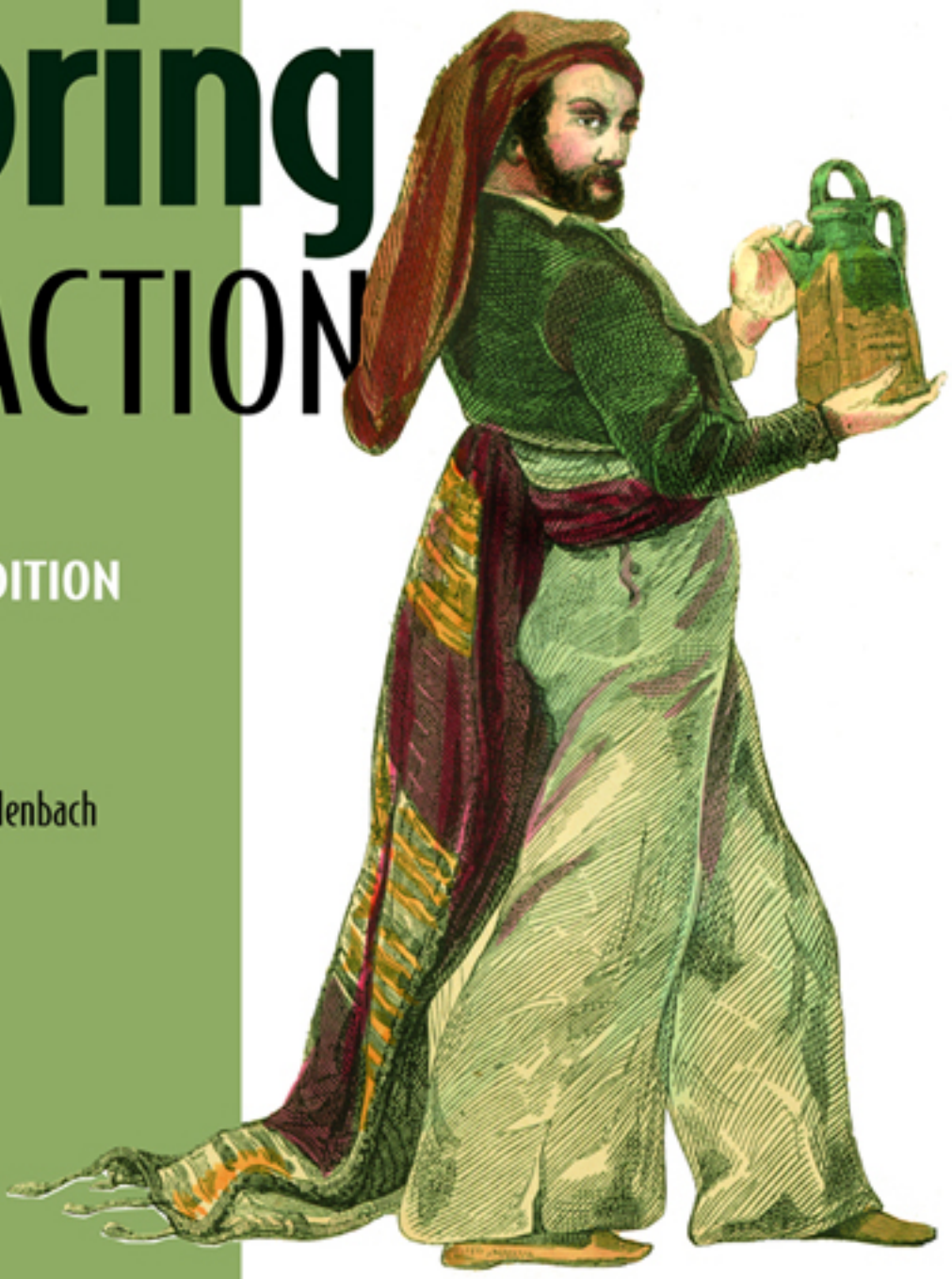
Sample Chapter

Updated for Spring 2.0

Spring IN ACTION

SECOND EDITION

Craig Walls
with Ryan Breidenbach



MANNING

13 Handling web requests

This chapter covers

- Mapping requests to Spring controllers
- Transparently binding form parameters
- Validating form submissions
- Mapping exceptions to views

As a JEE developer, you have more than likely developed a web-based application. In fact, for many Java developers, web-based applications are their primary focus. If you do have this type of experience, you are 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 and the one that follows, we'll explore the Spring MVC web framework. In this chapter, we'll focus on the parts of Spring MVC that process requests. You'll see how to extend Spring's rich set of controller objects to handle virtually any web functionality required in your application. You'll also see how Spring's handler mapping makes easy work of associating URL patterns with specific controller implementations. Chapter 14 will pick up where this chapter leaves off by showing you how to use Spring MVC views to send a response back to the user.

Before we go too deep with the specifics of Spring MVC's controllers and handler mappings, let's start with a high-level view of Spring MVC and build our first complete bit of web functionality.

13.1 Getting started with Spring MVC

Have you ever seen the children's game Mousetrap? It's a crazy game. 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 getting sprung 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 of 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 Goldberg-esque game of Mousetrap. Each of the components in Spring MVC performs a specific purpose. Let's start the exploration of Spring MVC by examining the lifecycle of a typical request.

13.1.1 A day in the life of a request

Every time that 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 Federal Express delivery person, a request lives to carry information from one place to another.

The request is a busy fellow. From the time that it leaves the browser until the time that it returns a response, it will make several stops, each time dropping off a bit of information and picking up some more. Figure 13.1 shows all the stops that the request makes.

When the request leaves the browser, it carries information about what the user is asking for. At very 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 Spring's `DispatcherServlet` ❶. Like most Java-based MVC 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 the 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 help deciding which controller to send the request to. So, the `DispatcherServlet`

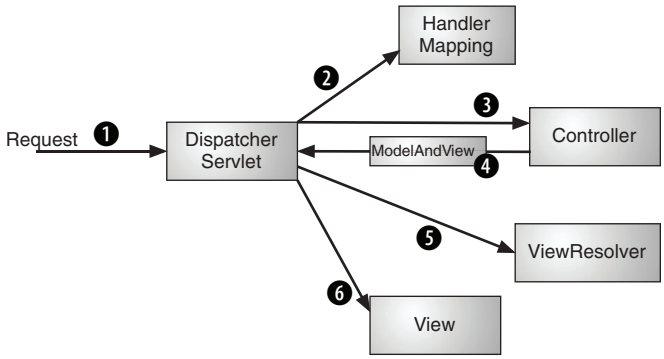


Figure 13.1 A request is dispatched by `DispatcherServlet` to a controller (which is chosen through a handler mapping). Once the controller is finished, the request is then sent to a view (which is chosen through a `ViewResolver`) to render output.

consults one or more handler mappings ❷ 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. ❸ At the controller, the request will drop off its payload (the information submitted by the user) and patiently wait for the controller to process 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.

So, the last thing that the controller will do is package up the model data and the name of a view into a `ModelAndView` object. ❹ It then sends the request, along with its new `ModelAndView` parcel, back to the `DispatcherServlet`. As its name implies, the `ModelAndView` object contains both the model data as well as a hint to what view should render the results.

So that the controller isn't coupled to a particular view, the `ModelAndView` doesn't carry a reference to the actual JSP. Instead it only carries a logical name that will be used to look up the actual view that will produce the resulting HTML. Once the `ModelAndView` is delivered to the `DispatcherServlet`, the `DispatcherServlet` asks a view resolver to help find the actual JSP. ❺

Now that the `DispatcherServlet` knows which view will render the results, the request's job is almost over. Its final stop is at the view implementation (probably a JSP) where it delivers the model data. ❻ With the model data delivered to the view, the request's job is done. The view will use the model data to render a page that will be carried back to the browser by the (not-so-hard-working) response object.

We'll discuss each of these steps in more detail throughout this and the next chapter. But first things first—you'll need to configure `DispatcherServlet` to use Spring MVC.

13.1.2 Configuring DispatcherServlet

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 your web application's web.xml file. Place the following `<servlet>` declaration in your application's web.xml file:

```
<servlet>
  <servlet-name>roadrantz</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 will 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 `roadrantz`, `DispatcherServlet` will try to load the application context from a file named roadrantz-servlet.xml.

Next you must indicate what URLs will be handled by the `DispatcherServlet`. Add the following `<servlet-mapping>` to web.xml to let `DispatcherServlet` handle all URLs that end in .htm:

```
<servlet-mapping>
  <servlet-name>roadrantz</servlet-name>
  <url-pattern>*.htm</url-pattern>
</servlet-mapping>
```

So, you're probably wondering why we chose this particular URL pattern. It could be because all of the content produced by our application is HTML. It could also be because we want to fool our friends into thinking that our entire application is composed of static HTML files. And it could be that we think .do is a silly extension.

But the truth of the matter is that the URL pattern is somewhat arbitrary and we could've chosen any URL pattern for `DispatcherServlet`. Our main reason for choosing *.htm is that this pattern is the one used by convention in most Spring MVC applications that produce HTML content. The reasoning behind this convention is that the content being produced is HTML and so the URL should reflect that fact.

Now that `DispatcherServlet` is configured in web.xml and given a URL mapping, you are ready to start writing the web layer of your application. However, there's still one more thing that we recommend you add to web.xml.

Breaking up the application context

As we mentioned earlier, `DispatcherServlet` will load the Spring application context from a single XML file whose name is based on its `<servlet-name>`. But this doesn't mean that you can't split your application context across multiple XML files. In fact, we recommend that you split your application context across application layers, as shown in figure 13.2.

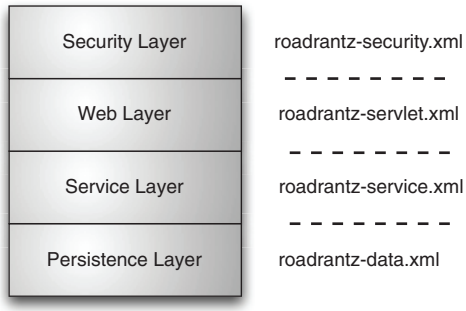


Figure 13.2
Breaking an application into separate tiers helps to cleanly divide responsibility. Security-layer code secures the application, web-layer code is focused on user interaction, service-layer code is focused on business logic, and persistence-layer code deals with database concerns.

As configured, `DispatcherServlet` already loads roadrantz-servlet.xml. You could put all of your application's `<bean>` definitions in roadrantz-servlet.xml, but eventually that file would become quite unwieldy. Splitting it into logical pieces across application layers can make maintenance easier by keeping each of the Spring configuration files focused on a single layer of the application. It also makes it easy to swap out a layer configuration without affecting other layers (swapping out a roadrantz-data.xml file that uses Hibernate with one that uses iBATIS, for example).

Because `DispatcherServlet`'s configuration file is roadrantz-servlet.xml, it makes sense for this file to contain `<bean>` definitions pertaining to controllers and other Spring MVC components. As for beans in the service and data layers, we'd like those beans to be placed in roadrantz-service.xml and roadrantz-data.xml, respectively.

Configuring a context loader

To ensure that all of these configuration files are loaded, you'll need to configure a context loader in your web.xml file. A context loader loads context configuration files in addition to the one that `DispatcherServlet` loads. The most commonly used context loader is a servlet listener called `ContextLoaderListener` that is configured in web.xml as follows:

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

NOTE Some web containers do not initialize servlet listeners before servlets—which is important when loading Spring context definitions. If your application is going to be deployed to an older web container that adheres to Servlet 2.2 or if the web container is a Servlet 2.3 container

that does not initialize listeners before servlets, you'll want to use `ContextLoaderServlet` instead of `ContextLoaderListener`.

With `ContextLoaderListener` configured, you'll need to tell it the location of the Spring configuration file(s) to load. If not specified otherwise, the context loader will look for a Spring configuration file at /WEB-INF/applicationContext.xml. But this location doesn't lend itself to breaking up the application context across application layers, so you'll probably want to override this default.

You can specify one or more Spring configuration files for the context loader to load by setting the `contextConfigLocation` parameter in the servlet context:

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>
    /WEB-INF/roadrantz-service.xml
    /WEB-INF/roadrantz-data.xml
    /WEB-INF/roadrantz-security.xml
  </param-value>
</context-param>
```

The `contextConfigLocation` parameter is specified as a list of paths (relative to the web application root). As configured here, the context loader will use `contextConfigLocation` to load three context configuration files—one for the security layer, one for the service layer, and one for the data layer.

`DispatcherServlet` is now configured and ready to dispatch requests to the web layer of your application. But the web layer hasn't been built yet! Don't fret. We'll build much of the web layer in this chapter. Let's start by getting an overview of how all the pieces of Spring MVC are assembled to produce web functionality.

13.1.3 Spring MVC in a nutshell

Every web application has a homepage. It's necessary to have a starting point in the application. It gives the user a place to launch from and a familiar place to return when they get lost. Otherwise, they would flail around, clicking links, getting frustrated, and probably ending up leaving and going to some other website.

The RoadRantz application is no exception to the homepage phenomenon. There's no better place to start developing the web layer of our application than with the homepage. In building the homepage, we get a quick introduction to the nuts and bolts of Spring MVC.

As you'll recall from the requirements for RoadRantz, the homepage should display a list of the most recently entered rants. The following list of steps defines the bare minimum that you must do to build the homepage in Spring MVC:

1. Write the controller class that performs the logic behind the homepage. The logic involves using a `RantService` to retrieve the list of recent rants.
2. Configure the controller in the `DispatcherServlet`'s context configuration file (roadrantz-servlet.xml).
3. Configure a view resolver to tie the controller to the JSP.
4. Write the JSP that will render the homepage to the user.

The first step is to build a controller object that will handle the homepage request. So with no further delay, let's write our first Spring MVC controller.

Building the controller

When you go out to eat at a nice restaurant, the person you'll interact with the most is the waiter or waitress. They'll take your order, hand it off to the cooks in the kitchen to prepare, and ultimately bring out your meal. And if they want a decent tip, they'll offer a friendly smile and keep the drinks filled. Although you know that other people are involved in making your meal a pleasant experience, the waiter or waitress is your interface to the kitchen.

Similarly, in Spring MVC, a controller is a class that is your interface to the application's functionality. As shown in figure 13.3, a controller receives the request, hands it off to service classes for processing, then ultimately collect the results in a page that is returned to you in your web browser. In this respect, a controller isn't much different than an `HttpServlet` or a Struts `Action`.

The homepage controller of the RoadRantz application is relatively simple. It takes no request parameters and simply produces a list of recently entered rants for display on the homepage. Listing 13.1 shows `HomePageController`, a Spring MVC controller that implements the homepage functionality.

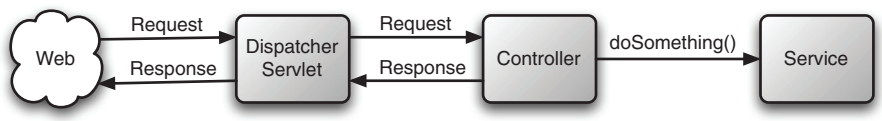


Figure 13.3 A controller handles web requests on behalf of the `DispatcherServlet`. A well-designed controller doesn't do all of the work itself—it delegates to a service layer object for business logic.

Listing 13.1 `HomePageController`, which retrieves a list of recent rants for display on the homepage

```
package com.roadrantz.mvc;
import java.util.List;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.AbstractController;
import com.roadrantz.service.RantService;

public class HomePageController extends AbstractController {
  public HomePageController() {}

  protected ModelAndView handleRequestInternal(
      HttpServletRequest request, HttpServletResponse response)
      throws Exception {

    List recentRants = rantService.getRecentRants();

    return new ModelAndView("home",
        "rants", recentRants);
  }

  private RantService rantService;
  public void setRantService(RantService rantService) {
    this.rantService = rantService;
  }
}
```

Retrieves list of rants

Goes to "home" view

Returns rants in model

Injects RantService

Where a Spring MVC controller differs from a servlet or a Struts `Action` is that it is configured as just another JavaBean in the Spring application context. This means you can take full advantage of dependency injection (DI) and Spring AOP with a controller class just as you would with any other bean.

In the case of `HomePageController`, DI is used to wire in a `RantService`. `HomePageController` delegates responsibility for retrieving the list of recent rants to the `RantService`.

Introducing ModelAndView

After the chef has prepared your meal, the waiter/waitress will pick it up and bring it to your table. On the way out, the last thing that they may do is add some final garnishments—perhaps a sprig of parsley.

Once the business logic has been completed by the service objects, it's time for the controller to send the results back to the browser. The last thing that `handleRequestInternal()` does is to return a `ModelAndView` object. The `ModelAndView` class represents an important concept in Spring MVC. In fact, every controller

execution method must return a `ModelAndView`. So, let's take a moment to understand how this important class works.

A `ModelAndView` object, as its name implies, fully encapsulates the view and model data that is to be displayed by the view. In the case of `HomePageController`, the `ModelAndView` object is constructed as follows:

```
new ModelAndView("home", "rants", recentRants);
```

The first parameter of this `ModelAndView` constructor is the logical name of a view component that will be used to display the output from this controller. Here the logical name of the view is `home`. A view resolver will use this name to look up the actual `View` object (you'll learn more about `View`s and view resolvers later in chapter 14).

The next two parameters represent the model object that will be passed to the view. These two parameters act as a name-value pair. The second parameter is the name of the model object given as the third parameter. In this case, the list of rants in the `recentRants` variable will be passed to the view with a name of `rants`.

Configuring the controller bean

Now that `HomePageController` has been written, it is time to configure it in the `DispatcherServlet`'s context configuration file (which is roadrantz-servlet.xml for the RoadRantz application). The following chunk of XML declares the `HomePageController`:

```
<bean name="/home.htm"
    class="com.roadrantz.mvc.HomePageController">
  <property name="rantService" ref="rantService" />
</bean>
```

As mentioned before, the `rantService` property is to be injected with an implementation of the `RantService` interface. In this `<bean>` declaration, we've wired the `rantService` property with a reference to another bean named `rantService`. The `rantService` bean itself is declared elsewhere (in roadrantz-service.xml, to be precise).

One thing that may have struck you as odd is that instead of specifying a bean `id` for the `HomePageController` bean, we've specified a `name`. And to make things even weirder, instead of giving it a real name, we've given it a URL pattern of /home.htm. Here the `name` attribute is serving double duty as both the name of the bean and a URL pattern for requests that should be handled by this controller. Because the URL pattern has special characters that are not valid in

an XML `id` attribute—specifically, the slash (`/`) character—the `name` attribute had to be used instead of `id`.

When a request comes to `DispatcherServlet` with a URL that ends with `/home.htm`, `DispatcherServlet` will dispatch the request to `HomePageController` for handling. Note, however, that the only reason that the bean's `name` attribute is used as the URL pattern is because we haven't configured a handler-mapping bean. The default handler mapping used by `DispatcherServlet` is `BeanNameUrlHandlerMapping`, which uses the base name as the URL pattern. Later (in section 13.2), you'll see how to use some of Spring's other handler mappings that let you decouple a controller's bean name from its URL pattern.

Declaring a view resolver

On the way back to the web browser, the results of the web operation need to be presented in a human-friendly format. Just like a waiter may place a sprig of parsley on a plate to make it more presentable, the resulting list of rants needs to be dressed up a bit before presenting it to the client. For that, we'll use a JSP page that will render the results in a user-friendly format.

But how does Spring know which JSP to use for rendering the results? As you'll recall, one of the values returned in the `ModelAndView` object is a logical view name. While the logical view name doesn't directly reference a specific JSP, it can be used to indirectly deduce which JSP to use.

To help Spring MVC figure out which JSP to use, you'll need to declare one more bean in roadrantz-servlet.xml: a view resolver. Put simply, a view resolver's job is to take the view name returned in the `ModelAndView` and map it to a view. In the case of `HomePageController`, we need a view resolver to resolve `home` (the logical view name returned in the `ModelAndView`) to a JSP file that renders the homepage.

As you'll see in section 13.4, Spring MVC comes with several view resolvers from which to choose. But for views that are rendered by JSP, there's none simpler than `InternalResourceViewResolver`:

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

`InternalResourceViewResolver` prefixes the view name returned in the `ModelAndView` with the value of its `prefix` property and suffixes it with the value from its `suffix` property. Since `HomePageController` returns a view name of home in the `ModelAndView`, `InternalResourceViewResolver` will find the view at /WEB-INF/jsp/home.jsp.

Creating the JSP

We've written a controller that will handle the homepage request and have configured it in the Spring application context. It will consult with a `RantService` bean to look up the most recently added rants. And when it's done, it will send the results on to a JSP. So now we only have to create the JSP that renders the homepage. The JSP in listing 13.2 iterates over the list of rants and displays them on the home page.

Listing 13.2 home.jsp, which displays a list of recent rants

```
<%@ page contentType="text/html" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>

<html>
  <head><title>Rantz</title></head>

  <body>
    <h2>Welcome to RoadRantz!</h2>

    <h3>Recent rantz:</h3>
    <ul>
    <c:forEach items="${rants}" var="rant">
    <li><c:out value="${rant.vehicle.state}"/> /
        <c:out value="${rant.vehicle.plateNumber}"/> --
        <c:out value="${rant.rantText}"/>
    </li>
    </c:forEach>
    </ul>
  </body>
</html>
```

Iterates over list of rants

Although we've left out any aesthetic elements in home.jsp for brevity's sake, it still serves to illustrate how the model data returned in `ModelAndView` can be used in the view. In `HomePageController`, we placed the list of rants in a model property named `rants`. When home.jsp is rendering the homepage, it references the list of rants as `${rants}`.

Be sure to name this JSP home.jsp and to place it in the /WEB-INF/jsp folder in your web application. That's where `InternalResourceViewResolver` will try to find it.

Putting it all together

The homepage is now complete. You've written a controller to handle requests for the homepage, configured it to rely on `BeanNameUrlHandlerMapping` to have a URL pattern of /home.htm, written a simple JSP that represents the homepage, and configured a view resolver to find the JSP. Now, how does this all fit together?

Figure 13.4 shows the steps that a request for /home.htm will go through given the work done so far.

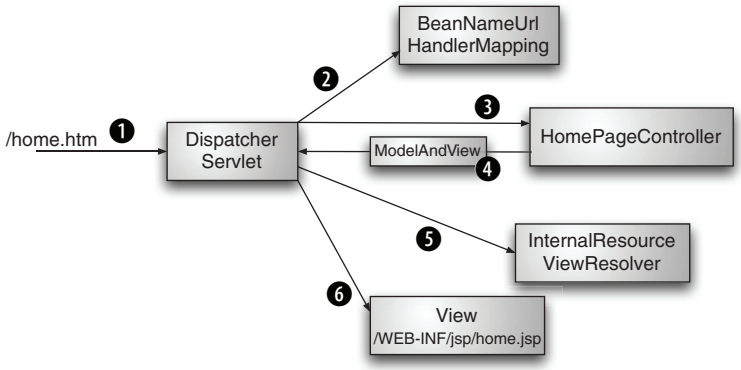


Figure 13.4 A homepage request is sent by `DispatcherServlet` to the `HomePageController` (as directed by `BeanNameUrlHandlerMapping`). When finished, `InternalResourceViewResolver` directs the request to home.jsp to render the homepage.

To recap this process:

1. `DispatcherServlet` receives a request whose URL pattern is `/home.htm`.
2. `DispatcherServlet` consults `BeanNameUrlHandlerMapping` to find a controller whose bean name is `/home.htm`; it finds the `HomePageController` bean.
3. `DispatcherServlet` dispatches the request to `HomePageController` for processing.
4. `HomePageController` returns a `ModelAndView` object with a logical view name of `home` and a list of rants in a property called `rants`.

5. `DispatcherServlet` consults its view resolver (configured as `InternalResourceViewResolver`) to find a view whose logical name is `home`. `InternalResourceViewResolver` returns the path to /WEB-INF/jsp/home.jsp.
6. `DispatcherServlet` forwards the request to the JSP at /WEB-INF/jsp/home.jsp to render the homepage to the user.

Now that you've seen the big picture of Spring MVC, let's slow down a bit and take a closer look at each of the moving parts involved in servicing a request. We'll start where it all begins—with handler mappings.

13.2 Mapping requests to controllers

When a courier has a package that is to be delivered to a particular office within a large office building, they'll need to know how to find the office. In a large office building with many tenants, this would be tricky if it weren't for a building directory. The building directory is often located near the elevators and helps anyone unfamiliar with the building locate the floor and suite number of the office they're looking for.

In the same way, when a request arrives at the `DispatcherServlet`, there needs to be some directory to help figure out how the request should be dispatched. Handler mappings help `DispatcherServlet` figure out which controller the request should be sent to. Handler mappings typically map a specific controller bean to a URL pattern. This is similar to how URLs are mapped to servlets using a `<servlet-mapping>` element in a web application's web.xml file or how `Actions` in Jakarta Struts are mapped to URLs using the `path` attribute of `<action>` in struts-config.xml.

In the previous section, we relied on the fact that `DispatcherServlet` defaults to use `BeanNameUrlHandlerMapping`. `BeanNameUrlHandlerMapping` was fine to get started, but it may not be suitable in all cases. Fortunately, Spring MVC offers several handler-mapping implementations to choose from.

All of Spring MVC's handler mappings implement the `org.springframework.web.servlet.HandlerMapping` interface. Spring comes prepackaged with four useful implementations of `HandlerMapping`, as listed in table 13.1.

You've already seen an example of how `BeanNameUrlHandlerMapping` works (as the default handler mapping used by `DispatcherServlet`). Let's look at how to use each of the other handler mappings, starting with `SimpleUrlHandlerMapping`.

Table 13.1 Handler mappings help `DispatcherServlet` find the right controller to handle a request.

Handler mapping	How it maps requests to controllers
`BeanNameUrlHandlerMapping`	Maps controllers to URLs that are based on the controllers' bean name.
`SimpleUrlHandlerMapping`	Maps controllers to URLs using a property collection defined in the Spring application context.
`ControllerClassNameHandlerMapping`	Maps controllers to URLs by using the controller's class name as the basis for the URL.
`CommonsPathMapHandlerMapping`	Maps controllers to URLs using source-level metadata placed in the controller code. The metadata is defined using Jakarta Commons Attributes (http://jakarta.apache.org/commons/attributes).

13.2.1 Using SimpleUrlHandlerMapping

`SimpleUrlHandlerMapping` is probably one of the most straightforward of Spring's handler mappings. It lets you map URL patterns directly to controllers without having to name your beans in a special way.

For example, consider the following declaration of `SimpleUrlHandlerMapping` that associates several of the RoadRantz application's controllers with their URL patterns:

```
<bean id="simpleUrlMapping" class=
    "org.springframework.web.servlet.handler.
        ➥ SimpleUrlHandlerMapping">
  <property name="mappings">
    <props>
      <prop key="/home.htm">homePageController</prop>
      <prop key="/rantsForVehicle.htm">
          ➥ rantsForVehicleController</prop>
      <prop key="/rantsForVehicle.rss">
          ➥ rantsForVehicleControllerRss</prop>
      <prop key="/rantsForDay.htm">rantsForDayController</prop>
      <prop key="/login.htm">loginController</prop>
      <prop key="/register.htm">registerMotoristController</prop>
      <prop key="/addRant.htm">addRantController</prop>
    </props>
  </property>
</bean>
```

`SimpleUrlHandlerMapping`'s `mappings` property is wired with a `java.util.Properties` using `<props>`. The key attribute of each `<prop>` element is a URL pattern.

Just as with `BeanNameUrlHandlerMapping`, all URL patterns are relative to `DispatcherServlet`'s `<servlet-mapping>`. URL. The value of each `<prop>` is the bean name of a controller that will handle requests to the URL pattern.

In case you're wondering where all of those other controllers came from, just hang tight. By the time this chapter is done, we'll have seen most of them. But first, let's explore another way to declare controller mappings using the class names of the controllers.

13.2.2 Using ControllerClassNameHandlerMapping

Oftentimes you'll find yourself mapping your controllers to URL patterns that are quite similar to the class names of the controllers. For example, in the RoadRantz application, we're mapping `rantsForVehicle.htm` to `RantsForVehicleController` and `rantsForDay.htm` to `RantsForDayController`.

Notice a pattern? In those cases, the URL pattern is the same as the name of the controller class, dropping the `Controller` portion and adding `.htm`. It seems that with a pattern like that it would be possible to assume a certain default for the mappings and not require explicit mappings.

In fact, that's roughly what `ControllerClassNameHandlerMapping` does:

```
<bean id="urlMapping"
    class="org.springframework.web.servlet.mvc.
    ➥ ControllerClassNameHandlerMapping"/>
```

By configuring `ControllerClassNameHandlerMapping`, you are telling Spring's `DispatcherServlet` to map URL patterns to controllers following a simple convention. Instead of explicitly mapping each controller to a URL pattern, Spring will automatically map controllers to URL patterns that are based on the controller's class name. Figure 13.5 illustrates how `RantsForVehicleController` will be mapped.

Put simply, to produce the URL pattern, the `Controller` portion of the controller's class name is removed (if it exists), the remaining text is lowercased, a slash (`/`) is added to the beginning, and ".htm" is added to the end to produce the URL pattern. Consequently, a controller bean whose class is `RantsForVehicleController` will be mapped to `/rantsforvehicle.htm`. Notice that the entire

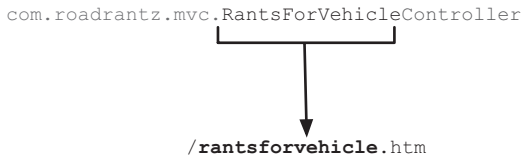


Figure 13.5 `ControllerClassNameHandlerMapping` maps a request to a controller by stripping `Controller` from the end of the class name and normalizing it to lowercase.

URL pattern is lowercased, which is slightly different from the convention we were following with `SimpleUrlHandlerMapping`.

13.2.3 Using metadata to map controllers

The final handler mapping we'll look at is `CommonsPathMapHandlerMapping`. This handler mapping considers source-level metadata placed in a controller's source code to determine the URL mapping. In particular, the metadata is expected to be an `org.springframework.web.servlet.handler.commonsattributes.PathMap` attribute compiled into the controller using the Jakarta Commons Attributes compiler.

To use `CommonsPathMapHandlerMapping`, simply declare it as a `<bean>` in your context configuration file as follows:

```
<bean id="urlMapping" class="org.springframework.web.
       servlet.handler.metadata.CommonsPathMapHandlerMapping"/>
```

Then tag each of your controllers with a `PathMap` attribute to declare the URL pattern for the controller. For example, to map `HomePageController` to `/home.htm`, tag `HomePageController` as follows:

```
/**
 * @@org.springframework.web.servlet.handler.
    ➥ commonsattributes.PathMap("/home.htm")
 */
public class HomePageController
    extends AbstractController {
…
}
```

Finally, you'll need to set up your build to include the Commons Attributes compiler so that the attributes will be compiled into your application code. We refer you to the Commons Attributes homepage (http://jakarta.apache.org/commons/attributes) for details on how to set up the Commons Attributes compiler in either Ant or Maven.

13.2.4 Working with multiple handler mappings

As you've seen, Spring comes with several useful handler mappings. But what if you can't decide which to use? For instance, suppose your application has been simple and you've been using `BeanNameUrlHandlerMapping`. But it is starting to grow and you'd like to start using `SimpleUrlHandlerMapping` going forward. How can you mix-'n'-match handler mappings during the transition?

As it turns out, all of the handler mapping classes implement Spring's `Ordered` interface. This means that you can declare multiple handler mappings in your application and set their `order` properties to indicate which has precedence with relation to the others.

For example, suppose you want to use both `BeanNameUrlHandlerMapping` and `SimpleUrlHandlerMapping` alongside each other in the same application. You'd need to declare the handler mapping beans as follows:

```
<bean id="beanNameUrlMapping" class="org.springframework.web.
       ➥ servlet.handler.BeanNameUrlHandlerMapping">
  <property name="order"><value>1</value></property>
</bean>
<bean id="simpleUrlMapping" class="org.springframework.web.
       ➥ servlet.handler.SimpleUrlHandlerMapping">
  <property name="order"><value>0</value></property>
  <property name="mappings">
…
  </property>
</bean>
```

Note that the lower the value of the `order` property, the higher the priority. In this case, `SimpleUrlHandlerMapping`'s `order` is lower than that of `BeanNameUrlHandlerMapping`. This means that `DispatcherServlet` will consult `SimpleUrlHandlerMapping` first when trying to map a URL to a controller. `BeanNameUrlHandlerMapping` will only be consulted if `SimpleUrlHandlerMapping` turns up no results.

Spring's handler mappings help `DispatcherServlet` know which controller a request should be directed to. After `DispatcherServlet` has figured out where to send the request, it's up to a controller to process it. Next up, let's have a look at how to create controllers in Spring MVC.

13.3 Handling requests with controllers

If `DispatcherServlet` is the heart of Spring MVC then controllers are the brains. When implementing the behavior of your Spring MVC application, you extend one of Spring's controller classes. The controller receives requests from `DispatcherServlet` and performs some business functionality on behalf of the user.

If you're familiar with other web frameworks such as Struts or WebWork, you may recognize controllers as being roughly equivalent in purpose to a Struts or WebWork action. One huge difference between Spring controllers and Struts/WebWork actions, however, is that Spring provides a rich controller hierarchy (as

shown in figure 13.6) in contrast to the rather flat action hierarchy of Struts or WebWork.

At first glance, figure 13.6 may seem somewhat daunting. Indeed, when compared to other MVC frameworks such as Jakarta Struts or WebWork, there's a lot more to swallow with Spring's controller hierarchy. In reality, however, this perceived complexity is actually quite simple and flexible.

At the top of the controller hierarchy is the `Controller` interface. Any class implementing this interface can be used to handle requests through the Spring

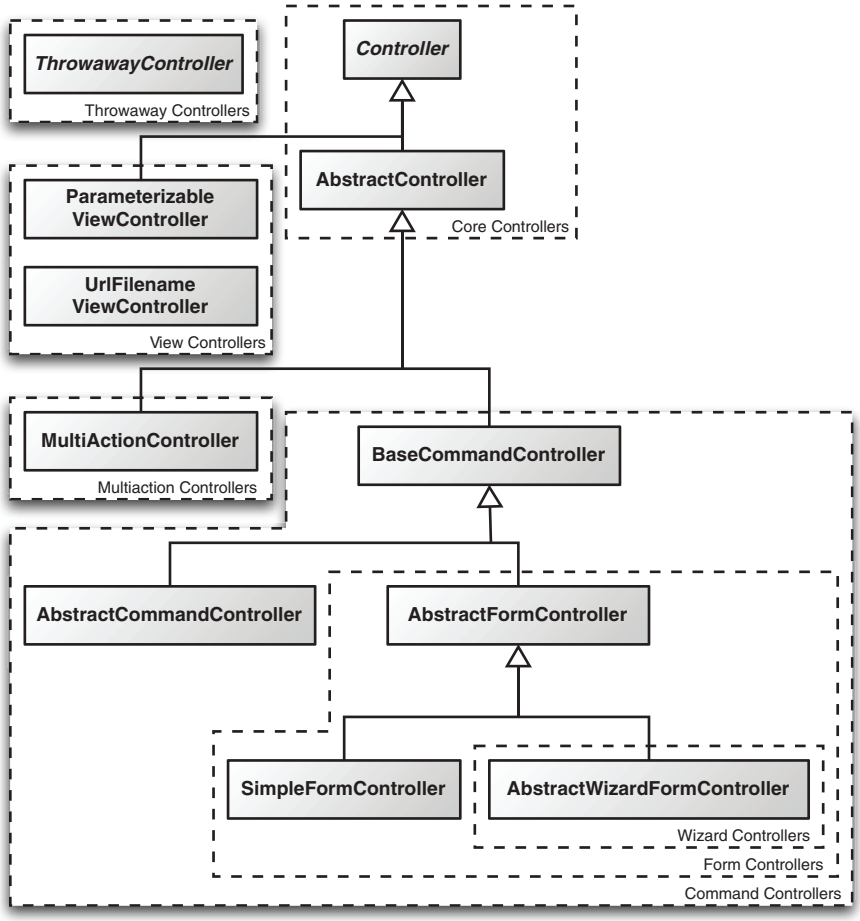


Figure 13.6 Spring MVC's controller hierarchy includes controllers for every occasion—from the simplest requests to more complex form processing.

MVC framework. To create your own controller, all you must do is write a class that implements this interface.

While you could write a class that directly implements the `Controller` interface, you're more likely to extend one of the classes lower in the hierarchy. Whereas the `Controller` interface defines the basic contract between a controller and Spring MVC, the various controller classes provide additional functionality beyond the basics.

The wide selection of controller classes is both a blessing and a curse. Unlike other frameworks that force you to work with a single type of controller object (such as Struts's `Action` class), Spring lets you choose the controller that is most appropriate for your needs. However, with so many controller classes to choose from, many developers find themselves overwhelmed and don't know how to decide.

To help you decide which controller class to extend for your application's controllers, consider table 13.2. As you can see, Spring's controller classes can be grouped into six categories that provide more functionality (and introduce more complexity) as you progress down the table. You may also notice from figure 13.5 that (with the exception of `ThrowawayController`) as you move down the controller hierarchy, each controller builds on the functionality of the controllers above it.

Table 13.2 Spring MVC's selection of controller classes.

Controller type	Classes	Useful when...
View	`ParameterizableViewController` `UrlFilenameViewController`	Your controller only needs to display a static view—no processing or data retrieval is needed.
Simple	`Controller (interface)` `AbstractController`	Your controller is extremely simple, requiring little more functionality than is afforded by basic Java servlets.
Throwaway	`ThrowawayController`	You want a simple way to handle requests as commands (in a manner similar to WebWork `Action`s).
Multiaction	`MultiActionController`	Your application has several actions that perform similar or related logic.

Table 13.2 Spring MVC's selection of controller classes. *(continued)*

Controller type	Classes	Useful when…
Command	`BaseCommandController` `AbstractCommandController`	Your controller will accept one or more parameters from the request and bind them to an object. Also capable of performing parameter validation.
Form	`AbstractFormController` `SimpleFormController`	You need to display an entry form to the user and also process the data entered into the form.
Wizard	`AbstractWizardFormController`	You want to walk your user through a complex, multipage entry form that ultimately gets processed as a single form.

You've already seen an example of a simple controller that extends `AbstractController`. In listing 13.1, `HomePageController` extends `AbstractController` and retrieves a list of the most recent rants for display on the home page. `AbstractController` is a perfect choice because the homepage is so simple and takes no input.

Basing your controller on `AbstractController` is fine when you don't need a lot of power. Most controllers, however, are going to be more interesting, taking parameters and requiring validation of those parameters. In the sections that follow, we're going to build several controllers that define the web layer of the RoadRantz application by extending the other implementations of the `Controller` classes in figure 13.6, starting with command controllers.

13.3.1 Processing commands

It's common for a web request to take one or more parameters that are used to determine the results. For instance, one of the requirements for the RoadRantz application is to display a list of rants for a particular vehicle.

Of course, you could extend `AbstractController` and retrieve the parameters your controller needs from the `HttpServletRequest`. But you would also have to write the logic that binds the parameters to business objects and you'd have to put validation logic in the controller itself. Binding and validation logic really don't belong in the controller.

In the event that your controller will need to perform work based on parameters, your controller class should extend a command controller class such as

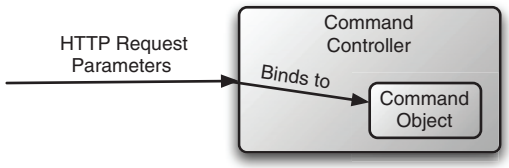


Figure 13.7
Command controllers relieve you from the hassle of dealing with request parameters directly. They bind the request parameters to a command object that you'll work with instead.

`AbstractCommandController`. As shown in figure 13.7, command controllers automatically bind request parameters to a command object. They can also be wired to plug in validators to ensure that the parameters are valid.

Listing 13.3 shows `RantsForVehicleController`, a command controller that is used to display a list of rants that have been entered for a specific vehicle.

Listing 13.3 `RantsForVehicleController`, which lists all rants for a particular vehicle

```
package com.roadrantz.mvc;
import java.util.List;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.validation.BindException;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.
    ➥ AbstractCommandController;
import com.roadrantz.domain.Vehicle;
import com.roadrantz.service.RantService;

public class RantsForVehicleController
    extends AbstractCommandController {

  public RantsForVehicleController() {
    setCommandClass(Vehicle.class);
    setCommandName("vehicle");
  }

  protected ModelAndView handle(HttpServletRequest request,
      HttpServletResponse response, Object command,
      BindException errors) throws Exception {

    Vehicle vehicle = (Vehicle) command;

    List vehicleRants =
        rantService.getRantsForVehicle(vehicle));

    Map model = errors.getModel();
    model.put("rants",4
        rantService.getRantsForVehicle(vehicle));
    model.put("vehicle", vehicle);

    return new ModelAndView("vehicleRants", model);
  }
```

Sets command class, name

Casts command object to Vehicle

Uses RantService to retrieve list of rants

Creates the model

Returns model

```
  private RantService rantService;
  public void setRantService(RantService rantService) {
    this.rantService = rantService;
  }
}
```

The `handle()` method of `RantsForVehicleController` is the main execution method for `AbstractCommandController`. This method is a bit more interesting than the `handleRequestInternal()` method from `AbstractController`. In addition to an `HttpServletRequest` and an `HttpServletResponse`, `handle()` takes an `Object` that is the controller's command.

A command object is a bean that is meant to hold request parameters for easy access. If you are familiar with Jakarta Struts, you may recognize a command object as being similar to a Struts `ActionForm`. The key difference is that unlike a Struts form bean that must extend `ActionForm`, a Spring command object is a POJO that doesn't need to extend any Spring-specific classes.

In this case, the command object is an instance of `Vehicle`, as set in the controller's constructor. You may recognize `Vehicle` as the domain class that describes a vehicle from chapter 5. Although command classes don't have to be instances of domain classes, it is sure handy when they are. `Vehicle` already defines the same data needed by `RantsForVehicleController`. Conveniently, it's also the exact same type needed by the `getRantsForVehicle()` method of `RantService`. This makes it a perfect choice for a command class.

Before the `handle()` method is called, Spring will attempt to match any parameters passed in the request to properties in the command object. `Vehicle` has two properties: `state` and `plateNumber`. If the request has parameters with these names, the parameter values will automatically be bound to the `Vehicle`'s properties.

As with `HomePageController`, you'll also need to register `RantsForVehicleController` in roadrantz-servlet.xml:

```
<bean id="rantsForVehicleController"
    class="com.roadrantz.mvc.RantsForVehicleController">
  <property name="rantService" ref="rantService" />
</bean>
```

Command controllers make it easy to handle requests with request parameters by binding the request parameters to command objects. The request parameters could be given as URL parameters (as is likely the case with `RantsForVehicleController`) or as fields from a web-based form. Although command controllers can process input from a form, Spring provides another type of controller with better support for form handling. Let's have a look at Spring's form controllers next.

13.3.2 Processing form submissions

In a typical web-based application, you're likely to encounter at least one form that you must fill out. When you submit that form, the data that you enter is sent to the server for processing, and once the processing is completed, you are either presented with a success page or are given the form page with errors in your submission that you must correct.

The core functionality of the RoadRantz application is the ability to enter a rant about a particular vehicle. In the application, the user will be presented with a form to enter their rant. Upon submission of that form, the expectation is that the rant will be saved to the database for later viewing.

When implementing the rant submission process, you might be tempted to extend `AbstractController` to display the form and to extend `AbstractCommandController` to process the form. This could certainly work, but would end up being more difficult than necessary. You would have to maintain two different controllers that work in tandem to process rant submissions. Wouldn't it be simpler to have a single controller handle both form display and form processing?

What you'll need in this case is a form controller. Form controllers take the concept of command controllers a step further, as shown in figure 13.8, by adding functionality to display a form when an HTTP GET request is received and process the form when an HTTP POST is received. Furthermore, if any errors occur in processing the form, the controller will know to redisplay the form so that the user can correct the errors and resubmit.

To illustrate how form controllers work, consider `AddRantFormController` in listing 13.4.

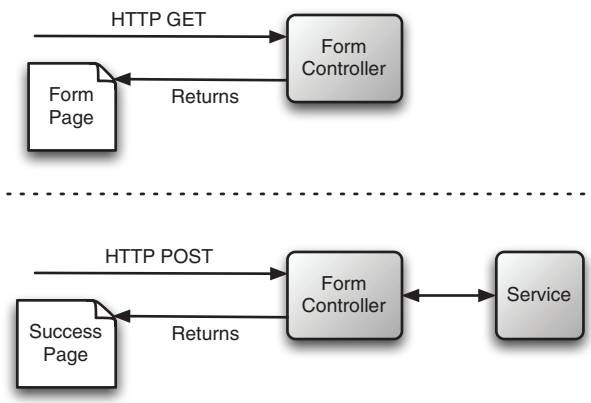


Figure 13.8
On an HTTP GET request, form controllers display a form to collect user input. Upon submitting the form with an HTTP POST, the form controller processes the input and returns a success page.

Listing 13.4 A controller for adding new rants

```
package com.roadrantz.mvc;
import java.util.HashMap;
import java.util.Map;
import javax.servlet.http.HttpServletRequest;
import org.springframework.validation.BindException;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.SimpleFormController;
import com.roadrantz.domain.Rant;
import com.roadrantz.domain.Vehicle;
import com.roadrantz.service.RantService;

public class AddRantFormController extends SimpleFormController {
  private static final String[] ALL_STATES = {
    "AL", "AK", "AZ", "AR", "CA", "CO", "CT", "DE", "DC", "FL",
    "GA", "HI", "ID", "IL", "IN", "IA", "KS", "KY", "LA", "ME",
    "MD", "MA", "MI", "MN", "MS", "MO", "MT", "NE", "NV", "NH",
    "NJ", "NM", "NY", "NC", "ND", "OH", "OK", "OR", "PA", "RI",
    "SC", "SD", "TN", "TX", "UT", "VA", "VT", "WA", "WV", "WI",
    "WY"
  };

  public AddRantFormController() {
    setCommandClass(Rant.class);
    setCommandName("rant");
  }

  protected Object formBackingObject(HttpServletRequest request)
      throws Exception {
    Rant rantForm = (Rant) super.formBackingObject(request);
    rantForm.setVehicle(new Vehicle());
    return rantForm;
  }

  protected Map referenceData(HttpServletRequest request)
      throws Exception {
    Map referenceData = new HashMap();
    referenceData.put("states", ALL_STATES);
    return referenceData;
  }

  protected ModelAndView onSubmit(Object command,
      BindException bindException) throws Exception {

    Rant rant = (Rant) command;

    rantService.addRant(rant);

    return new ModelAndView(getSuccessView());
  }

  private RantService rantService;
  public void setRantService(RantService rantService) {
```

Sets command class, name

Sets up Rant command with blank Vehicle

Provides list of states for form

Adds new rant

```
    this.rantService = rantService;
  }
}
```

Although it may not be obvious, `AddRantFormController` is responsible for both displaying a rant entry form and processing the results of that form. When this controller receives an HTTP GET request, it will direct the request to the form view. And when it receives an HTTP POST request, the `onSubmit()` method will process the form submission.

The `referenceData()` method is optional, but is handy when you need to provide any additional information for displaying the form. In this case, our form will need a list of states that will be displayed (presumably in a drop-down selection list). So, the `referenceData()` method of `AddRantFormController` adds an array of `String`s that contains all 50 U.S. states as well as the District of Columbia.

Under normal circumstances, the command object that backs the form is simply an instance of the command class. In the case of `AddRantFormController`, however, a simple `Rant` instance will not do. The form is going to use the nested `Vehicle` property within a `Rant` as part of the form-backing object. Therefore, it was necessary to override the `formBackingObject()` method to set the `vehicle` property. Otherwise, a `NullPointerException` would be thrown when the controller attempts to bind the `state` and `plateNumber` properties.

The `onSubmit()` method handles the form submission (an HTTP POST request) by passing the command object (which is an instance of `Rant`) to the `addRant()` method of the injected `RantService` reference.

What's not clear from listing 13.4 is how this controller knows to display the rant entry form. It's also not clear where the user will be taken after the rant has been successfully added. The only hint is that the result of a call to `getSuccessView()` is given to the `ModelAndView`. But where does the success view come from?

`SimpleFormController` is designed to keep view details out of the controller's Java code as much as possible. Instead of hard-coding a `ModelAndView` object, you configure the form controller in the context configuration file as follows:

```
<bean id="addRantController"
    class="com.roadrantz.mvc.AddRantFormController">
  <property name="formView" value="addRant" />
  <property name="successView" value="rantAdded" />
  <property name="rantService" ref="rantService" />
</bean>
```

Just as with the other controllers, the `addRantController` bean is wired with any services that it may need (e.g., `rantService`). But here you also specify a `formView` property and a `successView` property. The `formView` property is the logical name of a view to display when the controller receives an HTTP GET request or when any errors are encountered. Likewise, the `successView` is the logical name of a view to display when the form has been submitted successfully. A view resolver (see section 13.4) will use these values to locate the `View` object that will render the output to the user.

Validating form input

When `AddRantFormController` calls `addRant()`, it's important to ensure that all of the data in the `Rant` command is valid and complete. You don't want to let users enter only a state and no plate number (or vice versa). Likewise, what's the point in specifying a state and plate number but not providing any text in the rant? And it's important that the user not enter a plate number that isn't valid.

The `org.springframework.validation.Validator` interface accommodates validation for Spring MVC. It is defined as follows:

```
public interface Validator {
  void validate(Object obj, Errors errors);
  boolean supports(Class clazz);
}
```

Implementations of this interface should examine the fields of the object passed into the `validate()` method and reject any invalid values via the `Errors` object. The `supports()` method is used to help Spring determine whether the validator can be used for a given class.

`RantValidator` (listing 13.5) is a `Validator` implementation used to validate a `Rant` object.

Listing 13.5 Validating a Rant entry

```
package com.roadrantz.mvc;
import org.apache.oro.text.perl.Perl5Util;
import org.springframework.validation.Errors;
import org.springframework.validation.ValidationUtils;
import org.springframework.validation.Validator;
import com.roadrantz.domain.Rant;

public class RantValidator implements Validator {
  public boolean supports(Class clazz) {
      return clazz.equals(Rant.class);
  }

  public void validate(Object command, Errors errors) {
```

```
    Rant rant = (Rant) command;

    ValidationUtils.rejectIfEmpty(
        errors, "vehicle.state", "required.state",
        "State is required.");

    ValidationUtils.rejectIfEmpty(
        errors, "vehicle.plateNumber", "required.plateNumber",
        "The license plate number is required.");

    ValidationUtils.rejectIfEmptyOrWhitespace(
        errors, "rantText", "required.rantText",
        "You must enter some rant text.");

    validatePlateNumber(
        rant.getVehicle().getPlateNumber(), errors);
  }

  private static final String PLATE_REGEXP =
    "/[a-z0-9]{2,6}/i";

  private void validatePlateNumber(
      String plateNumber, Errors errors) {
    Perl5Util perl5Util = new Perl5Util();
    if(!perl5Util.match(PLATE_REGEXP, plateNumber)) {
      errors.reject("invalid.plateNumber",
          "Invalid license plate number.");
    }
  }
}
```

Validates required fields

Validates plate numbers

The only other thing to do is to configure `AddRantFormController` to use `RantValidator`. You can do this by wiring a `RantValidator` bean into the `AddRantFormController` bean (shown here as an inner bean):

```
<bean id="addRantController"
    class="com.roadrantz.mvc.AddRantFormController">
  <property name="formView" value="addRant" />
  <property name="successView" value="rantAdded" />
  <property name="rantService" ref="rantService" />
  <property name="validator">
    <bean class="com.roadrantz.mvc.RantValidator" />
  </property>
</bean>
```

When a rant is entered, if all of the required properties are set and if the plate number passes validation, `AddRantFormController`'s `onSubmit()` will be called and the rant will be added. However, if `RantValidator` rejects any of the fields, the user will be returned to the form view to correct the mistakes.

By implementing the `Validator` interface, you are able to programmatically take full control over the validation of your application's command objects. This may be perfect if your validation needs are complex and require special logic.

However, in simple cases such as ensuring required fields and basic formatting, writing our own implementation of the `Validator` interface is a bit too involved. It'd be nice if we could write validation rules declaratively instead of having to write validation rules in Java code. Let's have a look at how to use declarative validation with Spring MVC.

Validating with Commons Validator

One complaint that we've heard about Spring MVC is that validation with the `Validator` interface doesn't even compare to the kind of validation possible with Jakarta Struts. We can't argue with that complaint. Jakarta Struts has a very nice facility for declaring validation rules outside of Java code. The good news is that we can do declarative validation with Spring MVC, too.

But before you go digging around in Spring's JavaDoc for a declarative `Validator` implementation, you should know that Spring doesn't come with such a validator. In fact, Spring doesn't come with any implementations of the `Validator` interface and leaves it up to you to write your own.

However, you don't have to go very far to find an implementation of `Validator` that supports declarative validation. The Spring Modules project (https://springmodules.dev.java.net) is a sister project of Spring that provides several extensions to Spring whose scope exceeds that of the main Spring project. One of those extensions is a validation module that makes use of Jakarta Commons Validator (http://jakarta.apache.org/commons/validator) to provide declarative validation.

To use the validation module in your application, you start by making the springmodules-validator.jar file available in the application's classpath. If you're using Ant to do your builds, you'll need to download the Spring Modules distribution (I'm using version 0.6) and find the spring-modules-0.6.jar file in the dist directory. Add this JAR to the `<war>` task's `<lib>` to ensure that it gets placed in the WEB-INF/lib directory of the application's WAR file.

If you're using Maven 2 to do your build (as I'm doing), you'll need to add the following `<dependency>` to pom.xml:

```
<dependency>
  <groupId>org.springmodules</groupId>
  <artifactId>springmodules-validation</artifactId>
  <version>0.6</version>
```

```
    <scope>compile</scope>
  </dependency>
```

You'll also need to add the Jakarta Commons Validator JAR to your application's classpath. In Maven 2, it will look like this:

```
<dependency>
  <groupId>commons-validator</groupId>
  <artifactId>commons-validator</artifactId>
  <version>1.1.4</version>
  <scope>compile</scope>
</dependency>
```

Spring Modules provides an implementation of `Validator` called `DefaultBeanValidator`. `DefaultBeanValidator` is configured in roadrantz-servlet.xml as follows:

```
<bean id="beanValidator" class=
    "org.springmodules.commons.validator.DefaultBeanValidator">
  <property name="validatorFactory" ref="validatorFactory" />
</bean>
```

`DefaultBeanValidator` doesn't do any actual validation work. Instead, it delegates to Commons Validator to validate field values. As you can see, `DefaultBeanValidator` has a `validatorFactory` property that is wired with a reference to a `validatorFactory` bean. The `validatorFactory` bean is declared using the following XML:

```
<bean id="validatorFactory" class=
    "org.springmodules.commons.validator.DefaultValidatorFactory">
  <property name="validationConfigLocations">
    <list>
      <value>WEB-INF/validator-rules.xml</value>
      <value>WEB-INF/validation.xml</value>
    </list>
  </property>
</bean>
```

`DefaultValidatorFactory` is a class that loads the Commons Validator configuration on behalf of `DefaultBeanValidator`. The `validationConfigLocations` property takes a list of one or more validation configurations. Here we've asked it to load two configurations: validator-rules.xml and validation.xml.

The validator-rules.xml file contains a set of predefined validation rules for common validation needs such as email and credit card numbers. This file comes with the Commons Validator distribution, so you won't have to write it yourself—simply add it to the WEB-INF directory of your application. Table 13.3 lists all of the validation rules that come in validator-rules.xml.

Table 13.3 The validation rules available in Commons Validator's validator-rules.xml.

Validation rule	What it validates
`byte`	That the field contains a value that is assignable to byte
`creditCard`	That the field contains a `String` that passes a LUHN check and thus is a valid credit card number
`date`	That the field contains a value that fits a `Date` format
`double`	That the field contains a value that is assignable to `double`
`email`	That the field contains a `String` that appears to be an email address
`float`	That the field contains a value that is assignable to `float`
`floatRange`	That the field contains a value that falls within a range of `float` values
`intRange`	That the field contains a value that falls within a range of `int` values
`integer`	That the field contains a value that is assignable to `int`
`long`	That the field contains a value that is assignable to `long`
`mask`	That the field contains a `String` value that matches a given mask
`maxlength`	That the field has no more than a specified number of characters
`minlength`	That the field has at least a specific number of characters
`required`	That the field is not empty
`requiredif`	That the field is not empty, but only if another criterion is met
`short`	That the field contains a value that is assignable to `short`

The other file, validation.xml, defines application-specific validation rules that apply directly to the RoadRantz application. Listing 13.6 shows the contents of validation.xml as applied to RoadRantz.

Listing 13.6 Declaring validations in RoadRantz

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE form-validation PUBLIC
    "-//Apache Software Foundation//DTD
        Commons Validator Rules Configuration 1.1//EN"
    "http://jakarta.apache.org/commons/dtds/validator_1_1.dtd">

<form-validation>
  <formset>
    <form name="rant">
      <field property="rantText" depends="required">
```

Requires rant text

```
        <arg0 key="required.rantText" />
      </field>
      <field property="vehicle.state" depends="required">
        <arg0 key="required.state" />
      </field>
      <field property="vehicle.plateNumber"
          depends="required,mask">
        <arg0 key="invalid.plateNumber" />
        <var>
          <var-name>mask</var-name>
          <var-value>^[0-9A-Za-z]{2,6}$</var-value>
        </var>
      </field>
    </form>
  </formset>
</form-validation>
```

Requires vehicle state

Requires and masks plate number

If the contents of validation.xml look strangely familiar to you, it's probably because Struts uses the same validation file XML. Under the covers, Struts is using Commons Validator to do its validation. Now Spring Modules brings the same declarative validation to Spring.

One last thing to do is change the controller's declaration to wire in the new declarative implementation of `Validator`:

```
<bean id="addRantController"
      class="com.roadrantz.mvc.AddRantFormController">
  <property name="formView" value="addRant" />
  <property name="successView" value="rantAdded" />
  <property name="rantService" ref="rantService" />
  <property name="validator" ref="beanValidator" />
</bean>
```

A basic assumption with `SimpleFormController` is that a form is a single page. That may be fine when you're doing something simple such as adding a rant. But what if your forms are complex, requiring the user to answer several questions? In that case, it may make sense to break the form into several subsections and walk users through using a wizard. Let's see how Spring MVC can help you construct wizard forms.

13.3.3 Processing complex forms with wizards

Another feature of RoadRantz is that anyone can register as a user (known as a *motorist* in RoadRantz's terms) and be notified if any rants are entered for their vehicles. We developed the rant notification email in chapter 12. But we also need to provide a means for users to register themselves and their vehicles.

We could put the entire motorist registration form into a single JSP and extend `SimpleFormController` to process and save the data. However, we don't know how many vehicles the user will be registering and it gets tricky to ask the user for an unknown number of vehicle data in a single form.

Instead of creating one form, let's break motorist registration into several subsections and walk the user through the form using a wizard. Suppose that we partition the registration process questions into three pages:

- General user information such as first name, last name, password, and email address
- Vehicle information (state and plate number)
- Confirmation (for the user to review before committing their information)

Fortunately, Spring MVC provides `AbstractWizardFormController` to help out. `AbstractWizardFormController` is the most powerful of the controllers that come with Spring. As illustrated in figure 13.9, a wizard form controller is a special type of form controller that collects form data from multiple pages into a single command object for processing.

Let's see how to build a multipage registration form using `AbstractWizardFormController`.

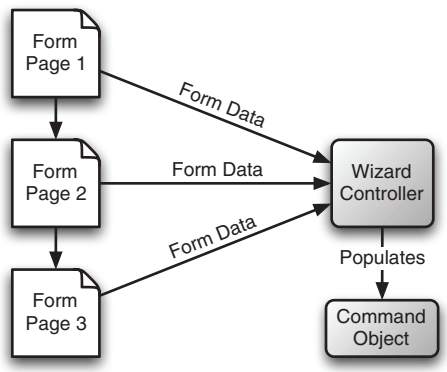


Figure 13.9
A wizard form controller is a special form controller that helps to split long and complex forms across multiple pages.

Building a basic wizard controller

To construct a wizard controller, you must extend the `AbstractWizardFormController` class. `MotoristRegistrationController` (listing 13.7) shows a minimal wizard controller to be used for registering a user in RoadRantz.

Listing 13.7 Registering motorists through a wizard

```
package com.roadrantz.mvc;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.validation.BindException;
import org.springframework.validation.Errors;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.
    ➥ AbstractWizardFormController;
import com.roadrantz.domain.Motorist;
import com.roadrantz.domain.Vehicle;
import com.roadrantz.service.RantService;

public class MotoristRegistrationController
    extends AbstractWizardFormController {
  public MotoristRegistrationController() {
    setCommandClass(Motorist.class);
    setCommandName("motorist");
  }

  protected Object formBackingObject(HttpServletRequest request)
      throws Exception {
    Motorist formMotorist = new Motorist();
    List<Vehicle> vehicles = new ArrayList<Vehicle>();
    vehicles.add(new Vehicle());
    formMotorist.setVehicles(vehicles);
    return formMotorist;
  }

  protected Map referenceData(HttpServletRequest request,
      Object command, Errors errors, int page) throws Exception {

    Motorist motorist = (motorist) command;
    Map refData = new HashMap();

    if(page == 1 && request.getParameter("_target1") != null) {
        refData.put("nextVehicle",
            motorist.getVehicles().size() - 1);
    }

    return refData;
  }

  protected void postProcessPage(HttpServletRequest request,
      Object command, Errors errors, int page) throws Exception {

    Motorist motorist = (Motorist) command;
```

Sets command class, name

Creates form backing object

Increments next vehicle pointer

```
    if(page == 1 && request.getParameter("_target1") != null) {
      motorist.getVehicles().add(new Vehicle());
    }
  }

  protected ModelAndView processFinish(HttpServletRequest request,
      HttpServletResponse response, Object command,
      BindException errors)
      throws Exception {

    Motorist motorist = (motorist) command;

    // the last vehicle is always blank...remove it
    motorist.getVehicles().remove(
        motorist.getVehicles().size() - 1);

    rantService.addMotorist(motorist);

    return new ModelAndView(getSuccessView(),
        "motorist", motorist);
  }

  // injected
  private RantService rantService;
  public void setRantService(RantService rantService) {
    this.rantService = rantService;
  }

  // returns the last page as the success view
  private String getSuccessView() {
    return getPages()[getPages().length-1];
  }
}
```

Adds new blank vehicle

Adds motorist

Just as with any command controller, you must set the command class when using a wizard controller. Here `MotoristRegistrationController` has been set to use `Motorist` as the command class. But because the motorist will also be registering one or more vehicles, the `formBackingObject()` method is overridden to set the `vehicles` property with a list of `Vehicle` objects. The list is also started with a blank `Vehicle` object for the form to populate.

Since the user can register any number of vehicles and since the vehicles list will grow with each vehicle added, the form view needs a way of knowing which entry in the list is the next entry. So, `referenceData()` is overridden to make the index of the next vehicle available to the form.

The only compulsory method of `AbstractWizardFormController` is `processFinish()`. This method is called to finalize the form when the user has finished completing it (presumably by clicking a Finish button). In `MotoristRegistra-`

tionController, processFinish() sends the data in the Motorist object to addMotorist() on the injected RantService object.

Notice there's nothing in MotoristRegistrationController that gives any indication of what pages make up the form or in what order the pages appear. That's because AbstractWizardFormController handles most of the work involved to manage the workflow of the wizard under the covers. But how does AbstractWizardFormController know what pages make up the form?

Some of this may become more apparent when you see how MotoristRegistrationController is declared in roadrantz-servlet.xml:

```
<bean id="registerMotoristController"
    class="com.roadrantz.mvc.MotoristRegistrationController">
  <property name="rantService" ref="rantService" />
  <property name="pages">
    <list>
      <value>motoristDetailForm</value>
      <value>motoristVehicleForm</value>
      <value>motoristConfirmation</value>
      <value>redirect:home.htm</value>
    </list>
  </property>
</bean>
```

So that the wizard knows which pages make up the form, a list of logical view names is given to the pages property. These names will ultimately be resolved into a View object by a view resolver (see section 13.4). But for now, just assume that these names will be resolved into the base filename of a JSP.

While this clears up how MotoristRegistrationController knows which pages to show, it doesn't tell us how it knows what order to show them in.

Stepping through form pages

The first page to be shown in any wizard controller will be the first page in the list given to the pages property. In the case of the motorist registration wizard, the first page shown will be the motoristDetailForm page.

To determine which page to go to next, AbstractWizardFormController consults its getTargetPage() method. This method returns an int, which is an index into the zero-based list of pages given to the pages property.

The default implementation of getTargetPage() determines which page to go to next based on a parameter in the request whose name begins with _target and ends with a number. getTargetPage() removes the _target prefix from the parameter and uses the remaining number as an index into the pages list. For

example, if the request has a parameter whose name is `_target2`, the user will be taken to the page rendered by the `motoristConfirmation` view.

Knowing how `getTargetPage()` works helps you to know how to construct your Next and Back buttons in your wizard's HTML pages. For example, suppose that your user is on the `motoristVehicleForm` page (index = 1). To create Next and Back buttons on the page, all you must do is create submit buttons that are appropriately named with the `_target` prefix:

```
<form method="POST" action="feedback.htm">
…
  <input type="submit" value="Back" name="_target0">
  <input type="submit" value="Next" name="_target2">
</form>
```

When the Back button is clicked, a parameter with its name, `_target0`, is placed into the request back to `MotoristRegistrationController`. The `getTargetPage()` method will process this parameter's name and send the user to the `motoristDetailForm` page (index = 0). Likewise, if the Next button is clicked, `getTargetPage()` will process a parameter named `_target2` and decide to send the user to the `motoristConfirmation` page (index = 2).

The default behavior of `getTargetPage()` is sufficient for most projects. However, if you would like to define a custom workflow for your wizard, you may override this method.

Finishing the wizard

That explains how to step back and forth through a wizard form. But how can you tell the controller that you have finished and that the `processFinish()` method should be called?

There's another special request parameter called `_finish` that indicates to `AbstractWizardFormController` that the user has finished filling out the form and wants to submit the information for processing. Just like the `_targetX` parameters, `_finish` can be used to create a Finish button on the page:

```
<form method="POST" action="feedback.htm">
…
  <input type="submit" value="Finish" name="_finish">
</form>
```

When `AbstractWizardFormController` sees the `_finish` parameter in the request, it will pass control to the `processFinish()` method for final processing of the form.

Unlike other form controllers, `AbstractWizardFormController` doesn't provide a means for setting the success view page. So, we've added a `getSuccessView()` method in `MotoristRegistrationController` to return the last page in the `pages` list. So, when the form has been submitted as finished, the `processFinish()` method returns a `ModelAndView` with the last view in the `pages` list as the view.

Canceling the wizard

What if your user is partially through with registration and decides that they don't want to complete it at this time? How can they abandon their input without finishing the form?

Aside from the obvious answer—they could close their browser—you can add a Cancel button to the form:

```
<form method="POST" action="feedback.htm">
…
  <input type="submit" value="Cancel" name="_cancel">
</form>
```

As you can see, a Cancel button should have `_cancel` as its name so that, when clicked, the browser will place a parameter into the request called `_cancel`. When `AbstractWizardFormController` receives this parameter, it will pass control to the `processCancel()` method.

By default, `processCancel()` throws an exception indicating that the cancel operation is not supported. So, you'll need to override this method so that it (at a minimum) sends the user to whatever page you'd like them to go to when they click Cancel. The following implementation of `processCancel()` sends the user to the success view:

```
protected ModelAndView processCancel(HttpServletRequest request,
    HttpServletResponse response, Object command,
    BindException bindException) throws Exception {

  return new ModelAndView(getSucessView());
}
```

If there is any cleanup work to perform upon a cancel, you could also place that code in the `processCancel()` method before the `ModelAndView` is returned.

Validating a wizard form a page at a time

As with any command controller, the data in a wizard controller's command object can be validated using a `Validator` object. However, there's a slight twist.

With other command controllers, the command object is completely populated at once. But with wizard controllers, the command object is populated a bit at a time as the user steps through the wizard's pages. With a wizard, it doesn't make much sense to validate all at once because if you validate too early, you will probably find validation problems that stem from the fact that the user isn't finished with the wizard. Conversely, it is too late to validate when the Finish button is clicked because any errors found may span multiple pages (which form page should the user go back to?).

Instead of validating the command object all at once, wizard controllers validate the command object a page at a time. This is done every time that a page transition occurs by calling the `validatePage()` method. The default implementation of `validatePage()` is empty (i.e., no validation), but you can override it to do your bidding.

To illustrate, on the `motoristDetailForm` page you ask the user for their email address. This field is optional, but if it is entered, it should be in a valid email address format. The following `validatePage()` method shows how to validate the email address when the user transitions away from the `motoristDetailForm` page:

```
protected void validatePage(Object command, Errors errors,
    int page) {

  Motorist motorist = (Motorist) command;
  MotoristValidator validator =
      (MotoristValidator) getValidator();

  if(page == 0) {
    validator.validateEmail(motorist.getEmail(), errors);
  }
}
```

When the user transitions from the `motoristDetailForm` page (index = 0), the `validatePage()` method will be called with 0 passed in to the `page` argument. The first thing `validatePage()` does is get a reference to the `Motorist` command object and a reference to the `MotoristValidator` object. Because there's no need to do email validation from any other page, `validatePage()` checks to see that the user is coming from page 0.

At this point, you could perform the email validation directly in the `validatePage()` method. However, a typical wizard will have several fields that will need to be validated. As such, the `validatePage()` method can become quite unwieldy. We recommend that you delegate responsibility for validation to a fine-grained field-level validation method in the controller's `Validator` object, as we've done here with the call to `MotoristValidator`'s `validateEmail()` method.

All of this implies that you'll need to set the `validator` property when you configure the controller:

```
<bean id="registerMotoristController"
    class="com.roadrantz.mvc.MotoristRegistrationController">
  <property name="rantService" ref="rantService" />
  <property name="pages">
    <list>
      <value>motoristDetailForm</value>
      <value>motoristVehicleForm</value>
      <value>motoristConfirmation</value>
      <value>redirect:home.htm</value>
    </list>
  </property>
  <property name="validator">
    <bean class="com.roadrantz.mvc.MotoristValidator" />
  </property>
</bean>
```

It's important to be aware that unlike the other command controllers, wizard controllers never call the standard `validate()` method of their `Validator` object. That's because the `validate()` method validates the entire command object as a whole, whereas it is understood that the command objects in a wizard will be validated a page at a time.

The controllers you've seen up until now are all part of the same hierarchy that is rooted with the `Controller` interface. Even though the controllers all get a bit more complex (and more powerful) as you move down the hierarchy, all of the controllers that implement the `Controller` interface are somewhat similar. But before we end our discussion of controllers, let's have a look at another controller that's very different than the others—the throwaway controller.

13.3.4 Working with throwaway controllers

One last controller that you may find useful is a throwaway controller. Despite the dubious name, throwaway controllers can be quite useful and easy to use. Throwaway controllers are significantly simpler than the other controllers, as evidenced by the `ThrowawayController` interface:

```
public interface ThrowawayController {
  ModelAndView execute() throws Exception;
}
```

To create your own throwaway controller, all you must do is implement this interface and place the program logic in the `execute()` method. Quite simple, isn't it?

But hold on. How are parameters passed to the controller? The execution methods of the other controllers are given `HttpServletRequest` and command

objects from which to pull the parameters. If the `execute()` method doesn't take any arguments, how can your controller process user input?

You may have noticed in figure 13.5 that the `ThrowawayController` interface is not even in the same hierarchy as the `Controller` interface. This is because throwaway controllers are very different from the other controllers. Instead of being given parameters through an `HttpServletRequest` or a command object, throwaway controllers act as their own command object. If you have ever worked with WebWork, this may seem quite natural because WebWork actions behave in a similar way.

From the requirements for RoadRantz, we know that we'll need to display a list of rants for a given month, day, and year. We could implement this using a command controller, as we did with `RantsForVehicleController` (listing 13.3). Unfortunately, no domain object exists that takes a month, day, and year. This means we'd need to create a special command class to carry this data. It wouldn't be so hard to create such a POJO, but maybe there's a better way.

Instead of implementing `RantsForDayController` as a command controller, let's implement it as a `ThrowawayController`, as shown in listing 13.8.

Listing 13.8 A throwaway controller that produces a list of rants for a given day

```
package com.roadrantz.mvc;
import java.util.Date;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.throwaway.
    ➥ ThrowawayController;
import com.roadrantz.service.RantService;

public class RantsForDayController implements ThrowawayController {
  private Date day;

  public ModelAndView execute() throws Exception {
    List<Rant> dayRants = rantService.getRantsForDay(day);

    return new ModelAndView("dayRants", "rants", dayRants);
  }

  public void setDay(Date day) {
    this.day = day;
  }

  private RantService rantService;
  public void setRantService(RantService rantService) {
    this.rantService = rantService;
  }
}
```

Gets list of rants for day

Binds day to request

Before `RantsForDayController` handles the request, Spring will call the `setDay()` method, passing in the value of the `day` request parameter. Once in the `execute()` method, `RantsForDayController` simply passes `day` to `rantService.getRantsForDay()` to retrieve the list of rants for that day. One thing that remains the same as the other controllers is that the `execute()` method must return a `ModelAndView` object when it has finished.

Just as with any controller, you also must declare throwaway controllers in the `DispatcherServlet`'s context configuration file. There's only one small difference, as you can see in this configuration of `RantsForDayController`:

```
<bean id="rantsForDayController"
    class="com.roadrantz.mvc.RantsForDayController"
    scope="prototype">
  <property name="rantService" ref="rantService" />
</bean>
```

Notice that the `scope` attribute has been set to `prototype`. This is where throwaway controllers get their name. By default all beans are singletons, and so unless you set `scope` to `prototype`, `RantsForDayController` will end up being recycled between requests. This means its properties (which should reflect the request parameter values) may also be reused. Setting `scope` to `prototype` tells Spring to throw the controller away after it has been used and to instantiate a fresh instance for each request.

There's just one more thing to be done before we can use our throwaway controller. `DispatcherServlet` knows how to dispatch requests to controllers by using a *handler adapter*. The concept of handler adapters is something that you usually don't need to worry about because `DispatcherServlet` uses a default handler adapter that dispatches to controllers in the `Controller` interface hierarchy.

But because `ThrowawayController` isn't in the same hierarchy as `Controller`, `DispatcherServlet` doesn't know how to talk to `ThrowawayController`. To make it work, you must tell `DispatcherServlet` to use a different handler adapter. Specifically, you must configure `ThrowawayControllerHandlerAdapter` as follows:

```
<bean id="throwawayHandler" class="org.springframework.web.
       ➥ servlet.mvc.throwaway.ThrowawayControllerHandlerAdapter"/>
```

By just declaring this bean, you are telling `DispatcherServlet` to replace its default handler adapter with `ThrowawayControllerHandlerAdapter`.

This is fine if your application is made up of nothing but throwaway controllers. But the RoadRantz application will use both throwaway and regular controllers alongside each other in the same application. Consequently, you still need

DispatcherServlet to use its regular handler adapter as well. Thus, you should also declare SimpleControllerHandlerAdapter as follows:

```
<bean id="simpleHandler" class="org.springframework.web.
       ➥ servlet.mvc.SimpleControllerHandlerAdapter"/>
```

Declaring both handler adapters lets you mix both types of controllers in the same application.

Regardless of what functionality your controllers perform, ultimately they'll need to return some results to the user. The result pages are rendered by views, which are selected by their logical name when creating a ModelAndView object. But there needs to be a mechanism to map logical view names to the actual view that will render the response. We'll see that in chapter 14 when we turn our attention to Spring's view resolvers.

But first, did you notice that all of Spring MVC's controllers have method signatures that throw exceptions? It's possible that things could go awry as a controller processes a request. If an exception is thrown from a controller, what will the user see? Let's find out how to control the behavior of errant controllers with an exception resolver.

13.4 Handling exceptions

There's a bumper sticker that says "Failure is not an option: it comes with the software." Behind the humor of this message is a universal truth. Things don't always go well in software. When an error happens (and it inevitably will happen), do you want your application's users to see a stack trace or a friendlier message? How can you gracefully communicate the error to your users?

SimpleMappingExceptionResolver comes to the rescue when an exception is thrown from a controller. Use the following <bean> definition to configure SimpleMappingExceptionResolver to gracefully handle any java.lang.Exceptions thrown from Spring MVC controllers:

```
<bean id="exceptionResolver" class="org.springframework.web.
       ➥ servlet.handler.SimpleMappingExceptionResolver">
  <property name="exceptionMappings">
    <props>
      <prop key="java.lang.Exception">friendlyError</prop>
    </props>
  </property>
</bean>
```

The exceptionMappings property takes a java.util.Properties that contains a mapping of fully qualified exception class names to logical view names. In this

case, the base `Exception` class is mapped to the `View` whose logical name is `friendlyError` so that if any errors are thrown, users won't have to see an ugly stack trace in their browser.

When a controller throws an `Exception`, `SimpleMappingExceptionResolver` will resolve it to `friendlyError`, which in turn will be resolved to a `View` using whatever view resolver(s) are configured. If the `InternalResourceViewResolver` from section 13.4.1 is configured then perhaps the user will be sent to the page defined in /WEB-INF/jsp/friendlyError.jsp.

13.5 Summary

The Spring Framework comes with a powerful and flexible web framework that is itself based on Spring's tenets of loose coupling, dependency injection, and extensibility.

At the beginning of a request, Spring offers a variety of handler mappings that help to choose a controller to process the request. You are given a choice to map URLs to controllers based on the controller bean's name, a simple URL-to-controller mapping, the controller class's name, or source-level metadata.

To process a request, Spring provides a wide selection of controller classes with complexity ranging from the very simple `Controller` interface all the way to the very powerful wizard controller and several layers in between, letting you choose a controller with an appropriate amount of power (and no more complexity than required). This sets Spring apart from other MVC web frameworks such as Struts and WebWork, where your choices are limited to only one or two `Action` classes.

All in all, Spring MVC maintains a loose coupling between how a controller is chosen to handle a request and how a view is chosen to display output. This is a powerful concept, allowing you to mix-'n'-match different Spring MVC parts to build a web layer most appropriate to your application.

In this chapter, you've been taken on a whirlwind tour of how Spring MVC handles requests. Along the way, you've also seen how most of the web layer of the RoadRantz application is constructed.

Regardless of what functionality is provided by a controller, you'll ultimately want the results of the controller to be presented to the user. So, in the next chapter, we'll build on Spring MVC by creating the view layer of the RoadRantz application. In addition to JSP, you'll learn how to use alternate template languages such as Velocity and FreeMarker. And you'll also learn how to dynamically produce non-HTML output such as Excel spreadsheets, PDF documents, and RSS feeds.