year?
12:00AM Jun 19 2010

Figure 7.5 When taken together with other Tiles elements, list.jsp shows a list of spittles for a given user.

But first, we need to create a way for users to register to the application. In doing so, we'll get a chance to write a controller that handles form submissions.

7.4 Processing forms

Working with forms in a web application involves two operations: displaying the form and processing the form submission. Therefore, in order to register a new `Spitter` in our application, we're going to need to add two handler methods to `SpitterController` to handle each of the operations. Since we're going to need the form in the browser before we can submit it, we'll start by writing the handler method that displays the registration form.

7.4.1 Displaying the registration form

When the form is displayed, it'll need a `Spitter` object to bind to the form fields. Since this is a new `Spitter` that we're creating, a newly constructed, uninitialized `Spitter` object will be perfect. The following `createSpitterProfile()` handler method will create a `Spitter` object and place it in the model.

Listing 7.8 Displaying the form for registering a spitter

```
@RequestMapping(method=RequestMethod.GET, params="new")
public String createSpitterProfile(Model model) {
  model.addAttribute(new Spitter());
  return "spitters/edit";
}
```

As with other handler methods, `createSpitterProfile()` is annotated with `@RequestMapping`. But, unlike previous handler methods, this one doesn't specify a path. Therefore, this method handles requests for the path specified in the class-level `@RequestMapping`—`/spitters` in the case of `SpitterController`.

What the method's `@RequestMapping` does specify is that this method will handle HTTP GET requests only. What's more, note the `params` attribute, which is set to `new`. This means that this method will only handle HTTP GET requests for /spitters if the request includes a `new` query parameter. Figure 7.6 illustrates the kind of URL that `createSpitterProfile` will handle.

As for the inner workings of `createSpitterProfile()`, it simply creates a new instance of a `Spitter` and adds it to the model. It then wraps up by returning spitters/edit as the logical name of the view that will render the form.

Speaking of the view, let's create it next.

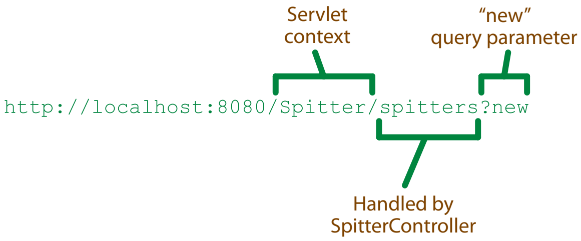

Servlet context
"new" query parameter
http://localhost:8080/Spitter/spitters?new
Handled by SpitterController

Figure 7.6 `@RequestMapping`'s `params` attribute can limit a handler method to only handling requests with certain parameters.

DEFINING THE FORM VIEW

As before, the logical view name returned from createSpitterProfile() will ultimately be mapped to a Tiles definition for rendering the form to the user. So we need to add a tile definition named spitters/edit to the Tiles configuration file. The following <definition> entry should do the trick.

```
<definition name="spitters/edit" extends="template">
 <put-attribute name="content"
                value="/WEB-INF/views/spitters/edit.jsp" />
</definition>
```

As before, the content attribute is where the main content of the page will go. In this case, it's the JSP file at /WEB-INF/views/spitters/edit.jsp, as shown next.

Listing 7.9 Rendering a form to capture user registration information

```
<%@ taglib prefix="sf" uri="http://www.springframework.org/tags/form"%>

<div>
<h2>Create a free Spitter account</h2>

<sf:form method="POST" modelAttribute="spitter">          Bind form to model attribute
   <fieldset>
   <table cellspacing="0">
      <tr>
         <th><label for="user_full_name">Full name:</label></th>
         <td><sf:input path="fullName" size="15" id="user_full_name"/></td>
      </tr>
      <tr>
         <th><label for="user_screen_name">Username:</label></th>
         <td><sf:input path="username" size="15" maxlength="15"
             id="user_screen_name"/>                       Username field
              <small id="username_msg">No spaces, please.</small>

            </td>
      </tr>
      <tr>
         <th><label for="user_password">Password:</label></th>
         <td><sf:password path="password" size="30"
                            showPassword="true"
                            id="user_password"/>          Password field
             <small>6 characters or more (be tricky!)</small>
            </td>
      </tr>

      <tr>
         <th><label for="user_email">Email Address:</label></th>

         <td><sf:input path="email" size="30"
                 id="user_email"/>                     Email field
             <small>In case you forget something</small>
            </td>
      </tr>
      <tr>
```

```
         <th></th>
         <td>
            <sf:checkbox path="updateByEmail"
                             id="user_send_email_newsletter"/>     Update-by-email
            <label for="user_send_email_newsletter"                checkbox
            >Send me email updates!</label>

         </td>
      </tr>
   </table>
</fieldset>
</sf:form>
</div>
```

What makes this JSP file different than the others we've created so far is that it uses Spring's form binding library. The `<sf:form>` tag binds the `Spitter` object (identified by the `modelAttribute` attribute) that `createSpitterProfile()` placed into the model to the various fields in the form.

The `<sf:input>`, `<sf:password>`, and `<sf:checkbox>` tags each have a `path` attribute that references the property of the `Spitter` object that the form is bound to. When the form is submitted, whatever values these fields contain will be placed into a `Spitter` object and submitted to the server for processing.

Note that the `<sf:form>` specifies that it'll be submitted as an HTTP POST request. What it doesn't specify is the URL. With no URL specified, it'll be submitted back to /spitters, the same URL path that displayed the form. That means that the next thing to do is to write another handler method that accepts POST requests for /spitters.

7.4.2 Processing form input

After the form is submitted, we'll need a handler method that takes a `Spitter` object (populated with data from the form) and saves it. Then, the last thing it should do is redirect to the user's profile page. The following listing shows `addSpitterFromForm()`, a method that processes the form submission.

Listing 7.10 The `addSpitter` method processes input from the spitter form.

```
@RequestMapping(method=RequestMethod.POST)
public String addSpitterFromForm(@Valid Spitter spitter,
                                 BindingResult bindingResult) {
  if(bindingResult.hasErrors()) {                              Check for errors
    return "spitters/edit";
  }
                                                                          Redirect
  spitterService.saveSpitter(spitter);              Save Spitter          after
                                                                          POST
  return "redirect:/spitters/" + spitter.getUsername();
}
```

Note that the `addSpitterFromForm()` method is annotated with an `@RequestMapping` annotation that isn't much different than the `@RequestMapping` that adorns the `createSpitterProfile()` method. Neither specify a URL path, meaning that both handle requests for /spitters. The difference is that where createSpitterProfile() handles GET requests, addSpitterFromForm() handles POST requests. That's perfect, since that's how the form will be submitted.

And when that form is submitted, the fields in the request will be bound to the Spitter object that's passed in as an argument to addSpitterFromForm(). From there, it's sent to the SpitterService's saveSpitter() method to be stored away in the database.

You may have also noticed that the Spitter parameter is annotated with @Valid. This indicates that the Spitter should pass validation before being passed in. We'll talk about validation in the next section.

Like the handler methods we've written before, this one ends by returning a String to indicate where the request should be sent next. This time, instead of specifying a logical view name, we're returning a special redirect view. The redirect: prefix signals that the request should be redirected to the path that it precedes. By redirecting to another page, we can avoid duplicate submission of the form if the user clicks the Refresh button in their browser.

As for the path that it's redirecting to, it'll take the form of /spitters/{username} where *{username}* represents the username of the Spitter that was just submitted. For example, if the user registered under the name habuma, then they'd be redirected to /spitters/habuma after the form submission.

HANDLING REQUESTS WITH PATH VARIABLES

The big question is what will respond to requests for /spitters/{username}? Actually, that's another handler method that we'll add to SpitterController:

```
@RequestMapping(value="/{username}", method=RequestMethod.GET)
public String showSpitterProfile(@PathVariable String username,
        Model model) {
  model.addAttribute(spitterService.getSpitter(username));
  return "spitters/view";
}
```

The showSpitterProfile() method isn't too dissimilar from the other handler methods we've seen. It's given a String parameter containing a username and uses it to retrieve a Spitter object. It then places that Spitter into the model and wraps up by returning the logical name of the view that will render the output.

But by now you've probably noticed a few things that make showSpitterProfile() different. First, the value attribute in the @RequestMapping contains some strange-looking curly braces. And the username parameter is annotated with @PathVariable.

Those two things work together to enable the showSpitterProfile() method to handle requests whose URLs have parameters embedded in their path. The {username} portion of the path is actually a placeholder that corresponds to the username method parameter that's annotated with @PathVariable. Whatever value is in that location in a request's path will be passed in as the value of username.

For example, if the request path is /username/habuma, then habuma will be passed in to showSpitterProfile() for the username.

We'll talk more about `@PathVariable` and how it helps us write handler methods that respond to RESTful URLs when we get to chapter 11.

But we still have some unfinished business with regard to `addSpitterFromForm()`. You've probably noticed that `addSpitterFromForm()`'s `Spitter` parameter is annotated with `@Valid`. Let's see how this annotation can be used to keep bad data from being submitted in a form.

7.4.3 Validating input

When a user registers with the Spitter application, there are certain requirements that we'd like to place on that registration. Specifically, a new user must give us their full name, email address, a username, and a password. Not only that, but the email address can't be just freeform text—it must look like an email address. Moreover, the password should be at least six characters long.

The `@Valid` annotation is the first line of defense against faulty form input. `@Valid` is actually a part of the JavaBean validation specification.[4] Spring 3 includes support for JSR-303, and we're using `@Valid` here to tell Spring that the `Spitter` object should be validated as it's bound to the form input.

Should anything go wrong while validating the `Spitter` object, the validation error will be carried to the `addSpitterFromForm()` method via the `BindingResult` that's passed in on the second parameter. If the `BindingResult`'s `hasErrors()` method returns `true`, then that means that validation failed. In that case, the method will return `spitters/edit` as the view name to display the form again so that the user can correct any validation errors.

But how will Spring know the difference between a valid `Spitter` and an invalid `Spitter`?

DECLARING VALIDATION RULES

Among other things, JSR-303 defines a handful of annotations that can be placed on properties to specify validation rules. We can use these annotations to define what "valid" means with regard to a `Spitter` object. The following shows the properties of the `Spitter` class that are annotated with validation annotations.

Listing 7.11 Annotating a `Spitter` for validation

```
@Size(min=3, max=20, message=
    "Username must be between 3 and 20 characters long.")

@Pattern(regexp="^[a-zA-Z0-9]+$",
        message="Username must be alphanumeric with no spaces")
private String username;

@Size(min=6, max=20,
        message="The password must be at least 6 characters long.")
private String password;
```

Enforce size

Ensure no spaces

[4] Also known as JSR-303 (http://jcp.org/en/jsr/summary?id=303)

```
@Size(min=3, max=50, message=
    "Your full name must be between 3 and 50 characters long.")   Enforce
private String fullName;                                           size

@Pattern(regexp="[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}",
        message="Invalid email address.")
private String email;                                 Match email pattern
```

The first three properties in listing 7.11 are annotated with JSR-303's `@Size` annotation to validate that those fields meet certain expectations on their length. The `username` property must be at least 3 and at most 20 characters long, whereas the `fullName` property must be between 3 and 50 characters in length. As for the `password` property, it must be at least 6 characters long and not exceed 20 characters.

To make sure that the value given to the email property fits the format of an email address, we've annotated it with `@Pattern` and specified a regular expression to match it against in the `regexp` attribute.[5] Similarly, we've used `@Pattern` on the `username` property to ensure that the username is only made up of alphanumeric characters with no spaces.

In all of the validation annotations, we've set the `message` attribute with the message to be displayed in the form when validation fails so that the user knows what needs to be corrected.

With these annotations in place, when a user submits a registration form to `SpitterController`'s `addSpitterFromForm()` method, the values in the `Spitter` object's fields will be weighed against the validation annotations. If any of those rules are broken, then the handler method will send the user back to the form to fix the problem.

When they arrive back at the form, we'll need a way to tell them what the problem was. So we're going to have to go back to the form JSP and add some code to display the validation messages.

DISPLAYING VALIDATION ERRORS

Recall that the `BindingResult` passed in as a parameter to `addSpitterFromForm()` knew whether the form had any validation errors. And we were able to ask if there were any errors by calling its `hasErrors()` method. But what we didn't see was that the actual error messages are also in there, associated with the fields that failed validation.

One way of displaying those errors to the users is to access those field errors through `BindingResult`'s `getFieldError()` method. But a much better way is to use Spring's form binding JSP tag library to display the errors. More specifically, the `<sf:errors>` tag can render field validation errors. All we need to do is sprinkle a few `<sf:errors>` tags around our form JSP.

[5] Trust me…that gobbledygook will validate an email address.

Listing 7.12 The `<sf:errors>` JSP tag can be used to display form validation errors.

```
<%@ taglib prefix="sf" uri="http://www.springframework.org/tags/form"%>

<div>
<h2>Create a free Spitter account</h2>

<sf:form method="POST" modelAttribute="spitter"
         enctype="multipart/form-data">
   <fieldset>
   <table cellspacing="0">
      <tr>
         <th><sf:label path="fullName">Full name:</sf:label></th>
         <td><sf:input path="fullName" size="15" /><br/>
             <sf:errors path="fullName" cssClass="error" />
         </td>
      </tr>
      <tr>
         <th><sf:label path="username">Username:</sf:label></th>
         <td><sf:input path="username" size="15" maxlength="15" />
              <small id="username_msg">No spaces, please.</small><br/>
             <sf:errors path="username" cssClass="error" />
            </td>
      </tr>
      <tr>
         <th><sf:label path="password">Password:</sf:label></th>
         <td><sf:password path="password" size="30"
                            showPassword="true"/>
             <small>6 characters or more (be tricky!)</small><br/>
             <sf:errors path="password" cssClass="error" />

            </td>
      </tr>
      <tr>
         <th><sf:label path="email">Email Address:</sf:label></th>
         <td><sf:input path="email" size="30"/>
             <small>In case you forget something</small><br/>
             <sf:errors path="email" cssClass="error" />
            </td>
      </tr>
      <tr>
         <th></th>
         <td>
            <sf:checkbox path="updateByEmail"/>
            <sf:label path="updateByEmail"
            >Send me email updates!</sf:label>

         </td>
      </tr>
      <tr>
        <th><label for="image">Profile image:</label></th>
        <td><input name="image" type="file"/>
      </tr>
      <tr>
         <th></th>
         <td><input name="commit" type="submit"
```

Display fullName errors

Display username errors

Display password errors

Display email errors

```
                              value="I accept. Create my account." /></td>
            </tr>
         </table>
      </fieldset>
      </sf:form>
      </div>
```

The `<sf:errors>` tag's `path` attribute specifies the form field for which errors should be displayed. For example, the following `<sf:errors>` displays errors (if there are any) for the field whose name is `fullName`:

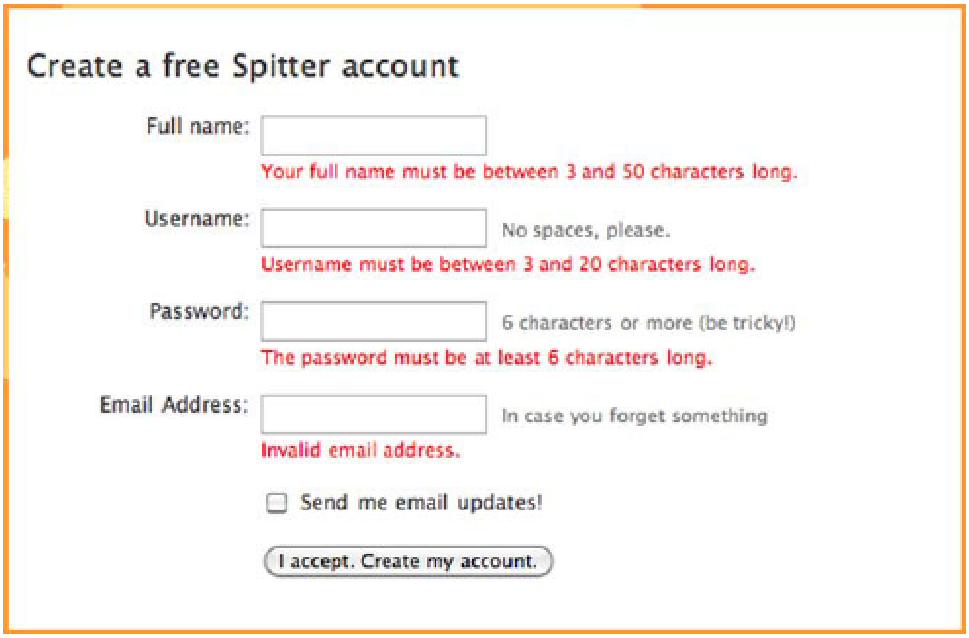
```
<sf:errors path="fullName" cssClass="error" />
```

If there are multiple errors for a single field, they'll all be displayed, separated by an HTML `<br/>` tag. If you'd rather have them separated some other way, then you can use the `delimiter` attribute. The following `<sf:errors>` snippet uses `delimiter` to separate errors with a comma and a space:

```
<sf:errors path="fullName" delimiter=", "
    cssClass="error" />
```

Note that there are four `<sf:errors>` tags in this JSP, one on each of the fields for which we declared validation rules. The `cssClass` attribute refers to a class that's declared in CSS to display in red so that it catches the user's attention.

With these in place, errors will be displayed on the page if any validation errors occur. For example, figure 7.7 shows what the form would look like if the user were to submit the form without filling in any of the fields.



Create a free Spitter account

Full name:
Your full name must be between 3 and 50 characters long.

Username: No spaces, please.
Username must be between 3 and 20 characters long.

Password: 6 characters or more (be tricky!)
The password must be at least 6 characters long.

Email Address: In case you forget something
Invalid email address.

□ Send me email updates!

Figure 7.7 With the `<sf:errors>` JSP tag on the registration page, validation problems will be shown to the user for them to fix and try again.

As you can see, validation errors are displayed on a per-field basis. But if you'd prefer to display all of the errors in one place (perhaps at the top of the form), you'll only need a single `<sf:errors>` tag, with its `path` attribute set to `*`:

```
<sf:errors path="*" cssClass="error" />
```

Now you know how to write controller handler methods that process form data. The one thing that's common about all of the form fields we've seen thus far is that they're textual data and were probably entered into the form by the user typing on a keyboard. But what if the thing that the user needs to submit in a form can't be banged out on a keyboard? What if the user needs to submit an image or some other kind of file?

7.5 Handling file uploads

Earlier, in section 7.2.4, I punted on how the user's profile picture would be displayed, simply displaying a default spitter_avatar.png for all users. But real Spitter users will want more identity than some generic icon. To grant them more individuality, we'll let them upload their profile photos as part of registration.

To enable file uploads in the Spitter application, we'll need to do three things:

- Add a file upload field to the registration form
- Tweak `SpitterController`'s `addSpitterFromForm()` to receive the uploaded file
- Configure a multipart file resolver in Spring

Let's start at the top of the list and prepare the registration form JSP to take a file upload.

7.5.1 Adding a file upload field to the form

Most form fields are textual and can easily be submitted to the server as a set of name-value pairs. In fact, a typical form submission has a content type of `application/x-www-form-urlencoded` and takes the form of name-value pairs separated by ampersands.

But I think you'll agree that files are a different kind of thing than most other field values that will be submitted in a form. Uploaded files are typically binary files and don't fit well into the name-value pair paradigm. Therefore, if we're going to let users upload images to be associated with their profile, we'll need to encode the form submission in some other way.

When it comes to submitting forms with files in tow, `multipart/form-data` is the content type of choice. We'll need to configure the form to submit as `multipart/form-data` content type by setting the `<sf:form>`'s `enctype` attribute as follows:

```
<sf:form method="POST"
         modelAttribute="spitter"
         enctype="multipart/form-data">
```

With enctype set to multipart/form-data, each field will be submitted as a distinct part of the POST request and not as just another name-value pair. This makes it possible for one of those parts to contain uploaded image file data.

Now we can add a new field to the form. A standard HTML <input> field with type set to file will do the trick:

```
<tr>
  <th><label for="image">Profile image:</label></th>
  <td><input name="image" type="file"/>
</tr>
```

This bit of HTML will render a basic file selection field on the form. Most browsers display this as a text field with a button to the side. Figure 7.8 shows what it looks like when rendered in the Safari browser on Mac OS X.

All of the parts of the form are in place for our users to submit a profile photo. When the form is submitted, it'll be posted as a multipart form where one of the parts contains the image file's binary data. Now we need to ready the server side of our application to be able to receive that data.

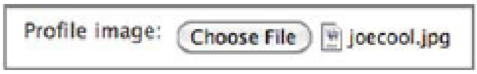


Figure 7.8 A file selection field in the Spitter registration form will enable users to put a face on their profile.

7.5.2 Receiving uploaded files

As before, the addSpitterFromForm() method will handle the registration form submissions. But we'll need to tweak that method to be able to accept an image upload. The following shows the new upload-ready addSpitterFromForm() method.

Listing 7.13 addSpitterFromForm() takes a MultipartFile as a parameter.

```
@RequestMapping(method=RequestMethod.POST)
public String addSpitterFromForm(@Valid Spitter spitter,
    BindingResult bindingResult,
    @RequestParam(value="image", required=false)       Accept file upload
        MultipartFile image) {
  if(bindingResult.hasErrors()) {
    return "spitters/edit";
  }

  spitterService.saveSpitter(spitter);

  try {
    if(!image.isEmpty()) {
      validateImage(image);                 Validate image

      saveImage(spitter.getId() + ".jpg", image); //       Save image file
    }
  } catch (ImageUploadException e) {
    bindingResult.reject(e.getMessage());
    return "spitters/edit";
  }

  return "redirect:/spitters/" + spitter.getUsername();
}
```

The first change made to addSpitterFromForm() is the addition of a new parameter. The image parameter is given as a MultipartFile and is annotated with @RequestParam to indicate that it's not required (so that a user can still register, even without a profile picture).

A little further down in the method, we check to make sure that the image isn't empty and, if it's not, then it's passed in to a validateImage() method and a saveImage() method. The validateImage(), as shown here, makes sure that the uploaded file meets our needs:

```
private void validateImage(MultipartFile image) {
  if(!image.getContentType().equals("image/jpeg")) {
    throw new ImageUploadException("Only JPG images accepted");
  }
}
```

We don't want to let users try to pass off zip or exe files as images. So validateImage() ensures that the uploaded file is a JPEG image. If this validation fails, an ImageUploadException (a simple extension of RuntimeException) will be thrown.

Once we're assured that the uploaded file is an image, we're ready to save it by calling the saveImage() method. The actual implementation of saveImage() could save the file almost anywhere, so long as it's available to the user's browser so that it can be rendered in the browser. Keeping it simple, let's start by writing an implementation of saveImage() that saves the image to the local file system.

SAVING FILES TO THE FILE SYSTEM

Even though our application will be accessible over the web, its resources ultimately reside in a file system on the host server. So it would seem natural to write the user profile pictures to a path on the local file system that the web server can serve the images from. The following implementation of saveImage does just that:

```
private void saveImage(String filename, MultipartFile image)
      throws ImageUploadException {
  try {
    File file = new File(webRootPath + "/resources/" + filename);
    FileUtils.writeByteArrayToFile(file, image.getBytes());
  } catch (IOException e) {
    throw new ImageUploadException("Unable to save image", e);
  }
}
```

Here, the first thing that saveImage() does is construct a java.io.File object whose path is based at the value of the webRootPath. We've purposefully left the value of that variable a mystery, as it depends on the server where the application is hosted. Suffice it to say that it could be configured by value injection, either through a setWebRootPath() method or perhaps using SpEL and an @Value annotation to read the value from a configuration file.

Once the `File` object is ready, we use `FileUtils` from Apache Commons IO[6] to write the image data to a file. If anything goes wrong, an `ImageUploadException` will be thrown.

Saving a file to the local file system like this works great, but leaves the management of the file system up to you. You'll be responsible for ensuring that there's plenty of space. It'll be up to you to make sure that it's backed up in case of a hardware failure. And it's your job to deal with synchronizing the image files across multiple servers in a cluster.

Another option is to let someone else take that hassle away from you. With a bit more code, we can save our images out on the cloud. Let's free ourselves from the burden of managing our own files by rewriting the `saveFile()` method to write to an Amazon S3 bucket.

SAVING FILES TO AMAZON S3

Amazon's *Simple Storage Service*, or *S3* as it's commonly referred to, is an inexpensive way to offload storage of files to Amazon's infrastructure. With S3, we can just write the files and let Amazon's system administrators do all of the grunt work.

The easiest way to use S3 in Java is with the JetS3t library.[7] JetS3t is an open source library for saving and reading files in the S3 cloud. We can use JetS3t to save user profile pictures. The following listing shows the new `saveImage()` method.

Listing 7.14 This `saveImage()` method posts a user's image to the Amazon S3 cloud

```
private void saveImage(String filename, MultipartFile image)
      throws ImageUploadException {

  try {
    AWSCredentials awsCredentials =
      new AWSCredentials(s3AccessKey, s3SecretKey);
    S3Service s3 = new RestS3Service(awsCredentials);     Set up S3 service

    S3Bucket imageBucket = s3.getBucket("spitterImages");
    S3Object imageObject = new S3Object(filename);        Create S3 bucket
                                                          and object
    imageObject.setDataInputStream(
            new ByteArrayInputStream(image.getBytes()));
    imageObject.setContentLength(image.getBytes().length);
    imageObject.setContentType("image/jpeg");             Set image data

    AccessControlList acl = new AccessControlList();      Set
    acl.setOwner(imageBucket.getOwner());                 permissions
    acl.grantPermission(GroupGrantee.ALL_USERS,
            Permission.PERMISSION_READ);
    imageObject.setAcl(acl);

    s3.putObject(imageBucket, imageObject);               Save image
  } catch (Exception e) {
    throw new ImageUploadException("Unable to save image", e);
  }
}
```

[6] http://commons.apache.org/io/

[7] http://bitbucket.org/jmurty/jets3t/wiki/Home

The first thing that `saveImage()` does is set up Amazon Web Service credentials. For this, you'll need an S3 access key and an S3 secret access key. These will be given to you by Amazon when you sign up for S3 service. They'll be given to `SpitterController` via value injection.

With the AWS credentials in hand, `saveImage()` creates an instance of JetS3t's `RestS3Service` through which it'll operate on the S3 file system. It gets a reference to the `spitterImages` bucket, creates an `S3Object` to contain the image, then fills that `S3Object` with image data.

Just before calling the `putObject()` method to write the image data to S3, `saveImage()` sets the permissions on the `S3Object` to allow all users to view it. This is important—without it, the images wouldn't be visible to our application's users.

As with the previous version of `saveImage()`, if anything goes wrong, a `ImageUploadException` will be thrown.

We're almost set for uploading profile pictures into the Spitter application. But there's one final bit of Spring configuration to tie it all together.

7.5.3 Configuring Spring for file uploads

On its own, `DispatcherServlet` doesn't know how to deal with multipart form data. We need a multipart resolver to extract the multipart data out of the POST request so that `DispatcherServlet` can give it to our controller.

To register a multipart resolver in Spring, we need to declare a bean that implements the `MultipartResolver` interface. Our choice of multipart resolvers is made easy by the fact that Spring only comes with one: `CommonsMultipartResolver`. It's configured in Spring as follows:

```
<bean id="multipartResolver" class=
     "org.springframework.web.multipart.commons.CommonsMultipartResolver"
     p:maxUploadSize="500000" />
```

Note that the multipart resolver's bean ID is significant. When `DispatcherServlet` looks for a multipart resolver, it'll look for it as a bean whose ID is `multipartResolver`. If the bean has any other ID, `DispatcherServlet` will overlook it.

7.6 Summary

In this chapter, we've built much of the web layer of the Spitter application. As we've seen, Spring comes with a powerful and flexible web framework. Employing annotations, Spring MVC offers a near-POJO development model, making simple work of developing controllers that handle requests and are simple to test. These controllers typically don't directly process requests, but instead delegate to other beans in the Spring application context that are injected into the controllers using Spring's dependency injection.

By employing handler mappings that choose controllers to handle requests and view resolvers to choose how results are rendered, Spring MVC maintains a loose coupling between how a controller is chosen to handle a request and how its view is chosen to display output. This sets Spring apart from many other MVC web frameworks, where choices are limited to one or two options.

Although the views developed in this chapter were written in JSP to produce HTML output, there's no reason why the model data produced by the controllers couldn't be rendered in some other form, including machine-readable XML or JSON. We'll see how to turn the Spitter application's web layer into a powerful web-based API in chapter 11 when we explore Spring's REST support further.

But for now, we'll continue looking at how to build user-facing web applications with Spring by exploring Spring Web Flow, an extension to Spring MVC that enables conversation-oriented web development in Spring.