Building portlet applications

This chapter covers

- Creating portlet controllers
- Mapping portlet request to controllers
- Processing form submissions
- Handling exceptions

"Think outside the box!"

Despite this commonly touted advice, we often think *inside* the box. *Hollywood Squares*, your local post office, high school yearbooks, Sudoku puzzles, and the opening credits of *The Brady Bunch* are all places where we're forced to think in terms of boxes. How can we be expected to think outside of the box when we are constantly confronted with boxes?

Another place where we commonly think inside the box is with windows-based operating systems such as Microsoft Windows and Mac OS. If you've been around long enough, you'll remember that in the old days of MS-DOS, you could run only one program at a time. But ever since Microsoft Windows and other similar operating systems came along, we have been able to not only run multiple applications simultaneously, but also see them all on the screen at the same time.

In many ways, most web applications are like those old MS-DOS applications in that you can only access them one at a time. They consume the web browser's entire content pane. If you want to use a different web application, you must navigate away from the first application or open a separate browser window.

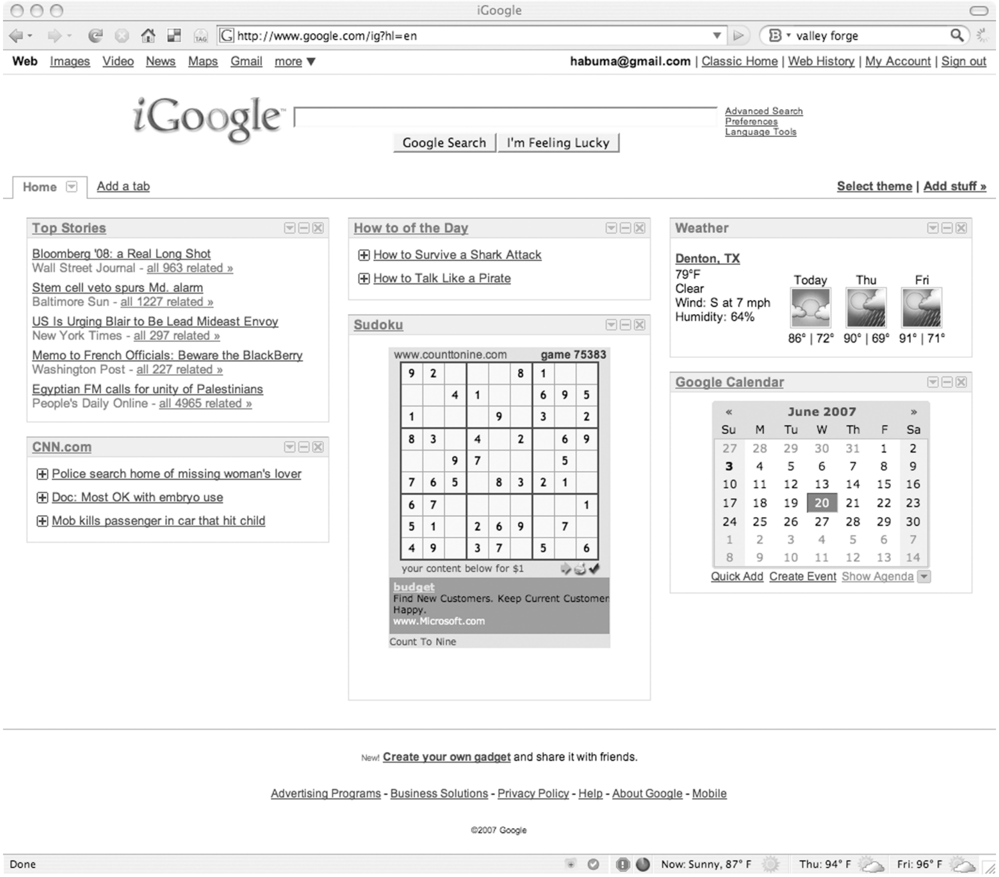
Then there are portals. Unlike conventional websites, it is possible to view several different applications at once on a single portal page. Each application is allotted a certain amount of browser real estate and is displayed alongside other applications on the same page.

Moreover, portals are often personalized according to each user's preferences, interests, and activity. This means that each user of a portal will get their own version of the portal page tailor-made for them.

For an example of a typical portal site, have a look at Google's iGoogle portal in figure P.1.

Notice that although figure P.1 is showing a single web page, there is a lot of information aggregated on that one page. There are two different sets of news headlines (Top Stories and CNN.com), a list of how-tos, a Sudoku game, a weather forecast, and a calendar. That's the whole point of portals: to collect multiple, possibly unrelated functionality and information in one convenient location. Otherwise, you might have to visit and potentially log into six different websites to get all of the information shown on this single page.

Although there are no distinct lines, it's not hard to imagine each section of the portal being contained within a box. These mini-applications within a portal page are commonly referred to as *portlets*.


iGoogle
http://www.google.com/ig?hl=en
valley forge
Web Images Video News Maps Gmail more ▼
habuma@gmail.com | Classic Home | Web History | My Account | Sign out
iGoogle
Advanced Search
Preferences
Language Tools
Google Search
I'm Feeling Lucky
Home
Add a tab
Select theme | Add stuff »

Top Stories

Bloomberg '08: a Real Long Shot
Wall Street Journal - all 963 related »

Stem cell veto spurs Md. alarm
Baltimore Sun - all 1227 related »

US Is Urging Blair to Be Lead Mideast Envoy
New York Times - all 297 related »

Memo to French Officials: Beware the BlackBerry
Washington Post - all 227 related »

Egyptian FM calls for unity of Palestinians
People's Daily Online - all 4965 related »

CNN.com

⊞ Police search home of missing woman's lover

⊞ Doc: Most OK with embryo use

⊞ Mob kills passenger in car that hit child

How to of the Day

⊞ How to Survive a Shark Attack

⊞ How to Talk Like a Pirate

Sudoku

www.counttonine.com game 75383

9	2				8	1		
		4	1			6	9	5
1				9		3		2
8	3		4		2		6	9
		9	7				5	
7	6	5		8	3	2	1	
6	7							1
5	1		2	6	9		7	
4	9		3	7		5		6

your content below for $1

budget
Find New Customers. Keep Current Customer
Happy.
www.Microsoft.com

Count To Nine

Weather

Denton, TX
79°F
Clear
Wind: S at 7 mph
Humidity: 64%

Today	Thu	Fri
86° | 72°	90° | 69°	91° | 71°

Google Calendar

« June 2007 »

Su	M	Tu	W	Th	F	Sa
27	28	29	30	31	1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
1	2	3	4	5	6	7
8	9	10	11	12	13	14

New! Create your own gadget and share it with friends.

Advertising Programs - Business Solutions - Privacy Policy - Help - About Google - Mobile
©2007 Google
Done
Now: Sunny, 87° F
Thu: 94° F
Fri: 96° F

Figure P.1 iGoogle is a typical example of a portal-based website, aggregating several functions on one single page.

You've probably already had a run-in or two with a portal-based site. In addition to iGoogle, one of the most commonly referred to portal examples on the Internet is My Yahoo! (http://my.yahoo.com). Corporate intranets are also often portal based. Even Amazon.com has some portal qualities if you look at it closely enough.

Portal-based websites have been around almost as long as the Web itself. But it wasn't until the Java Community Process produced JSR-168, the Portlet Specification, that there was a standard approach to building portlets. The Portlet Specification standardizes how portlets should be developed and deployed in Java. In short, it defines the contract between a portlet and a portal server that will host the portlets.

In this chapter, we're going to look at Spring Portlet MVC, an MVC framework geared toward building applications that live in boxes. More specifically, Spring Portlet MVC is used to build applications based on the Java Portlet API.

"Oh no…not **another** MVC framework!"

Before you close this chapter in disgust, hear me out. Although Spring Portlet MVC is a separate framework from Spring MVC, there's a lot of commonality shared between the two. In fact, as you'll soon find out, Spring Portlet MVC bears a striking resemblance to Spring MVC and even reuses some of Spring MVC's classes. This is good news because it means that you'll be able to leverage what you know about Spring MVC to develop Spring Portlet MVC applications. If you've already read chapters 13 and 14 of the book, you are well on your way to understanding how to build portlets with Spring Portlet MVC.

I'm going to assume that you already have a basic understanding of the Java Portlet Specification and know how to build basic portlets. If you are new to Java portlets or simply need a refresher, I suggest that you have a look at *Portlets and Apache Portals* (Manning, 2005) or *Building Portals with the Java Portlet API* (Apress, 2004). Both are excellent resources on building Java portlets.

P.1 Thinking inside the box

Even though many parallels can be drawn between Java's portlet specification and the servlet specification, portlets and servlets are about as different as apples and… well, some other kind of apple.

On the surface, the Java Portlet Specification mirrors the Java Servlet Specification in many ways:

- A servlet is written by either implementing the `javax.servlet.Servlet` interface or, more commonly, extending the abstract `javax.servlet.http.HttpServlet` class. Similarly, a portlet is written by either implementing the `javax.portlet.Portlet` interface or by extending the abstract `javax.portlet.GenericPortlet` class.
- When implementing `javax.servlet.Servlet`, the key processing method to implement is `service()`. When implementing `javax.portlet.Portlet`,

there are two processing methods to implement: `processAction()` and `render()`.

- If you choose to extend `javax.servlet.http.HttpServlet`, you may choose to implement `doGet()`, `doPost()`, `doPut()`, or `doDelete()` to process requests. With `javax.portlet.GenericPortlet`, you may choose to implement `doView()`, `doEdit()`, or `doHelp()` to process portlet requests.

However, just as there are many similarities, there are also several differences between servlets and portlets:

- A servlet's output is typically a web page that consumes the browser's entire content pane. Portlets, on the other hand, must share space on a web page with other portlets.
- A portlet can support several *modes*. Most of a portlet's functionality is presented in "view" mode. But a portlet may also provide an "edit" mode for configuring the portlet and a "help" mode to provide help information. In contrast, servlets don't have the concept of modes and are effectively always in view mode.
- The lifecycle of a portlet request is much more complex than that of a servlet. While a servlet only processes one type of request, portlets process both `ActionRequest`s and `RenderRequest`s. A portlet is only asked to process an `ActionRequest` if the user's action targets that particular portlet. But a portlet will always process `RenderRequest`s, even if the user is interacting with a different portlet.

These differences not only help explain why portlet applications need an MVC framework, but they also explain why Spring needs a separate MVC framework specifically for portlets.

P.1.1 Why portlets need MVC

While MVC frameworks such as Struts, WebWork, and Spring MVC provide a great deal of benefits when developing servlet-based web applications, it's very possible to construct a complex web application using nothing but servlets and no MVC framework at all. When a user clicks on a link or submits a form, the link's (or the form's) URL could be mapped to a servlet that handles the request. If the user wants to view a different page in the application, they just click on a link that takes them to a different servlet.

NOTE I'm not suggesting that you shouldn't use an MVC framework to build a web application—I'm only saying that you don't have to. Even though it's possible to build a web application based only on servlets, MVC frameworks simplify matters by centralizing common functionality such as security and internationalization.

Servlet-based applications can easily go from page to page because their view is rendered entirely in the browser. The browser isn't tied to any particular web page or servlet. Navigating from one servlet's URL to another replaces the one servlet's view with another.

For a moment, imagine that you've been asked to develop a function-rich servlet-based web application. While you're at it, imagine that you're only allowed to use one servlet to handle all of the application's functionality. The servlet may perform dozens of distinct functions, but you can never navigate away from the servlet. Seems fairly limiting, doesn't it?

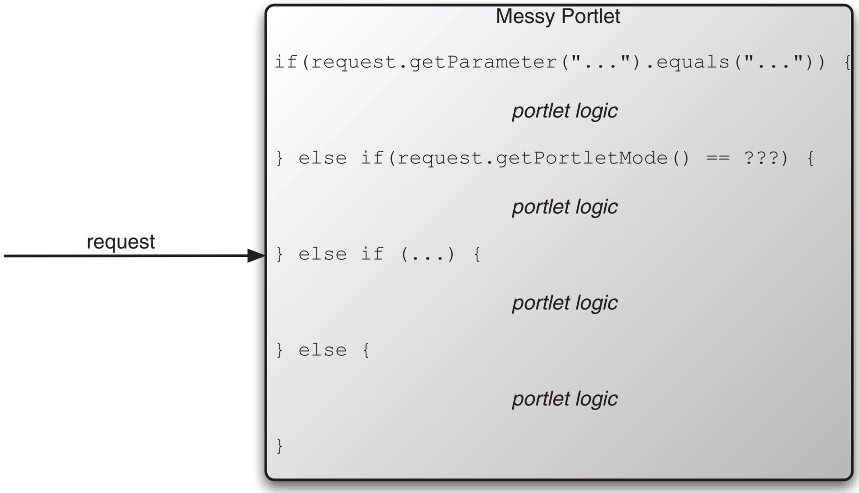
Portlet applications are similarly constrained. Unlike servlets that can be navigated to and from in a web browser, a portlet is a fixed component among many in a portal application's view. Each portlet is assigned to a specific space within the portal. While a portlet can render anything it wants in its assigned space, there's no way for a portlet to navigate to another portlet within that space.

If your portlet development plans only include simple portlets, such as a weather display portlet or an RSS viewer portlet, you won't find this chapter helpful. Portlets that have limited functionality and only display one or two views do not need an MVC framework. The portlet API's `Portlet` interface and `GenericPortlet` class will probably be sufficient for your needs. However, if your portlet's functionality is much more interesting than a "Hello World" application, you may want to consider using a portlet MVC framework.

Without an MVC framework, developing a feature-rich portlet application can be a daunting task. As shown in figure P.2, the portlet's `processAction()` and `render()` methods could easily become a twisted mess of `if/else if` and/or `switch` blocks that handle the various requests that are fielded by the portlet.

But when an MVC framework is applied to portlets, as shown in figure P.3, a single front controller portlet can handle virtually any request. The front controller portlet will handle all of the application's requests, then dispatch them to an appropriate controller to perform the actual business logic.

In short, portlet applications need an MVC framework to be able to handle complex functionality. But where can we find such a framework?



```
if(request.getParameter("...").equals("...")) {
                    portlet logic
} else if(request.getPortletMode() == ???) {
                    portlet logic
} else if (...) {
                    portlet logic
} else {
                    portlet logic
}
```

Figure P.2 Without an MVC framework, a portlet would be responsible for dispatching requests on its own.

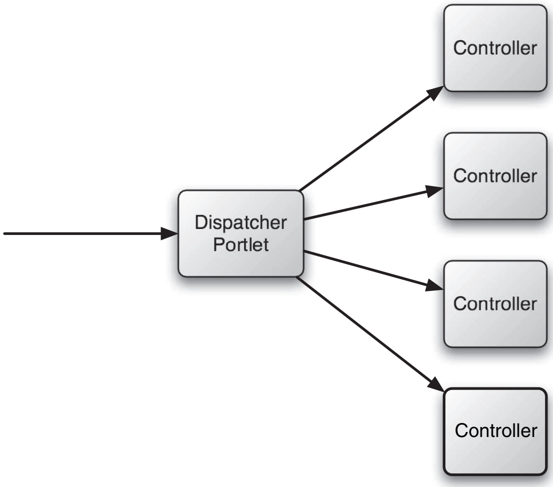


Figure P.3
Adding functionality to an MVC-based portlet application has no impact on its complexity.

P.1.2 Introducing Spring Portlet MVC

Recognizing the need for a portlet-based MVC framework, the Spring team created Spring Portlet MVC in Spring 2. Spring Portlet MVC is a Spring-based MVC framework specifically for building portlet applications. Using Spring Portlet MVC we are able to build function-rich web applications that work within the confines of a portlet box in a portal.

A day in the life of a portlet request

As we follow the path of a portlet request through a Spring Portlet MVC application, we find that it isn't much different from the path that a servlet request follows through a Spring MVC application. In fact, you'll find that figure P.4, which illustrates a portlet request's journey, is almost indistinguishable from figure 13.1 in chapter 13 of the book, which showed the course of a servlet request.

When the request is sent to the application from the portlet container, the first stop ❶ it makes is at `DispatcherPortlet`. `DispatcherPortlet` performs a very similar job to its servlet cousin, `DispatcherServlet`, by delegating responsibility for processing a request to controllers.

In order for `DispatcherPortlet` to know which controller to send the request to, it consults one or more handler mappings ❷. Portlet handler mappings are similar to Spring MVC handler mappings, except that they map portlet modes and parameters instead of URL patterns to controllers.

Once a suitable controller has been chosen, `DispatcherPortlet` sends the request straight away to the controller for processing ❸.

Here's where a portlet request's journey starts to vary from that of a servlet request. Remember that portlets process two different kinds of requests: action requests and render requests. If the request is an action request, then its journey

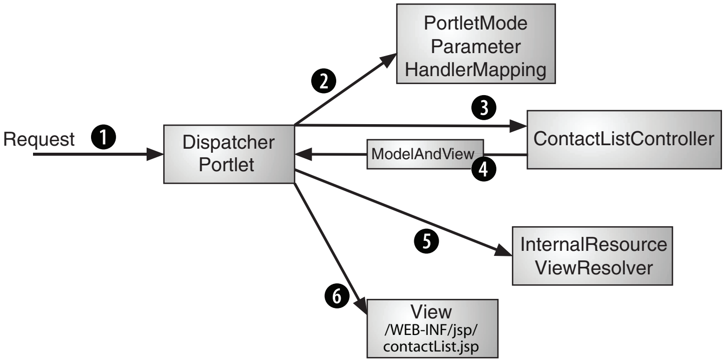


Figure P.4 `DispatcherPortlet` dispatches portlet requests to controllers and views, relying on handler mappings and view resolvers to guide its work.

is over once the controller completes its work. But if the request is a render request, it still has a few more stops to make before it can call it a day.

In the case of a render request, the controller will return a ModelAndView object ❹ back to DispatcherPortlet. This is the same ModelAndView that would be returned from a Spring MVC controller. It contains a logical name of a view to be rendered in the portlet box along with any model data that is to be displayed in the rendered view.

At this point, DispatcherPortlet is ready to send the request to a view implementation so that the results can be displayed in the portlet. But first, it must look up the actual view implementation by its logical view name by consulting a view resolver ❺. Since view resolution works pretty much the same, whether you're dealing with a conventional web application or a portlet application, Spring Portlet MVC is able to use any of the same view resolvers that work with Spring MVC.

The final stop for a render request is at the actual view implementation (probably a JSP). The view will use the model data contained in the request to produce output in the portlet's space within the portal page.

With this basic background information behind us, let's move forward and start creating a portlet application using Spring Portlet MVC.

P.2 Getting started with Spring Portlet MVC

Portals are all about putting the most useful information and applications together on one convenient web page. Very few pieces of information are nearly as useful as a list of friends, colleagues, and associates along with their contact information. Therefore, as a demonstration of Spring Portlet MVC, we're going to build a rolodex application. The Rolodex portlet will list a user's contacts and allow the user to add and edit contact information.

By now you've probably figured out that DispatcherPortlet is at the center of any Spring Portlet MVC application. Therefore, the first thing we'll need to do when building the Rolodex portlet is configure DispatcherPortlet.

P.2.1 Configuring DispatcherPortlet

When we were building the RoadRantz application using Spring MVC, we configured DispatcherServlet in web.xml. In Spring MVC, DispatcherServlet acts as a front controller, receiving all HTTP requests bound for the application, and then dispatches them to an appropriate controller for processing.

For portlet applications, Spring's Portlet MVC framework also has a front controller that will dispatch requests. As we've already discussed, however, portlets do not receive HTTP requests; they receive action and render requests from the portlet container. Therefore, instead of configuring a DispatcherServlet, we'll need to configure a front controller that will dispatch action and render requests.

DispatcherPortlet is Spring Portlet MVC's answer to Spring MVC's DispatcherServlet. DispatcherPortlet is itself a portlet that sits in front of a Spring portlet application, receiving portlet requests and dispatching them to Spring portlet controllers. Listing P.1 shows our Rolodex application's portlet.xml file, which contains a <portlet> entry for Spring's DispatcherPortlet.

Listing P.1 The Rolodex application's portlet.xml file

```
<portlet-app
    xmlns="http://java.sun.com/xml/ns/portlet/
        ➥ portlet-app_1_0.xsd"
    version="1.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-
        ➥ instance"
    xsi:schemaLocation="http://java.sun.com/xml/
        ➥ ns/portlet/portlet-app_1_0.xsd
                        http://java.sun.com/xml/
        ➥ ns/portlet/portlet-app_1_0.xsd">

  <portlet>
    <portlet-name>Rolodex</portlet-name>
    <portlet-class>
        org.springframework.web.portlet.
                ➥ DispatcherPortlet
    </portlet-class>
    <supports>
      <mime-type>text/html</mime-type>
      <portlet-mode>view</portlet-mode>
      <portlet-mode>edit</portlet-mode>
      <portlet-mode>help</portlet-mode>
    </supports>
    <portlet-info>
      <title>My Contacts</title>
      <short-title>My Contacts</short-title>
      <keywords>Contacts,Rolodex</keywords>
    </portlet-info>
    <portlet-preferences>
      <preference>
        <name>pageSize</name>
        <value>5</value>
      </preference>
    </portlet-preferences>
  </portlet>
</portlet-app>
```

Configures DispatcherPortlet

Sets portlet's modes

Declares pageSize preference

The <portlet> element in listing P.1 shows a fairly typical portlet configuration. The <supports> section describes the different modes that are supported by the portlet—in this case, view, edit, and help modes are supported.

Another point of interest is the <portlet-preferences> section. This is the section where we declare properties that can be customized by each portlet user. For our Rolodex portlet application, we've defined a pageSize preference, which will be used to specify how many Rolodex entries will be displayed on the screen at a time.

The most interesting parts of listing P.1 with regard to Spring Portlet MVC are the <portlet-class> and <portlet-name> entries. The <portlet-class> element is where we specify that the portlet in question is Spring's DispatcherPortlet. This is effectively the entry point into our Rolodex application.

As you'll remember from chapter 13 in the book, DispatcherServlet automatically loads its Spring context from a file whose name is based on its <servlet-name> entry in web.xml. Likewise, DispatcherPortlet will, by default, load its Spring context from a file whose name is based on its <portlet-name> entry in portlet.xml. In our example, the portlet's name is Rolodex; therefore, it will load its Spring context from a file named Rolodex-portlet.xml. This is the file where most of the beans we'll create in this chapter will be declared.

Setting up ViewRendererServlet

As you'll see in this chapter, Spring's portlet MVC framework closely resembles Spring's web MVC framework. Even though the two frameworks are quite similar, the differences between the Servlet API and Portlet API forced the Spring team to create portlet-specific versions of many of the Spring MVC classes.

But Spring MVC's view classes are a different story. For the most part, implementations of Spring MVC's ViewResolver and View will work fine with Spring Portlet MVC. The only thing is that those classes deal with ServletRequests, but portlet applications are in the business of working with PortletRequests.

Therefore, there's one last bit of infrastructural configuration we need to do before we can start developing our Spring portlet application. To bridge the gap between portlets and Spring MVC's view classes, we'll need to configure a ViewRendererServlet in the portlet application's web.xml file:

```
<servlet>
  <servlet-name>ViewRendererServlet</servlet-name>
  <servlet-class>
    org.springframework.web.servlet.
        ➥ ViewRendererServlet
  </servlet-class>
```

```
    <load-on-startup>1</load-on-startup>
  </servlet>
```

When DispatcherPortlet is ready to display the view, it will delegate to ViewRendererServlet to do its dirty work. By default, DispatcherPortlet will assume that ViewRendererServlet is mapped to /WEB-INF/servlet/view, so we'll need to also place the appropriate <servlet-mapping> in web.xml:

```
<servlet-mapping>
  <servlet-name>ViewRendererServlet</servlet-name>
  <url-pattern>/WEB-INF/servlet/view</url-pattern>
</servlet-mapping>
```

With DispatcherPortlet and ViewRendererServlet in place, the stage is set for us to start writing the Rolodex portlet application. With no further delay, let's set up our first portlet controller.

P.2.2 Creating your first portlet controller

When we first set out to create the web layer of the RoadRantz application, we started by building the RoadRantz home page. Similarly, the first portlet controller we'll create will be one that drives the main screen of the Rolodex application. ContactsController (listing P.2) retrieves a list of Contacts from a RolodexService to be displayed within the Rolodex portlet.

Listing P.2 A basic Spring portlet controller that lists contacts in a Rolodex

```
package com.springinaction.rolodex.controller;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import javax.portlet.RenderRequest;
import javax.portlet.RenderResponse;
import org.springframework.web.portlet.ModelAndView;
import org.springframework.web.portlet.mvc.AbstractController;
import com.springinaction.rolodex.service.RolodexService;

public class ContactsController
    extends AbstractController {

  protected
      ModelAndView handleRenderRequestInternal(
          RenderRequest request,                      Handles render
          RenderResponse response)                    request
          throws Exception {

    String userName =
        ControllerUtil.getUserName(request);
    List contacts =                                 Retrieves contacts
        rolodexService.getContacts(userName);       for user
    Map model = new HashMap();
    model.put("contacts", contacts);
    model.put("pageSize",
        request.getPreferences().getValue(
            "pageSize",
            PreferencesCommand.DEFAULT_PAGE_SIZE));    Returns
                                                        ModelAndView
    return new ModelAndView("contactList", model);
  }

  private RolodexService rolodexService;
  public void setRolodexService(
      RolodexService rolodexService) {                 Injects
    this.rolodexService = rolodexService;              RolodexService
  }
}
```

ContactsController extends AbstractController, which is the simplest of Spring's portlet controllers. Although its name is the same as its Spring MVC counterpart, this AbstractController class deals with portlet-specific requests.

When extending AbstractController, we can override either handleActionRequestInternal(), handleRenderRequestInternal(), or both. As you might guess, the handleActionRequestInternal() method is called during the action phase of the portlet lifecycle while handleRenderRequestInternal() is called during the render phase.

Because ContactsController will only be querying data to be displayed, there is no need to override the handleActionRequestInternal() method. Instead, handleRenderRequestInternal() is the method we need.

The first thing that handleRenderRequestInternal() does is look up the user's identity and use it to retrieve a list of Contacts from the injected RolodexService. The actual implementation of the RolodexService is not relevant to our discussion of Spring Portlet MVC. In fact, ContactsController only knows about the service through an interface. Therefore, any implementation of RolodexService will do and we'll leave it to the reader's imagination as to the details of how RolodexService is implemented.

In case you're curious, the code behind the ControllerUtil.getUserName() method is as follows:

```
public static String getUserName(
    PortletRequest request) {
  Principal userPrincipal =
      request.getUserPrincipal();

  return (userPrincipal == null) ?
```

```
         null :
         userPrincipal.getName();
}
```

Once the list of Contacts has been retrieved, it is placed into a Map that is used to hold model data to be displayed by the view. In addition to the list of contacts, the pageSize preference is also retrieved from the portlet's preferences so that the view will know how many contacts to show at one time.

Finally, the handleRenderRequestInternal() method creates a ModelAndView object containing the model Map with contactList as the view name. Again, although the ModelAndView class shares the same name as a similarly purposed class in Spring MVC, this ModelAndView is portlet specific. Be sure to choose the correct ModelAndView class when writing a portlet controller.

Configuring the controller bean

Now that we've written our first portlet controller class, it's time to configure it in the Spring application context. To do that, we place the following bit of XML in Rolodex-portlet.xml:

```
<bean id="contactsController"
    class="com.springinaction.rolodex.controller.
                 ➥ ContactsController">
  <property name="rolodexService"
      ref="rolodexService"/>
</bean>
```

The ContactsController uses a RolodexService to retrieve the list of contacts. Therefore, we must inject a reference to an implementation of RolodexService into ContactsController's rolodexService property.

We'll remain enigmatic about the actual identity of the rolodexService bean that is injected into ContactsController. It could be a local service bean. Or maybe it's an RMI service. Could it be a stateless session EJB? Who knows? The only thing that's important here is that it implements the RolodexService interface and is wired in Spring with the name rolodexService. (Isn't dependency injection fun?)

Declaring a view resolver

The last thing that ContactsController does is to return a ModelAndView object specifying a view with the name contactList. Ultimately, this view name will need to be resolved to an actual view implementation. This could be a Velocity or FreeMarker template, but in this case, it's a JSP.

As you saw in the previous chapter, the way that DispatcherServlet resolves a view name to find a JSP is through a view resolver. DispatcherPortlet is no different in this regard. For our example, I've chosen to use InternalResourceViewResolver:

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

We discussed this same InternalResourceViewResolver in chapter 13 of the book (section 14.1). Thanks to the ViewRendererServlet that we added to the web.xml file, we're able to use any of the view resolvers discussed in section 13.4 with Spring Portlet MVC. I chose InternalResourceViewResolver because it is simple.

As configured here, DispatcherPortlet will send the request to /WEB-INF/jsp/contactList.jsp (via ViewRendererServlet) to render the output of ContactsController.

Creating the JSP

The view for the Rolodex portlet's main view simply lists the contacts that are found for the user. The contactList.jsp file is listed in its entirety in listing P.3.

Listing P.3 contactList.jsp, which displays the list of contacts retrieved by ContactsController

```
<%@ taglib prefix="portlet"
    uri="http://java.sun.com/portlet" %>
<%@ taglib prefix="c"
    uri="http://java.sun.com/jstl/core_rt" %>
<%@ taglib prefix="display"
    uri="http://displaytag.sf.net/el" %>

<portlet:defineObjects/>

<%
  if(renderRequest.getUserPrincipal() != null) {
%>

<portlet:renderURL var="addUrl">            Creates
  <portlet:param name="action"              editContact
      value="editContact"/>                 URL
</portlet:renderURL>

<table width="100%">
  <tr><td align="right">
```

```
    <a href='<%= addUrl %>'>Add Contact</a>
  </td></tr>
</table>

<%
  }
%>

<display:table name="contacts"
    pagesize="${pageSize}" export="false"
    id="contact" style="width:100%" sort="list"
    defaultsort="1">
  <display:column sortable="true" title="Name">
    <c:out value="${contact.lastName}"/>,
    <c:out value="${contact.firstName}"/>
  </display:column>
  <display:column sortable="true" title="Phone">
    <c:out value="${contact.phone1}"/>
  </display:column>
  <display:column sortable="false" title="">
    <portlet:renderURL var="detailUrl">
      <portlet:param name="action"
          value="contactDetail"/>
      <portlet:param name="contactId"
          value="${contact.id}"/>
    </portlet:renderURL>
    <portlet:renderURL var="editUrl">
      <portlet:param name="action"
          value="editContact"/>
      <portlet:param name="contactId"
          value="${contact.id}"/>
    </portlet:renderURL>
    <portlet:actionURL var="deleteUrl">
      <portlet:param name="action"
          value="deleteContact"/>
      <portlet:param name="contactId"
          value="${contact.id}"/>
    </portlet:actionURL>
    <a href="${detailUrl}"><img src=
        "/Rolodex/images/view.gif" border="0"
        title="View contact details"></a>
    <c:if test="${not empty contact.ownerName}">
      <a href="${editUrl}"><img src=
          "/Rolodex/images/edit.gif" border="0"
          title="Edit contact"></a>
      <a href="${deleteUrl}"><img src=
          "/Rolodex/images/trash.gif" border="0"
          title="Delete contact"></a>
    </c:if>
  </display:column>
</display:table>
```

Creates contactDetail URL

Creates editContact URL

Creates deleteContact URL

The bulk of contactList.jsp is the `<display:table>` tag, which displays the list of contacts. `<display:table>` is the core JSP tag provided by the DisplayTag tag library. DisplayTag makes short work of rendering a collection of data in a table. It includes such features as sorting and pagination, and can even export table data to a Microsoft Excel spreadsheet. If you've never used DisplayTag before, you really should check it out at http://displaytag.sourceforge.net.

Draw your attention to the links that are created in contactList.jsp. The first link, near the top, is a link for the user to add a new contact. The URL for that link is created using the following `<portlet:renderURL>` tag:

```
<portlet:renderURL var="addUrl">
  <portlet:param name="action"
      value="editContact"/>
</portlet:renderURL>
```

URLs within a portal are a bit different than URLs within a conventional web application. Rather than hopping from one page to another, portal links usually change the state of the current portlet within the same portal page. Thus, these URLs must be encoded with portal-specific information. The `<portlet:renderURL>` tag is a standard portlet tag library tag that produces URLs appropriate for portlets running within a portal page.

Here, we're creating a link to the `editContact` action within the Rolodex portlet. Remember this action, because we're going to map it to a Spring portlet controller soon. You'll also find links being created for `contactDetail` and `deleteContact` actions within the `<display:table>` tag. These actions will also need to be mapped to Spring portlet controllers. As it turns out, the next thing we're going to do is map portlet requests to controllers.

P.3 Mapping requests to controllers

In a conventional web application, URLs are rather straightforward. Either a web page resides at a specified URL or it does not. As a result, mapping web requests to Spring MVC controllers is a simple matter of associating a URL pattern to a controller. Web requests have only one mode: the view mode.

Portlet requests aren't quite so one-dimensional. Portlets have the notion of modes, such as view, help, and edit. Each mode is almost like a subapplication within the main portlet application and can respond to individual actions that are mode specific. For example, within the Rolodex portlet, the Add Contact action is specific to the view mode. There's little point in responding to a Add Contact request from within the help mode, because Add Contact isn't a function of help.

Consequently, mapping portlet requests to controllers is a bit more complex than mapping web requests in Spring MVC. Portlet request mappings must not only consider the requested action, but also the mode within which the request was made.

Spring comes with three handler mappings that accommodate the idiosyncrasies of dealing with portlet requests. These handler-mapping classes (which are all in the `org.springframework.web.portlet.handler` package) are listed in table P.1.

Table P.1 Portlet handler mappings help `DispatcherPortlet` find the right portlet controller to handle a request.

Handler mapping	How it maps portlet requests to controllers
`ParameterHandlerMapping`	Maps requests to controllers by considering a parameter in the request.
`PortletModeHandlerMapping`	Maps the portlet's mode to a controller.
`PortletModeParameterHandlerMapping`	Combination of `ParameterHandlerMapping` and `PortletModeHandlerMapping`. Both a parameter and the portlet's mode are used as the key to finding a controller.

Which handler mapping you choose will depend largely on the complexity of your portlet application. If your portlet only has a single mode, like many controllers, `ParameterHandlerMapping` may be the most appropriate choice. On the other end of the spectrum, if your portlet supports several modes with only one controller per mode, you might want to consider `PortletModeHandlerMapping`.

`PortletModeParameterHandlerMapping` is the most flexible of Spring's portlet handler mappings. It combines the parameter mapping capability of `ParameterHandlerMapping` with the mode mapping capability of `PortletModeHandlerMapping`.

For the Rolodex portlet, we'll use a combination of `PortletModeHandlerMapping` and `PortletModeParameterHandlerMapping`. The edit and help modes have simple needs, having a single controller class for each. This makes `PortletModeHandlerMapping` sufficient for those modes. The edit mode, on the other hand, is a bit more complex and will need the more capable mapping features of `PortletModeParameterHandlerMapping`.

But before we get too carried away with those handler mappings, let's take a quick look at how we might configure a `ParameterHandlerMapping`.

P.3.1 Mapping portlet parameters to controllers

Let's pretend for a moment that the Rolodex portlet only supports the view mode. With no need for the help and edit modes, we can focus our attention on mapping the controllers that make up the view mode.

When a portlet application only supports a single mode, `ParameterHandlerMapping` is the best choice for mapping requests to controllers. `ParameterHandlerMapping` decides which controller should receive the request by considering a parameter in the portlet request.

If we were to use `ParameterHandlerMapping` for the Rolodex portlet application, we could declare it in Spring like this:

```
<bean id="parameterHandlerMapping"
    class="org.springframework.web.portlet.handler.
                ➥ ParameterHandlerMapping">
  <property name="parameterMap">
    <map>
      <entry key="contacts"
             value-ref="contactsController"/>
      <entry key="editContact"
             value-ref="editContactController"/>
      <entry key="contactDetail"
             value-ref="contactDetailController"/>
      <entry key="deleteContact"
             value-ref="deleteContactController"/>
      <entry key="searchContacts"
             value-ref="searchContactsController"/>
    </map>
  </property>
  <property name="interceptors">
    <list>
      <ref bean="parameterMappingInterceptor"/>
    </list>
  </property>
</bean>
```

The mapping is defined in the `parameterMap` property. Here we've mapped five different parameter values to five different controllers. But where does the parameter values come from?

Take a moment and look back at the contactList.jsp file in Listing P.3. As we've already discussed, links within the portlet page are created using the `<portlet:renderURL>` tag. For example, the link for adding a new contact is created like this:

```
<portlet:renderURL var="addUrl">
  <portlet:param name="action"
      value="editContact"/>
</portlet:renderURL>
```

The main thing to pay attention to is the <portlet:param> tag contained within the <portlet:renderURL> tag. In this case, we've specified that the URL should have a parameter named action with a value of editContact. This is the parameter that ParameterHandlerMapping uses when looking up a controller.

In the case of the Add Contact link, the action parameter is set to editContact. Looking up the editContact parameter in ParameterHandlerMapping's parameterMap property, we see that this link takes the user to the controller whose bean name is editContactController.

Forwarding the action parameter to the RenderRequest

Pay special attention to how the interceptors property has been wired. The interceptors property is used to associate portlet handler interceptors (any implementation of Spring's org.springframework.web.portlet.HandlerInterceptor interface) to be invoked around the invocation of controllers. This is similar in concept to servlet filters or even aspects.

In this case, we're wiring a very important handler interceptor to ParameterHandlerMapping. Remember that portlets have two phases in their lifecycle: Action and Render. The action parameter that is sent in the URL created by <portlet:renderURL> only goes into the ActionRequest by default.

But ParameterMappingInterceptor makes sure that the parameter used by ParameterHandlerMapping makes it to the RenderRequest so that RenderRequests are mapped to controllers properly. ParameterMappingInterceptor is configured in Spring as follows:

```
<bean id="parameterMappingInterceptor"
    class="org.springframework.web.portlet.handler.
            ➥ ParameterMappingInterceptor"/>
```

Using a different mapping parameter

By default, ParameterHandlerMapping examines the value of a parameter named action. If for some reason that won't work for you (maybe your portal server already uses action for some other purpose), you can configure ParameterHandlerMapping to use a different parameter. For example, the following ParameterHandlerMapping declaration uses the parameterName property to specify that the mapping parameter should be called doThis:

```
<bean id="parameterHandlerMapping"
    class="org.springframework.web.portlet.handler.
               ➥ ParameterHandlerMapping">
  <property name="parameterMap">
     …
  </property>
```

```
    <property name="parameterName" value="doThis" />
  </bean>
```

If you decide to change the mapping parameter name, be sure to remember to make corresponding changes in the JSP files:

```
<portlet:renderURL var="addUrl">
  <portlet:param name="doThis"
      value="editContact"/>
</portlet:renderURL>
```

You'll also need to make sure to tell `ParameterMappingInterceptor` that you've changed the name of the mapping parameter:

```
<bean id="parameterMappingInterceptor"
    class="org.springframework.web.portlet.handler.
              ➥ ParameterMappingInterceptor">
  <property name="parameterName" value="doThis" />
</bean>
```

Mapping portlet request parameters to controllers is easy enough, but it doesn't address more complex portlet needs. What if a portlet supports multiple modes? When modes are involved, we'll need to consider using `PortletModeHandlerMapping`.

P.3.2 Mapping portlet modes to controllers

As configured in the portlet.xml file (listing P.1), the Rolodex application's `DispatcherPortlet` will support three modes: view, edit, and help. The `view` mode is the main mode of the portlet, and we'll see how to map its controllers in the next section. For now, we'll need to map default controllers for each of the modes supported by the portlet. As we're mapping portlet modes to controllers, this sounds like a job for `PortletModeHandlerMapping`.

`PortletModeHandlerMapping`, as you might guess, works by mapping a portlet's mode name to a controller. Here's what it looks like in Spring as configured for the Rolodex portlet's modes:

```
<bean id="portletModeHandlerMapping"
    class="org.springframework.web.portlet.handler.
               ➥PortletModeHandlerMapping">
  <property name="order" value="2" />
  <property name="portletModeMap">
    <map>
      <entry key="view"
          value-ref="contactsController"/>
      <entry key="help"
          value-ref="modeNameViewController"/>
```

```
        <entry key="edit"
            value-ref="preferencesController"/>
      </map>
    </property>
  </bean>
```

At a glance, this `<bean>` declaration doesn't look much different from the one we defined for `ParameterNameHandlerMapping`. This time, however, we're mapping portlet mode names to controllers in the `portletModeMap` property. As declared here, the help mode's default controller is the one whose bean name is `modeNameViewController`. Likewise, when the user enters the edit mode, the request will be sent to the controller whose name is `preferencesController`.

When we first view the portlet in view mode, there will be no `action` parameter in the request. Therefore, we'll need to map a default controller for the view mode. Here we've mapped it to the `contactsController` bean, which is the `ContactsController` that we wrote in listing P.2.

Even though we'll map the controllers for the view mode in the next section using `PortletModeParameterHandlerMapping`, we still need to define the default controller for the view mode.

You have probably also noticed that we've set the `order` property to 2. The purpose of the order property will become apparent in a moment. First, however, we need to map the rest of the controllers that make up the Rolodex portlet application. Let's see how `PortletModeParameterHandlerMapping` can be used to map portlet parameters within the view mode to the controllers that handle their requests.

P.3.3 Mapping both modes and parameters to controllers

`PortletModeParameterHandlerMapping` is a functional blend of `ParameterHandlerMapping` and `PortletModeHandlerMapping`. Where `ParameterHandlerMapping` maps parameter values to controllers and `PortletModeHandlerMapping` maps portlet modes to controllers, `PortletModeParameterHandlerMapping` maps parameter values to controllers within the context of a given portlet mode.

For example, consider the following declaration of `PortletModeParameterHandlerMapping`. The core of the Rolodex portlet's functionality takes place within the view mode. That being so, this `PortletModeParameterHandlerMapping` maps parameter values within the view mode:

```
  <bean id="portletModeParameterHandlerMapping"
      class="org.springframework.web.portlet.handler.
          ➥ PortletModeParameterHandlerMapping">
    <property name="order" value="1" />
    <property name="interceptors">
      <list>
        <ref bean="parameterMappingInterceptor"/>
      </list>
    </property>
    <property name="portletModeParameterMap">
      <map>
        <entry key="view">
          <map>
            <entry key="contacts"
                value-ref="contactsController"/>
            <entry key="editContact"
                value-ref="editContactController"/>
            <entry key="contactDetail"
                value-ref="contactDetailController"/>
            <entry key="deleteContact"
                value-ref="deleteContactController"/>
            <entry key="searchContacts"
                value-ref="searchContactsController"/>
          </map>
        </entry>
        <entry key="edit">
          <map/>
        </entry>
        <entry key="help">
          <map/>
        </entry>
      </map>
    </property>
  </bean>
```

The portletModeParameterMap property is where the mapping is defined for PortletModeParameterHandlerMapping. The first thing you'll observe is that this property is a bit more complex than the corresponding properties for ParameterHandlerMapping and PortletModeHandlerMapping. Instead of a simple name-value pair, the portletModeParameterMap property takes a <map> of <map>s.

Each <entry> in the outer <map> defines the mappings for each of the portlet's supported modes. Meanwhile the inner <map> entries map parameter values to controller bean names in a fashion similar to ParameterHandlerMapping. The difference here is that the parameter-to-controller mappings are only applicable within the portlet mode that they're mapped to.

For the Rolodex portlet's view mode, we've mapped the same parameters and controllers as we did in the ParameterHandlerMapping example. We're letting PortletModeHandlerMapping handle the edit and help modes, so we've given them an empty <map>.

You'll notice that we also had to wire in a reference to `ParameterHandlerInterceptor` to the `interceptors` property. As was the case with `ParameterHandlerMapping`, this handler interceptor will make sure that the mapping parameter is copied into the `RenderRequest` so that it will be properly mapped to a controller.

Chaining portlet handler mappings

When using `PortletModeHandlerMapping` along with `PortletModeParameterHandlerMapping`, we need to make sure that `PortletModeParameterHandlerMapping` gets first crack at deciding which controller to send requests to. If not, `PortletModeHandlerMapping` will always decide on `ContactsController`, regardless of the value of the mapping parameter.

That's why we set the `order` property on both of the handler mappings. In the case of `PortletModeParameterHandlerMapping`, the order property is set to 1 to ensure that it is called first. If `PortletModeParameterHandlerMapping` fails to determine which controller to send a portlet request to then `PortletModeHandlerMapping`, whose `order` is set to 2, will get a chance.

The actual values assigned to the `order` properties of each handler mapping aren't important. The only thing that's important is that the value assigned to the `order` property of `PortletModeParameterHandlerMapping` is lower than that assigned to the `order` property of `PortletModeHandlerMapping`. In short, the handler mapping with the lowest `order` is given first shot at mapping requests to controllers.

P.4 Handling portlet requests with controllers

As I write this, I'm sitting in a small Mexican food restaurant within walking distance of my office. It's my lunch hour and I'm enjoying a delicious beef and cheese burrito with a side of rice and beans. I've found that my work can be very productive if I have a plate of spicy food to munch on while I write.

When I walked into the restaurant and approached the counter, I asked the nice woman behind the counter for the #4 lunch special. She took my order to be processed while I found a small table in the corner to work at. Moment later, she emerged from the back with my food.

Although she took my order and brought it out to me, I foster no illusion that she was the one who prepared the food. After she jotted down the order on the order pad, she dispatched it to one of the cooks in the back to prepare. Once they were done, she carried the resulting dish out to my table.

Just as the woman behind the counter didn't process the request for a burrito, `DispatcherPortlet` doesn't directly process portlet requests. It simply accepts the request and then dispatches it to a controller to "cook" up a result.

Just like Spring MVC, Spring Portlet MVC comes with a rich selection of controllers suitable for handling requests in a portlet application. Figure P.5 shows all of Spring's portlet controllers and how they relate to each other.

At a glance, you might think that figure P.5 is a duplicate of figure 13.6 from chapter 13 of the book. However, take a closer look. Indeed, there are some familiar names, such as `Controller`, `AbstractCommandController` and `SimpleForm-`

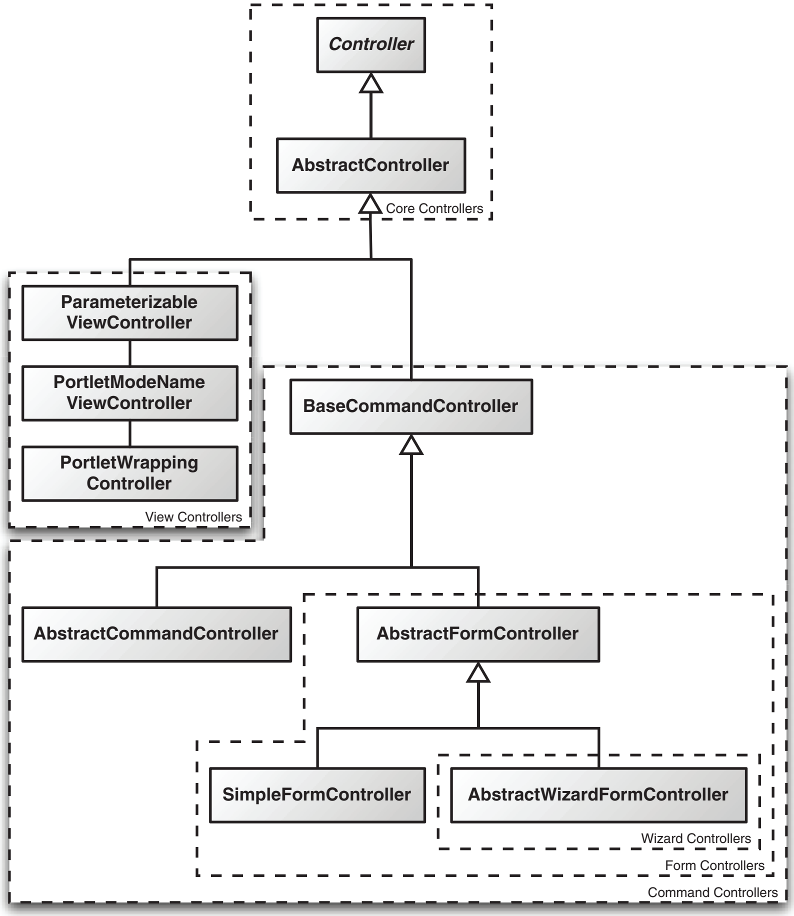


Figure P.5 Spring's selection of portlet controllers closely mirros Spring MVC's controllers for servlet-based web applications.

Controller. There's even an AbstractWizardFormController. Looking even closer, you'll find a couple of new controllers in there as well.

However familiar they may seem, the controllers in figure P.5 are slightly different from those you learned about in chapter 13. Where the controllers from that chapter were based on the Servlet API, these controllers are based on the Portlet API.

Nevertheless, one of the great things about Spring's portlet MVC framework is that it mostly mirrors Spring's web MVC framework. This means that if you're familiar with the web framework, you're well on your way to understanding the portlet framework.

Although Spring's portlet MVC framework is similar to Spring's web MVC framework, it may be worthwhile to review what each of the controllers in figure P.5 do. Table P.2 briefly describes each controller.

You've already seen an example of how to use AbstractController to display the main page of the Rolodex portlet application. Now we'll look at a few more of Spring's portlet controllers as we flesh out much of the functionality of the Rolodex application. We won't have opportunity to use all of the controllers in table P.2, but we'll use enough of them to get a taste of how Spring's portlet controllers compare to their Spring MVC counterparts.

Table P.2 Examining Spring's portlet controllers.

Controller type	Classes	Useful when...
Simple	Controller (interface) AbstractController	Your controller is extremely simple and does not require parameter binding or form-processing capabilities.
View	ParameterizableViewController PortletModeNameViewController PortletWrappingController	Your controller performs no processing whatsoever and only needs to display a simple view.
Command	BaseCommandController AbstractCommandController	Your controller needs parameters to be bound to a command object and (optionally) validated.
Form	AbstractFormController SimpleFormController	Your controller needs to display a form and subsequently process the submission of that form.
Wizard	AbstractWizardFormController	You want your controller to walk the user through a complex series of form pages that ultimately are processed as a single form submission.

To get started, let's look at simplest of all of Spring's portlet controllers, PortletModeNameViewController, and see how to use it to add a help page to the application.

P.4.1 Displaying mode-specific pages

When we declared the PortletModeHandlerMapping bean, we mapped the help mode to a bean named modeNameViewController. That bean is declared as follows:

```
<bean id="modeNameViewController"
    class="org.springframework.web.portlet.mvc.
                ➥ PortletModeNameViewController"/>
```

PortletModeNameViewController is a simple portlet controller that simply returns a ModelAndView whose logical view name is set to the name of the portlet mode. Since we've declared this controller to be the target of the help mode, the logical view name will be help. If we're using the InternalResourceViewResolver from section P.2.3, then the view will be found in /WEB-INF/jsp/help.jsp.

Many of the controllers that you'll use in a Spring portlet application require you to subclass an abstract controller class and write code that defines the controller's functionality. PortletModeNameViewController, however, is completely self-contained. Just wire it up and it's ready to go. No coding required!

Not all portlet controllers are so simple that they only need to display a view. In any interesting portlet application, there will be controllers that take parameters as input to perform their functionality. For more complex controller needs, we'll need to create a command controller.

P.4.2 Processing portlet commands

Portlet command controllers are similar in purpose to the web command controllers we discussed in chapter 13, section 13.3.1. Just like their web MVC counterparts, portlet command controllers automatically copy request parameter values into a command object for processing. This frees you from having to deal directly with the portlet's ActionRequest and RenderRequest objects.

To illustrate how to write a portlet command controller, consider SearchContactsController in listing P.4. SearchContactsController extends the portlet version of AbstractCommandController to automatically copy request parameters into a SearchCommand object.

Listing P.4 A command controller for searching through the Rolodex

```
package com.springinaction.rolodex.controller;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import javax.portlet.ActionRequest;
import javax.portlet.ActionResponse;
import javax.portlet.RenderRequest;
import javax.portlet.RenderResponse;
import org.springframework.validation.
           ➥ BindException;
import org.springframework.web.portlet.
           ➥ ModelAndView;
import org.springframework.web.portlet.mvc.
           ➥ AbstractCommandController;
import com.springinaction.rolodex.service.
           ➥ RolodexService;

public class SearchContactsController
    extends AbstractCommandController {

  public SearchContactsController() {
    setCommandClass(SearchCommand.class);
  }

  protected void handleAction(
      ActionRequest request,
      ActionResponse response,
      Object command,
      BindException bindException)
      throws Exception { }
```

Does nothing for action requests

```
  protected ModelAndView handleRender(
      RenderRequest request,
      RenderResponse response,
      Object command,
      BindException bindException)
      throws Exception {

    SearchCommand searchCommand =
        (SearchCommand) command;

    String userName =
        ControllerUtil.getUserName(request);

    List<Contact> contacts = rolodexService.
        searchContacts(userName, searchCommand);
```

Looks up contact list

```
    Map model = new HashMap();
    model.put("contacts", contacts);
    model.put("pageSize",
        request.getPreferences().getValue(
            "pageSize", "5"));
```

Sets model data

```
    return new ModelAndView(
        "searchResults", model);          Returns ModelAndView
  }

  private RolodexService rolodexService;
  public void setRolodexService(
      RolodexService rolodexService) {
    this.rolodexService = rolodexService;       Injects RolodexService
  }
}
```

When we developed a web command controller, we overrode AbstractCommandController's handle() method to process an HttpServletRequest. With conventional web requests, there is only one type of request; thus, there is only one handle() method to implement a Spring MVC's AbstractCommandController.

But portlet requests aren't so simple. A portlet controller could end up handing two different portlet requests: an ActionRequest if the portlet is the target of an action URL and a RenderRequest every time that the portlet needs to render output. Consequently, the portlet version of AbstractCommandController has two methods that must be overridden: handleAction() and handleRender().

Both of these methods are abstract and must be implemented in any class that extends AbstractCommandController. In the case of SearchContactsController, however, there is no need for action request processing. Therefore, the handleAction() method is left empty. The handleRender() method is where all of the functionality of SearchContactsController takes place.

SearchContactsController starts by casting the command object that it receives to SearchCommand, the actual command class. The command class is specified in the constructor in the call to setCommandClass().

As for SearchCommand, it's a simple POJO with two properties: firstName and lastName. These properties can be used to search the Rolodex.

```
package com.springinaction.rolodex.controller;

public class SearchCommand {
  private String firstName;
  private String lastName;

  public SearchCommand() {}

  public String getFirstName() {
    return firstName;
  }

  public void setFirstName(String firstName) {
    this.firstName = firstName;
    }

    public String getLastName() {
      return lastName;
    }

    public void setLastName(String lastName) {
      this.lastName = lastName;
    }
  }
```

Once SearchContactsController has a reference to a SearchCommand object, it looks up the current portlet user's name. Then it retrieves a list of matching Contacts by passing the username and the SearchCommand object to the searchContacts() method of the injected RolodexService. Finally, it places the list of Contacts in the model Map along with the value of the pageSize preference and returns a ModelAndView object.

As with any Spring portlet controller, we need to declare a <bean> entry in Spring. The following XML in Rolodex-portlet.xml should do the trick:

```
<bean id="searchContactsController"
    class="com.springinaction.rolodex.controller.
                ➥ SearchContactsController">
  <property name="rolodexService"
      ref="rolodexService"/>
</bean>
```

Because SearchContactsController uses a RolodexService to perform the actual Contact search, we must wire a reference to the rolodexService bean into the rolodexService property. (And yes, we're still keeping the identity of the rolodexService bean to ourselves. It still isn't relevant to the discussion of Spring portlets, and in fact, it could be anything that implements the RolodexService interface.)

Command controllers are a wonderful way to project request parameters onto an object for simplified processing. When those parameters are coming from a form submission, however, Spring offers an even better controller option. To wrap up our discussion of Spring's portlet controllers, let's see how to use a form controller to add and edit contacts in the Rolodex.

P.4.3 Processing form submissions

Aside from simply listing contacts, one of the primary functions of the Rolodex portlet is the ability for the user to add and edit contacts in the Rolodex. To provide this functionality, we must first show the user a form for them to enter the contact information. Then, upon submission of that form, we'll need a controller that will save the contact information.

As you'll recall, Spring's form controllers pull double duty by both displaying a form (upon an HTTP GET request) and processing the form (upon an HTTP POST request). You'll find that Spring's portlet form controllers offer the same behavior, only within a portlet application. This makes a form controller the perfect choice for implementing the add/edit functionality. Listing P.5 shows such a form controller for adding and editing contact information.

Listing P.5 A form controller for adding and editing contacts in the Rolodex

```
package com.springinaction.rolodex.controller;
import javax.portlet.ActionRequest;
import javax.portlet.ActionResponse;
import javax.portlet.PortletException;
import javax.portlet.PortletRequest;
import javax.portlet.RenderRequest;
import javax.portlet.RenderResponse;
import org.apache.commons.lang.StringUtils;
import org.springframework.validation.BindException;
import org.springframework.web.portlet.bind.PortletRequestUtils;
import org.springframework.web.portlet.mvc.SimpleFormController;
import org.springframework.web.servlet.ModelAndView;
import com.springinaction.rolodex.domain.Contact;
import com.springinaction.rolodex.service.RolodexService;

public class EditContactController
    extends SimpleFormController {

  public EditContactController() {
    setCommandClass(Contact.class);
  }

  protected Object formBackingObject(
      PortletRequest request) throws Exception {

    int contactId = PortletRequestUtils.getIntParameter(
        request, "contactId", -1);

    Contact contact =
        (contactId < 0) ?
        new Contact() :
        rolodexService.getContact(contactId);

    if(contact == null) {
      throw new PortletException("Contact not found");
    }

  }

  protected void processFormSubmission(
```

Creates or looks up contact

```
        ActionRequest request,
        ActionResponse response, Object command,
        BindException bindException)
        throws Exception {

      Contact contact = (Contact) command;

      String userName =
          ControllerUtil.getUserName(request);

      rolodexService.addContact(contact, userName);

      response.setRenderParameter(
          "action", "contacts");
    }

    // injected
    private RolodexService rolodexService;
    public void setRolodexService(
        RolodexService rolodexService) {
      this.rolodexService = rolodexService;
    }
  }
```

Saves contact

Of all of Spring's portlet controllers, `SimpleFormController` is the one that most closely resembles its conventional web counterpart. In fact, the only sign that this is a portlet controller is that the `processFormSubmission()` method takes an `ActionRequest` and an `ActionResponse`.

The action-phase `processFormSubmission()` is where most of the action (no pun intended) happens in this controller. This method will be called when the form is submitted with a POST request. The first thing it does is cast the command object to `Contact`, as that is the actual command class that is specified in the controller. Then it uses the `addContact()` method of the injected `RolodexService` to add the `Contact` to the Rolodex.

Redirecting portlet views

The very last thing that `processFormSubmission()` does is to set a render parameter in the response. This behavior is a bit unusual and deserves some explanation.

Upon a successful submission of the form, we want the user to be presented with the list of contacts in the Rolodex. Since we already have a perfectly good controller for that, `ContactsController`, there's no reason to re-create that functionality in `EditContactController`. Instead, it's better to simply redirect the request to `ContactsController`.

If the Rolodex application were a Spring MVC application, we could simply return a `ModelAndView` object whose view is `redirect:/contacts.html`. This would force the request to be redirected to `ContactsController`, where the contact list would be rendered.

Unfortunately, portlets don't support the notion of redirect. There's no clear way to redirect a portlet request from one controller to another. Thus, the last instruction in `processFormSubmission()` implements a common trick that simulates a forward. Here's how it works…

Remember that portlet requests go through two phases: the action phase and the render phase. As it turns out, portlet requests are mapped to controllers twice, once for each phase. At the point when the action phase `processFormSubmission()` method is called, the render request still hasn't been mapped to a controller. By setting the render request's `action` parameter to `contacts`, we have effectively changed the fate of the render request.

As a result, even though the action request was mapped to `EditContactController` because its `action` parameter was set to `editContact`, the render request will be mapped to `ContactsController`, because we've changed the `action` parameter to `contacts`.

Wiring the form controller

Now that we've seen how `EditContactController` works, let's see how it's declared in the Spring application context (Rolodex-portlet.xml):

```
<bean id="editContactController"
    class="com.springinaction.rolodex.controller.
               ➥ EditContactController">
  <property name="formView"
      value="editContact" />
  <property name="rolodexService" ref="rolodexService" />
</bean>
```

`EditContactController` is a form controller; therefore, we must set its `formView` property. The `formView` property specifies a logical view name that will be used to display the form when the controller handles an HTTP GET request or when a form submission error occurs and the user must correct their entries. The `formView` will ultimately be resolved to an actual view implementation (such as a JSP) by a view resolver.

In this case, when the user clicks the Add Contact link, they will navigate to this controller. Initially, the request will be an HTTP GET. Therefore, the `editContact` view will be displayed. Using the `InternalResourceViewResolver` defined earlier in this chapter, the form view will resolve to /WEB-INF/jsp/editContact.jsp.

Normally, the successView property specifies the logical view name for the view that should be rendered upon a successful form submission. But as we've discussed, we're simulating a redirect in the processFormSubmission() method. Consequently, the successView serves no purpose in this controller.

Creating a portlet form in JSP

I thought you might be interested in seeing what the editContact.jsp file looks like; therefore, I've provided an abridged form of it in listing P.6.

Listing P.6 editContact.jsp, which defines a form for creating and editing Rolodex contacts

```
<%@ taglib prefix="portlet"
    uri="http://java.sun.com/portlet" %>
<%@ taglib prefix="spring"
    uri="http://www.springframework.org/tags" %>
<%@ taglib prefix="form"
    uri="http://www.springframework.org/tags/form" %>
<h2>Contact Edit</h2>

<portlet:actionURL var="actionUrl">
  <portlet:param name="action" value="editContact"/>
</portlet:actionURL>

<portlet:renderURL var="contactsUrl">
  <portlet:param name="action" value="contacts"/>
</portlet:renderURL>

<form:form method="POST"
    action="${actionUrl}" commandName="command">

  <form:hidden path="id" />

  <table width="100%" border="0">
    <tr>
      <td align="right">First name:</td>
      <td>
        <form:input path="firstName" size="20" />
      </td>
    </tr>
    <tr>
      <td align="right">Last name:</td>
      <td>
        <form:input path="lastName" size="20" />
      </td>
    </tr>
    <tr>
      <td align="right">Primary phone #:</td>
      <td>
        <form:input path="phone1" size="15" />
```

Uses Spring's form-binding tags

Creates action URL

Creates render URL

```
          </td>
        </tr>
        <tr>
          <td align="right">Alternate phone #:</td>
          <td>
            <form:input path="phone2" size="15" />
          </td>
        </tr>
        <tr><td align="center" colspan="2">
          <input type="submit" value="Save"/>&nbsp;
          <input type="button" value="Cancel"
              onclick="window.location.href=
                  '${contactsUrl}';"/>
        </td></tr>
      </table>
    </form:form>
```

For brevity's sake, we've cut out a few of the form fields from editContact.jsp. Nevertheless, what's left highlights a few points of that we'd like to draw your attention to.

First, you'll notice that there are two different URLs being created. The `actionUrl`, which is the URL that the form will be posted to, is defined with its `action` parameter set to `editContact`. This sends the form submission request to the `EditController` for processing in the action-phase `processFormSubmission()` method.

As for the `contactsUrl` URL, it is used by the form's Cancel button to send the user back to the contact list if they decide to cancel the form.

Another thing that you may find interesting about editContact.jsp is that it makes use of the new form-binding tag library that was introduced in Spring 2. Fortunately for portlet developers, the form-binding tab library works as well for Spring portlet applications as it does with conventional Spring MVC applications.

P.5 Handling portlet exceptions

We have one more loose end to tie up before we wrap up our discussion of Spring Portlet MVC—exception handling.

When an exception is thrown during the course of processing a portlet request, most portal containers display a rather unfriendly message in the portlet box—typically the exception's stack trace. However, we want our portlet application to handle uncaught exceptions in a more graceful way.

Fortunately, Spring's portlet framework provides a version of SimpleMappingExceptionResolver that gracefully handles exceptions that escape from a portlet request. Just like its Spring MVC counterpart (see chapter 13, section 13.4), this class will catch any exceptions that leak out of a portlet request and send the request to a view that is appropriate for the exception.

To use SimpleMappingExceptionResolver, simply declare it as a <bean> in the Spring application context like this:

```
<bean id="defaultExceptionHandler"
    class="org.springframework.web.portlet.handler.
               ➥ SimpleMappingExceptionResolver">
  <property name="exceptionMappings">
    <props>
      <prop key="javax.portlet.PortletSecurityException">
          notAuthorized</prop>
      <prop key="javax.portlet.UnavailableException">
          notAvailable</prop>
    </props>
  </property>
  <property name="defaultErrorView"
      value="defError" />
</bean>
```

The exceptionMappings property maps exceptions to the view that should be rendered if the exception is thrown. The key of each <prop> is the fully qualified class name of the exception to be handled. The value is the logical name of the view to be rendered.

For example, if for some reason a javax.portlet.UnavailableException is thrown while processing a portlet request, the user will be sent to the view whose name is notAvailable. If we're using the InternalResourceViewResolver that was declared earlier in this chapter, the JSP at /WEB-INF/jsp/notAvailable.jsp will be used to render a friendly error message to the user.

The defaultErrorView property defines a catchall exception mapping. If an exception is thrown that can't be found in the exceptionMappings property, the value of defaultErrorView will be used as the name of the error view.

P.6 Summary

Portal websites are a great way of aggregating several sources of information and applications into one convenient web page personalized for the user. Each of the applications presented on a portal page are commonly referred to as portlets.

The Java Portlet Specification standardized development and deployment of portlet applications in much the same way that the Java Servlet Specification standardized development and deployment of web applications. As it turns out, the portlet API bears a striking resemblance to the servlet API, which makes it easy for servlet-savvy developers to learn the ins and outs of portlet development.

Due to the fact that a portlet is fixed to a certain location of a portal page, creating a feature-rich portlet application can be tricky without an MVC framework. Fortunately for Spring developers, Spring 2 introduced Spring Portlet MVC, a portlet-based MVC framework that echoes Spring's web MVC framework in many ways.

In this chapter, we explored Spring Portlet MVC and built a typical portlet application. We started by configuring `DispatcherPortlet`, the front controller for all Spring portlet applications. We then created several controller classes to perform the logic behind the portlet and wired them into Spring along with handler mappings and view resolvers.

By now you've probably caught on that Spring Portlet MVC is a lot like Spring MVC and, in fact, reuses as much of Spring MVC as will fit within the unique characteristics of Java portlets. This is a real boon if you're already familiar with Spring MVC because it means that it's not a huge stretch to learn how to build portlet applications with Spring Portlet MVC.

appendix C: Spring XML configuration reference

In the early days of Spring (pre-2.0), configuring a Spring application context was fairly simple. `<beans>`, `<bean>`, `<property>`, `<value>`, and `<ref>` were sufficient for most circumstances. But Spring 2 added a wealth of new configuration elements. I have found it handy to have a quick reference to all of the XML elements offered in Spring 2 and thought you might find it helpful as well. Therefore, in this appendix I've catalogued all of the Spring XML elements that come with Spring 2.

In addition, a few peripheral frameworks have also adopted Spring 2's configuration support, including Spring Web Flow and the DWR Ajax framework. For your reference, I've documented the configuration elements for those frameworks as well.

C.1 Core bean-wiring elements

At its core, Spring is used to configure JavaBeans and their properties. The elements in this section are (for the most part) the ones that represent the core of Spring and are the ones you'll find yourself using most often.

Schema http://www.springframework.org/schema/beans

Usage Since the `<beans>` element is the root element of the Spring configuration, this schema is the absolute minimum for any Spring configuration XML. The following `<beans>` elements declares this schema as the default schema:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/
              ➥ spring-beans-2.0.xsd">
…
</beans>
```

`<alias>`

Defines an alias for a declared `<bean>`.

May be included in: `<beans>`

Attribute	Description
`alias`	The alias by which the bean (specified in the `name` attribute) will be referred to.
`name`	The original name of the bean to be aliased.

<arg-type>

Defines an argument type for a replaced method. Used with <replaced-method> to further qualify the signature of a replaced method.

May be included in: <replaced-method>

Attribute	Description
match	An argument type to match in the signature of the method to be replaced.

<bean>

Declares a Spring-managed bean.

May be included in: <beans>, <constructor-arg>, <entry>, <key>, <list>, <property>, <set>

May Contain: <constructor-arg>, <description>, <lookup-method>, <meta>, <property>, <replaced-method>

Attribute	Description
abstract	Declares that this bean declaration is abstract (i.e., that it will be sub-beaned with the parent attribute of another bean). *Valid values:* true, false *Default:* false
autowire	Declares how the container should autowire properties of this bean. *Valid values:* byType, byName, default, no, autodetect, constructor *Default:* default (determined by the default-autowire attribute of the <beans> element)
autowire-candidate	If set to false, this bean will be excluded as a candidate for autowiring. This helps avoid mishaps where autowiring picks chooses a bean that isn't suitable for autowiring. Also helps avoid ambiguous autowire candidates.
class	The fully qualified class name of the bean.
dependency-check	How Spring should enforce the setting of properties on this bean. simple indicates that all simple properties (int, String, double) should be injected. object indicates that all properties of complex types should be set. *Valid values:* default, none, all, objects, simple *Default:* default (determined by the default-dependency-check attribute of the <beans> element)
depends-on	Specifies the name of a bean that this bean depends upon. This forces the container to instantiate and configure the dependency bean before this bean is created and configured.
destroy-method	The name of a method to call when this bean is destroyed.

<bean> *(continued)*

Attribute	Description
factory-bean	Used with factory-method to specify a bean that will provide the factory method to create a bean.
factory-method	The name of a method that will be used instead of the constructor to construct an instance of this bean. When used alone, the method must be a static method of the bean specified by class. When used with factory-bean, the method must be a nonstatic method of the bean specified by factory-bean.
id	The bean's identifier (or name).
init-method	The name of a method that will be called immediately after the bean has been created and configured.
lazy-init	Specifies that the container should wait to create the bean until it is referred to. *Valid values:* default, true, false *Default:* default (determined by the default-lazy-init attribute of the <beans> element)
name	The name of the bean.
parent	The name of an abstract bean definition that will serve as the basis for this bean's definition.
scope	Specifies the scope of the bean. This attribute supersedes the singleton attribute, as singleton is a limited form of scoping. Valid values include request, session, globalSession, singleton, and prototype. But can also be a custom scope.
singleton deprecated	Declares this bean to be a singleton (i.e., only one instance is created) as opposed to being a prototype (i.e., one instance is created per reference). This attribute has been replaced by the scope attribute in Spring 2 and is no longer available when using the Spring 2 XSD. It is, however, still available in the DTD. *Valid values*: true, false *Default*: true

<beans>

The root element of the Spring XML configuration.

May contain: <alias>, <bean>, <description>, <import>, <aop:config>, <aop:spring-configured>, <jee:jndi-lookup>,<jee:local-slsb>, <jee:remote-slsb>, <lang:bsh>, <lang:groovy>, <lang:jruby>, <tx:advice>, <tx:annotation-driven>, <util:constant>, <util:list>, <util:map>, <util:constant>, <util:properties>, <util:property-path>, <util:set>

Attribute	**Description**
default-autowire	The default autowiring strategy to be used for all beans in this configuration. *Valid values:* byType, byName, no, autodetect, constructor *Default:* no
default-dependency-check	The default dependency-checking strategy to be used for all beans in this configuration. *Valid values:* none, all, objects, simple *Default:* none
default-destroy-method	The default method to be called on each bean when that bean is destroyed.
default-init-method	The default method to be called on each bean when that bean is created and configured.
default-lazy-init	The default lazy-initialization strategy to be applied to all beans in this configuration. *Valid values:* true, false *Default:* false
default-merge	The default collection merge behavior. If true then collection properties on parent beans will be merged into collection properties on child beans. *Valid values*: true, false *Default:* false

<constructor-arg>

Specifies a constructor argument for construction of a bean. Can be used for constructor injection or can be used with the `<bean>` element's `factory-method` attribute to specify arguments for the factory method.

May be included in: `<bean>`

May contain: `<bean>`, `<description>`, `<idref>`, `<list>`, `<map>`, `<null>`, `<props>`, `<ref>`, `<set>`, `<value>`, `<util:constant>`, `<util:properties>`, `<util:property-path>`

Attribute	Description
`index`	Can be used to specify which constructor argument this particular `<constructor-arg>` applies to when multiple constructor arguments are specified.
`ref`	The ID of another declared bean that is to be wired into this constructor argument.
`type`	Can be used to specify the type of the constructor argument to help Spring determine which constructor argument that this particular `<constructor-arg>` applies to when multiple constructor arguments are specified.
`value`	Specifies the simple value to be wired into a constructor argument.

<description>

Provides a description for the context, bean, property, or constructor argument. Used to provide documentation for tools, such as Spring BeanDoc.

May be included in: `<bean>`, `<beans>`, `<constructor-arg>`, `<property>`

<entry>

Specifies an entry in a map.

May be included in: `<map>`

May contain: `<bean>`, `<idref>`, `<key>`, `<list>`, `<map>`, `<null>`, `<props>`, `<ref>`, `<set>`, `<value>`, `<util:constant>`, `<util:properties>`, `<util:property-path>`

Attribute	Description
`key`	Specify the key of the map entry as a simple value (i.e., `String`, `int`, etc.)
`key-ref`	Wire in another bean in the Spring context as the key of the map entry.
`value`	Specify the value of the map entry as a simple value (i.e., `String`, `int`, etc.)
`value-ref`	Wire in another bean in the Spring context as the value of the map entry.

`<idref>`

Defines a value that can be validated to be the name of a bean.

May be included in: `<constructor-arg>`, `<entry>`, `<key>`, `<list>`, `<property>`, `<set>`

Attribute	Description
`bean`	Specifies the name of a bean (in any context).
`local`	Specifies the name of a bean in the local context.

`<import>`

Imports another Spring XML configuration file into this Spring configuration.

May be included in: `<beans>`

Attribute	Description
`resource`	The name of a resource file that contains another Spring application context definition.

`<key>`

Defines the key of a map entry.

May be included in: `<entry>`

May contain: `<bean>`, `<idref>`, `<list>`, `<map>`, `<null>`, `<props>`, `<ref>`, `<set>`, `<value>`, `<util:constant>`, `<util:properties>`, `<util:property-path>`

`<list>`

Defines a collection of values as a list.

May be included in: `<constructor-arg>`, `<entry>`, `<key>`, `<list>`, `<property>`, `<set>`

May contain: `<bean>`, `<idref>`, `<list>`, `<map>`, `<null>`, `<props>`, `<ref>`, `<set>`, `<value>`, `<util:constant>`, `<util:properties>`, `<util:property-path>`

Attribute	Description
`merge`	If `true`, this list will be merged with a list specified by a parent bean (if any). *Valid values:* `true`, `false` *Default:* Determined by the value of `default-merge` on `<beans>` element (`false` if `default-merge` isn't specified)
`value-type`	Specifies the value type of the collection. Optional, but can be used to help property editors determine the proper type to place in a collection when specified as Strings in the context configuration.

<lookup-method>

Specifies a lookup method style of injection where an existing abstract or concrete method in a bean is replaced with a method that returns the wired value.

May be included in: <bean>

Attribute	Description
bean	The ID of the bean to be wired into the lookup method.
name	The name of the method that will be replaced with the lookup method.

<map>

Defines a map of key/value pairs.

May be included in: <constructor-arg>, <entry>, <key>, <list>, <property>, <set>
May contain: <entry>

Attribute	Description
key-type	Specifies the default type of the map key. Optional, but useful in guiding property editors in converting String values.
merge	If true, this map will be merged with a map specified by a parent bean (if any). *Valid values:* true, false *Default:* Determined by the value of default-merge on <beans> element (false if default-merge isn't specified)
value-type	Specifies the default type of the map value. Optional, but useful in guiding property editors in converting String values.

<meta>

Adds metadata values to a bean or a bean property.

May be included in: <bean>, <property>

Attribute	Description
key	The metadata key.
value	The value of the metadata.

<null>

Defines a null value to be injected into a property.

May be included in: <constructor-arg>, <entry>, <key>, <list>, <property>, <set>

<prop>

Defines a member of a set of properties defined by `<props>`. The content for the `<prop>` element is its value.

May be included in: `<props>`

Attribute	Description
`key`	The property key, defined as a `String`.

<property>

Defines a bean property to be set by setter injection.

May be included in: `<bean>`

May contain: `<bean>`, `<description>`, `<idref>`, `<list>`, `<map>`, `<meta>`, `<null>`, `<props>`, `<ref>`, `<set>`, `<value>`, `<util:constant>`, `<util:properties>`, `<util:property-path>`

Attribute	Description
`name`	The name of the bean property.
`ref`	Refers to the ID of a bean that is to be injected into this property.
`value`	Injects a simple value (`String`, `int`, etc.) into the property.

<props>

Defines a value of type `java.util.Properties`.

May be included in: `<constructor-arg>`, `<entry>`, `<key>`, `<list>`, `<property>`, `<set>`

May contain: `<prop>`

Attribute	Description
`merge`	If `true`, this property set will be merged with a property set specified by a parent bean (if any). *Valid values:* `true`, `false` *Default:* Determined by the value of `default-merge` on `<beans>` element (`false` if `default-merge` isn't specified)

<ref>

Defines a value that is a reference to a bean in the Spring context.

May be included in: `<constructor-arg>`, `<entry>`, `<key>`, `<list>`, `<property>`, `<set>`

Attribute	Description
`bean`	The name of the bean to be referenced (can be in any context).
`local`	The name of a bean to be referenced in the local context.
`parent`	The name of a bean to be referenced in the parent context.

`<replaced-method>`

Replaces an existing method (abstract or concrete) in a bean with a new implementation defined by an implementation of `MethodReplacer`.

May be included in: `<bean>`

May contain: `<arg-type>`

Attribute	Description
`name`	The name of the method that is to be replaced. (The signature can be further qualified by using `<arg-type>`.)
`replacer`	The ID of a bean that implements `MethodReplacer`.

`<set>`

Defines a collection of values as a set.

May be included in: `<constructor-arg>`, `<entry>`, `<key>`, `<list>`, `<property>`, `<set>`

May contain: `<bean>`, `<idref>`, `<list>`, `<map>`, `<null>`, `<props>`, `<ref>`, `<set>`, `<value>`, `<util:property-path>`

Attribute	Description
`merge`	If `true`, this set will be merged with a set specified by a parent bean (if any). *Valid values:* `true`, `false` *Default:* Determined by the value of `default-merge` on `<beans>` element (`false` if `default-merge` isn't specified)
`value-type`	Specifies the default type of the values in the collection. Optional, but helpful in guiding property editors in converting Strings.

`<value>`

Defines a value as a simple type (`String`, `int`, etc.). Note that even though the value may be specified as a `String`, it could be used to inject into a more complex type if an appropriate property editor is in place. For example, a property of type `java.io.File` can be injected using a `String` value that is the path of the file.

May be included in: `<constructor-arg>`, `<entry>`, `<key>`, `<list>`, `<property>`, `<set>`

Attribute	Description
`type`	Forces the type of the value. Useful when a property's setter method is overloaded to accept multiple types.

C.2 AOP elements

Aspect-oriented programming has always been a cornerstone feature of Spring. And in Spring 2, it only gets better with the addition of the elements described in this section.

Schema: http://www.springframework.org/schema/aop

Usage: The following <beans> declaration declares the AOP schema in the aop namespace (in addition to the default beans schema):

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/
            ➥ spring-beans-2.0.xsd
       http://www.springframework.org/schema/aop
       http://www.springframework.org/schema/aop/
           ➥ spring-aop-2.0.xsd">
…
</beans>
```

<aop:advisor>

Defines an AOP advisor.

May be included in: <aop:config>

Attribute	Description
advice-ref	The ID of advice to be associated with this advisor.
id	The ID of this advisor.
order	Specifies an order in which this advisor will be applied with respect to other advisors' order attributes.
pointcut	A pointcut expression to be used with this advisor (in AspectJ syntax).
pointcut-ref	The ID of an <aop:pointcut> definition to be used with this advisor.

`<aop:after>`

Defines an AOP after advice.

be included in: `<aop:aspect>`

Attribute	Description
`arg-names`	Comma-separated list of arguments to be passed from the pointcut expression to the advice method.
`method`	The method that implements the advice.
`pointcut`	A pointcut expression to be used with this advice (in AspectJ syntax).
`pointcut-ref`	The ID of an `<aop:pointcut>` definition to be used with this advice.

`<aop:after-returning>`

Defines an AOP after returning advice.

May be included in: `<aop:aspect>`

Attribute	Description
`arg-names`	Comma-separated list of arguments to be passed from the pointcut expression to the advice method.
`method`	The method that implements the advice.
`pointcut`	A pointcut expression to be used with this advice (in AspectJ syntax).
`pointcut-ref`	The ID of an `<aop:pointcut>` definition to be used with this advice.
`returning`	The name of the parameter in the advice method that should be used as the return value.

`<aop:after-throwing>`

Defines an AOP after advice.

May be included in: `<aop:aspect>`

Attribute	Description
`arg-names`	Comma-separated list of arguments to be passed from the pointcut expression to the advice method.
`method`	The method that implements the advice.
`pointcut`	A pointcut expression to be used with this advice (in AspectJ syntax).

<aop:after-throwing> *(continued)*

Attribute	Description
pointcut-ref	The ID of an <aop:pointcut> definition to be used with this advice.
throwing	The name of a parameter in the advice method that should be used as the throwing exception.

<aop:around>

Defines an AOP around advice.

May be included in: <aop:aspect>

Attribute	Description
arg-names	Comma-separated list of arguments to be passed from the pointcut expression to the advice method.
method	The method that implements the advice.
pointcut	A pointcut expression to be used with this advice (in AspectJ syntax).
pointcut-ref	The ID of an <aop:pointcut> definition to be used with this advice.

<aop:aspect>

Defines a singleton aspect.

May be included in: <aop:config>

May contain: <aop:after>, <aop:after-returning>, <aop:after-throwing>, <aop:around>, <aop:before>, <aop:declare-parents>, <aop:pointcut>

Attribute	Description
id	The ID of this aspect.
order	Specifies the order that this aspect should be applied in, relative to other aspects.
ref	The ID of a <bean> that implements the advice for this aspect.

<aop:aspectj-autoproxy>

Declares an autoproxy for @AspectJ-annotated aspects. Has the side effect of also autoproxying Spring AOP advisors.

May contain: <aop:include>

Attribute	Description
proxy-target-class	If true, forces autoproxy creator to use class proxying. *Valid values:* true, false *Default:* false

<aop:before>

Defines an AOP before advice.

May be included in: <aop:aspect>

Attribute	Description
arg-names	Comma-separated list of arguments to be passed from the pointcut expression to the advice method.
method	The method that implements the advice.
pointcut	A pointcut expression to be used with this advice (in AspectJ syntax).
pointcut-ref	The ID of an <aop:pointcut> definition to be used with this advice.

<aop:config>

The top-level AOP configuration element. Most AOP configuration elements must be declared within the scope of an <aop:config> element.

May be included in: <beans>
May contain: <aop:advisor>, <aop:aspect>, <aop:pointcut>

Attribute	Description
proxy-target-class	If true, forces autoproxy creator to use class proxying. *Valid values:* true, false *Default:* false

<aop:declare-parents>

Defines an AOP introduction.

May be included in: <aop:aspect>

Attribute	Description
default-impl	The fully qualified class name of a class that provides the default implementations of the methods required by the introduction interface.
implement-interface	The fully qualified class name of an interface to be introduced.
types-matching	A pattern specifying types to which the introduction interface should be introduced.

<aop:include>

Used with `<aop:aspectj-autoproxy>` to limit which @AspectJ beans are autoproxied to those whose name matches a pattern.

May be included in: `<aop:aspectj-autoproxy>`

Attribute	Description
`name`	The name pattern to match when autoproxying @AspectJ beans.

<aop:pointcut>

Defines an AOP pointcut.

May be included in: `<aop:aspect>`, `<aop:config>`

Attribute	Description
`expression`	The expression that defines the pointcut (e.g., `"execution(* *.get*(..))"` if using AspectJ expressions).
`id`	The ID of the pointcut. Useful if defining a pointcut that will be used by more than one aspect or advice.
`type`	The type of expression to be used, either AspectJ-style or regular expression. Valid values: `aspectj`, `regex` Default: `aspectj`

<aop:scoped-proxy>

Specifies that a bean should be proxied with a scoped proxy. Beans marked with `<aop:scoped-proxy>` will be exposed via a proxy, with the actual bean being retrieved from some other scope (such as HttpSession) as/when needed.

May be included in: `<bean>`

Attribute	Description
`proxy-target-class`	If `true`, a CGLIB-based proxy will be created for the scoped bean. This means that CGLIB will be required in the classpath. If false, a JDK interface-based proxy will be created. This requires no additional items in the classpath, but does require that all objects that the bean is injected into access it through an interface that the bean implements. *Default:* `true`

<aop:spring-configured>

Defines a bean to be configured (i.e., injected) by Spring, even if the bean is created outside of Spring. Used with the `@Configured` annotation.

May be included in: `<beans>`

C.3 Java Enterprise Edition elements

Even though Spring eliminates many of the needs for EJBs, there's no reason that you can't use EJBs alongside POJOs within an application. In fact, the elements in this section make it possible to declare references to EJBs and then wire them into your Spring-managed POJOs.

Schema: http://www.springframework.org/schema/jee

Usage: The following `<beans>` declaration declares the Java Enterprise Edition schema in the `jee` namespace (in addition to the default `beans` schema):

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jee="http://www.springframework.org/schema/jee"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/
            ➥ spring-beans-2.0.xsd
       http://www.springframework.org/schema/jee
       http://www.springframework.org/schema/jee/
           ➥ spring-jee-2.0.xsd">
…
</beans>
```

`<jee:environment>`

Specifies the JNDI environment. In normal (e.g., non-Spring) JNDI, this is usually specified as a `java.util.Hashtable`. Here, it is specified as name-value pairs in the content of the `<jee:environment>` element. For example:

```
<jee:environment>
  foo=bar
  dog=canine
</jee:environment>
```

May be included in: `<jee:lookup>`, `<jee:local-slsb>`, `<jee:remote-slsb>`

`<jee:jndi-lookup>`

Creates a bean by looking up a value from JNDI.

May be included in: `<beans>`

May contain: `<jee:environment>`

Attribute	Description
`cache`	Specifies whether or not the value should be cached. *Valid values:* `true`, `false` *Default:* `true`
`expected-type`	The type of object that is to be retrieved from JNDI.
`id`	The ID of the bean.
`jndi-name`	The name of the object in JNDI

`<jee:jndi-lookup>` *(continued)*

Attribute	Description
`lookup-on-startup`	Specifies whether Spring should retrieve the object on container startup or wait until it is requested. *Valid values:* `true`, `false` *Default:* `true`
`proxy-interface`	The interface that is to be applied to the object retrieved from JNDI.
`resource-ref`	Specifies whether or not this is a resource reference. If `true` then `java:comp/env/` will be prepended to the `jndi-name`. *Valid values:* `true`, `false` *Default:* `false`

`<jee:local-slsb>`

Defines a reference to a local stateless session EJB that can be wired as a bean in a Spring context.

May contain: `<jee:environment>`

Attribute	Description
`business-interface`	The fully qualified name of the interface that declares the business methods of the EJB.
`cache-home`	Specifies whether or not the home interface should be cached. *Valid values:* `true`, `false` *Default:* `true`
`id`	The ID of the bean.
`jndi-name`	The name of the EJB in JNDI
`lookup-home-on-startup`	Whether or not the EJB's home interface is retrieved on startup. Can be set to `false` to delay lookup of the home interface until the EJB is needed (to allow the EJB server to start later). *Valid values:* `true`, `false` *Default:* `true`
`resource-ref`	Specifies whether or not this is a resource reference. If `true`, then `java:comp/env/` will be prepended to the `jndi-name`. *Valid values:* `true`, `false` *Default:* `false`

`<jee:remote-slsb>`

Defines a reference to a remote stateless session EJB that can be wired as a bean in a Spring context.

May contain: `<jee:environment>`

Attribute	Description
`business-interface`	The fully qualified name of the interface that declares the business methods of the EJB.
`cache-home`	Specifies whether or not the home interface should be cached. *Valid values:* `true, false` *Default:* `true`
`home-interface`	The fully qualified name of the EJB's home interface.
`id`	The ID of the bean.
`jndi-name`	The name of the object in JNDI.
`lookup-home-on-startup`	Specifies whether or not the EJB's home interface is retrieved on startup. Can be set to `false` to delay lookup of the home interface until the EJB is needed (to allow the EJB server to start later). *Valid values:* `true`, `false` *Default:* `true`
`refresh-home-on-connect-failure`	Specifies whether or not the home interface should be refreshed when a connection fails. *Valid values:* `true`, `false` *Default:* `false`
`resource-ref`	Specifies whether or not this is a resource reference. If `true` then `java:comp/env/` will be prepended to the `jndi-name`. *Valid values:* `true`, `false` *Default:* `false`

C.4 Script language elements

Dynamic languages are all the rage. With the elements in this section, you can reap the benefits of dynamic scripting languages such as Ruby, Groovy, and BeanShell within your Spring applications.

Schema: http://www.springframework.org/schema/lang

Usage: The following `<beans>` declaration declares the scripting schema in the `lang` namespace (in addition to the default `beans` schema):

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:lang="http://www.springframework.org/schema/lang"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/
            ➥ spring-beans-2.0.xsd
        http://www.springframework.org/schema/lang
        http://www.springframework.org/schema/lang/
            ➥ spring-lang-2.0.xsd">
…
</beans>
```

`<lang:bsh>`

Defines a bean that is scripted in BeanShell (BSH).

May be included in: `<beans>`
May contain: `<lang:inline-script>`, `<lang:property>`

Attribute	Description
`id`	The ID of the scripted bean.
`refresh-check-delay`	Specifies how often the script is refreshed (in milliseconds). *Default:* `No refresh`
`scope`	Specifies the scope of the scripted bean. Set to `singleton` by default, which will use one shared instance for all attempts to retrieve this bean. `prototype` specifies an independent instance each time the bean is retrieved
`script-interfaces`	Comma-delimited list of interfaces that the script will implement.
`script-source`	The path to the script source

`<lang:groovy>`

Defines a bean that is scripted in Groovy.

May be included in: `<beans>`
May contain: `<lang:inline-script>`, `<lang:property>`

Attribute	Description
`id`	The ID of the scripted bean
`customizer-ref`	Reference to a bean that implements `GroovyObjectCustomizer`. Allows for postinstantiation customization of the Groovy bean.
`refresh-check-delay`	Specifies how often the script is refreshed (in milliseconds). *Default:* `No refresh`

<lang:groovy> *(continued)*

Attribute	Description
scope	Specifies the scope of the scripted bean. Set to singleton by default, which will use one shared instance for all attempts to retrieve this bean. prototype specifies an independent instance each time the bean is retrieved.
script-source	The path to the script source.

<lang:inline-script>

Used to script a bean directly in the Spring configuration file instead of referring to a script file. The script is included as the content of the <lang:inline-script> tag. To accommodate characters that have special meaning in XML (less-than and greater-than signs, for instance), you should wrap the script in <![CDATA[…]]>.

May be included in: <lang:bsh>, <lang:groovy>, <lang:jruby>

<lang:jruby>

Defines a bean that is scripted in Ruby (JRuby).

May be included in: <beans>

May contain: <lang:inline-script>, <lang:property>

Attribute	Description
id	The ID of the scripted bean.
refresh-check-delay	Specifies how often the script is refreshed (in milliseconds). *Default:* No refresh
scope	Specifies the scope of the scripted bean. Set to singleton by default, which will use one shared instance for all attempts to retrieve this bean. prototype specifies an independent instance each time the bean is retrieved.
script-interfaces	Comma-delimited list of interfaces that the Ruby object will implement.
script-source	The path to the script source.

<lang:property>

Injects a property value into the scripted bean. Functionally equivalent to the <property> element in section C.1.

May be included in: <lang:bsh>, <lang:groovy>, <lang:jruby>

May contain: <bean>, <description>, <idref>, <list>, <map>, <meta>, <null>, <props>, <ref>, <set>, <value>

Attribute	Description
name	The name of the bean property.
ref	Refers to the ID of a bean that is to be injected into this property.
value	Injects a simple value (String, int, etc.) into the property.

C.5 Transaction declaration elements

Declarative transaction support for POJOs is arguably the killer feature of Spring. With the elements in this section (and a little help from the AOP elements), declarative transactions are now much easier.

Schema: http://www.springframework.org/schema/tx

Usage: The following `<beans>` declaration declares the transaction schema in the `tx` namespace (in addition to the default `beans` schema):

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/
            ➥ spring-beans-2.0.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/
            ➥ spring-tx-2.0.xsd">
…
</beans>
```

`<tx:advice>`

Defines transaction advice.

May be included in: `<beans>`
May contain: `<tx:attributes>`

Attribute	Description
`id`	The ID of the advice.
`transaction-manager`	The ID of the bean that declares the transaction manager to use when applying transactions.

`<tx:annotation-driven>`

Declares that Spring should apply transactions to beans that are annotated with `@Transactional` or that have methods that are annotated with `@Transactional`.

May be included in: `<beans>`

Attribute	Description
`order`	Specifies an order of the execution of the transaction advisor, relative to other advice executing at a specific joinpoint.

<tx:annotation-driven> *(continued)*

Attribute	Description
proxy-target-class	If true, a CGLIB-based proxy will be created for the transactional bean. This means that CGLIB will be required in the classpath. If false, a JDK interface-based proxy will be created. This requires no additional items in the classpath, but does require that all objects that the bean is injected into access it through an interface that the bean implements.
transaction-manager	The ID of the bean that declares the transaction manager to use when applying transactions.

<tx:attributes>

Defines transaction attributes to be applied by default to all methods matched by the advisor's pointcut. For more fine-grained transaction control over individual methods, consider using <tx:method>.

May be included in: <tx:advice>
May contain: <tx:method>

Attribute	Description
isolation	Specifies the isolation level for the transaction. *Valid values:* DEFAULT, READ_UNCOMMITTED, READ_COMMITTED, REPEATABLE_READ, SERIALIZABLE *Default:* DEFAULT
name	Defines the name of the transaction. Can be null. Useful for display in a transaction monitor, if applicable.
no-rollback-for	Comma-separated list of exceptions for which the transaction should not roll back.
propagation	Specifies the propagation behavior of the transaction. *Valid values:* REQUIRED, SUPPORTS, MANDATORY, REQUIRES_NEW, NOT_SUPPORTED, NEVER, NESTED *Default:* REQUIRED
read-only	Specifies whether the transaction is read-only. *Valid values:* true, false *Default:* false
rollback-for	Comma-separated list of exceptions for which the transaction should be rolled back. Note that unless specified by no-rollback-for, all runtime exceptions trigger a rollback
timeout	Specifies the transaction timeout in seconds. *Default:* -1 (no timeout)

<tx:method>

Defines transaction attributes based on a pattern defined in the `name` attribute.

May be included in: `<tx:attributes>`

Attribute	Description
`isolation`	Specifies the isolation level for the transaction. *Valid values:* `DEFAULT, READ_UNCOMMITTED, READ_COMMITTED, REPEATABLE_READ, SERIALIZABLE` *Default:* `DEFAULT`
`name`	Defines a pattern for matching method names. May include wildcards (e.g., `"get*"`).
`no-rollback-for`	Comma-separated list of exceptions for which the transaction should not roll back.
`propagation`	Specifies the propagation behavior of the transaction. *Valid values:* `REQUIRED`, `SUPPORTS`, `MANDATORY`, `REQUIRES_NEW`, `NOT_SUPPORTED`, `NEVER`, `NESTED` *Default:* `REQUIRED`
`read-only`	Specifies whether the transaction is read-only. *Valid values:* `true`, `false` *Default:* `false`
`rollback-for`	Comma-separated list of exceptions for which the transaction should be rolled back. Note that unless specified by `no-rollback-for`, all runtime exceptions trigger a rollback.
`timeout`	Specifies the transaction timeout in seconds. *Default:* `-1` (no timeout)

C.6 Utility elements

Sometimes it's nice to refer to constants, property files, or collections as beans. The elements in the `util` namespace help you define beans from things that aren't normally thought of as beans.

Schema: http://www.springframework.org/schema/util

Usage: The following `<beans>` declaration declares the utility schema in the `util` namespace (in addition to the default `beans` schema):

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:util="http://www.springframework.org/schema/util"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
```

```
          http://www.springframework.org/schema/beans/
              ➥ spring-beans-2.0.xsd
          http://www.springframework.org/schema/util
          http://www.springframework.org/schema/util/
              ➥ spring-util-2.0.xsd">
…
</beans>
```

`<util:constant>`

Define a bean whose value is drawn from a public static field on a type.

May be included in: `<beans>`

Attribute	Description
`id`	The ID of the constant bean.
`static-field`	The fully qualified type and static field from which to draw the constant value.

`<util:list>`

Defines a list collection (`java.util.List`) as a bean.

May be included in: `<beans>`

May contain: `<bean>`, `<idref>`, `<list>`, `<map>`, `<null>`, `<props>`, `<ref>`, `<set>`, `<value>`

Attribute	Description
`id`	The ID of the list bean.
`list-class`	The fully qualified class name of a `java.util.List` implementation to use for this list bean. *Default:* `java.util.ArrayList`
`merge`	If `true`, this list will be merged with a list specified by a parent bean (if any). *Valid values:* `true`, `false` *Default:* Determined by the value of `default-merge` on `<beans>` element (`false` if `default-merge` isn't specified)
`value-type`	Specifies a default type for the values in the collection. Optional, but helpful for property editors in converting `Strings`.

<util:map>

Defines a map collection (java.util.Map) as a bean.
The type of map depends on what is available. If running on JDK 1.4 or higher, java.util.LinkedHashMap is the default. If running on JDK 1.3 and Commons Collections is available in the classpath then LinkedMap is used. As a last resort, java.util.HashMap is used.
In any event, the map type can be explicitly set with the map-class attribute.

May be included in: <beans>

May contain: <entry>

Attribute	Description
id	The ID of the map bean.
key-type	Specifies the default type of the map key. Optional, but useful in guiding property editors in converting String values.
map-class	The fully qualified class name of a java.util.Map implementation to use for this map bean. *Default:* java.util.LinkedHashMap (JDK 1.4 or higher)
merge	If true, this map will be merged with a list specified by a parent bean (if any). *Valid values:* true, false *Default:* Determined by the value of default-merge on <beans> element (false if default-merge isn't specified).
value-type	Specifies the default type of the map value. Optional, but useful in guiding property editors in converting String values.

<util:properties>

Loads a properties file into Spring so that it can be injected into a bean property of type java.util.Properties.

May be included in: <beans>

May contain: <prop>

Attribute	Description
id	The name that the properties will be referred to within the Spring context.
location	The location of the properties file.
merge	If true, the properties will be merged with the properties specified by a parent bean (if any). *Valid values:* true, false *Default:* Determined by the value of default-merge on <beans> element (false if default-merge isn't specified)

`<util:property-path>`

Reference a property on a bean and expose its value as a bean.

May be included in: `<beans>`

Attribute	Description
`id`	The ID of the new bean.
`path`	The path to the property to be exposed as a bean.

`<util:set>`

Defines a set collection (`java.util.Set`) as a bean.
The type of set depends on what is available. If running on JDK 1.4 or higher, `java.util.LinkedHashSet` will be used. If running on JDK 1.3 and Commons Collections is in the classpath, `ListOrderedSet` is used. As a last resort, `java.util.HashSet` is chosen. In any event, the set type can be explicitly chosen using the `set-class` attribute.

May be included in: `<beans>`

May contain: `<bean>`, `<idref>`, `<list>`, `<map>`, `<null>`, `<props>`, `<ref>`, `<set>`, `<value>`

Attribute	Description
`id`	The ID of the set bean.
`merge`	If `true`, this set will be merged with a set specified by a parent bean (if any). *Valid values:* `true`, `false` *Default:* Determined by the value of `default-merge` on `<beans>` element (`false` if `default-merge` isn't specified)

`<util:set>` *(continued)*

Attribute	Description
`set-class`	The fully qualified class name of a `java.util.Set` implementation to use for this set bean. *Default:* `java.util.LinkedHashSet` (if JDK 1.4 or higher)
`value-type`	Specifies the default type of the collection values. Optional, but useful in guiding property editors in converting `String` values.

C.7 Spring Web Flow configuration elements

When you need to build a web application that leads the user through a specific flow, you can't go wrong with Spring Web Flow. Spring Web Flow helps you define an application's flow external to the application's logic.

The final release of Spring Web Flow includes a handful of elements that simplify configuration of Spring Web Flow within a Spring application context. Those configuration elements are documented in this section.

It's important to understand that the elements in this section define bean that process flows, not for defining flows. The flow definition elements are documented separately in appendix E.

Schema: http://www.springframework.org/schema/webflow-config

Usage: The following `<beans>` declaration declares the utility schema in the `util` namespace (in addition to the default `beans` schema):

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:flow="http://www.springframework.org/schema/
        ➥ webflow-config"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/
            ➥ spring-beans-2.0.xsd
        http://www.springframework.org/schema/webflow-config
        http://www.springframework.org/schema/webflow-config/
                        ➥ spring-webflow-config-1.0.xsd">
…
</beans>
```

`<flow:alwaysRedirectOnPause>`

Specifies whether or not a browser redirect is performed each time a flow execution pauses.

May be included in: `<flow:execution-attributes>`

Attribute	Description
`value`	If `true`, always perform a redirect when a flow execution is paused. *Valid values:* `true`, `false` *Default:* `true`

`<flow:attribute>`

Sets a flow execution attribute.

May be included in: `<flow:execution-attributes>`

Attribute	Description
`name`	The name of the attribute.
`type`	The attribute's type.
`value`	The value to assign to the attribute.

`<flow:execution-attributes>`

Configures flow execution attributes. This is a container element for one or more `<flow:attribute>` elements.

May be included in: `<flow:executor>`

May contain: `<flow:alwaysRedirectOnPause>`, `<flow:attribute>`

`<flow:execution-listeners>`

Configures flow execution listeners. This is a container element for one or more `<flow:listener>` elements.

May be included in: `<flow:executor>`

May contain: `<flow:listener>`

`<flow:executor>`

Deploys a flow executor.

May be included in: `<beans>`

May contain: `<flow:execution-attributes>`, `<flow:execution-listeners>`, `<flow:repository>`

Attribute	Description
`id`	The bean ID of the flow executor.
`registry-ref`	References the flow registry containing the flows to be executed by this executor.
`repository-type`	The type of repository. *Valid values:* `simple`, `continuation`, `client`, `singlekey` *Default:* `continuation`

<flow:listener>

Declares a flow execution listener that will observe the execution of one or more flows.

May be included in: <flow:execution-listeners>

Attribute	Description
criteria	Used to restrict the flow definitions that this listener listens to. *Default:* * (i.e., all flows)
ref	References the <bean> that implements the listener.

<flow:location>

Specifies a path to a flow definition resource.

May be included in: <flow:registry>

Attribute	Description
path	The path to the flow definition. May include Ant-style wildcard paths to load multiple flow definitions.

<flow:registry>

Declares a flow definition registry made up of flows specified by one or more <flow:location> elements. Each flow will be identified by the flow's resource filename without the extension. For example, a flow contained in pizza-flow.xml will be identified as pizza-flow.

May be included in: <beans>
May contain: <flow:location>

Attribute	Description
id	The bean ID of the flow registry.

<flow:repository>

Defines a flow repository.

May be included in: <flow:executor>

Attribute	Description
conversation-manager-ref	References a conversation manager bean that this repository should use.
max-continuations	The maximum number of flow execution continuations allowed by this repository per conversation. Only relevant when the repository is a "continuation" repository.
max-conversations	The maximum number of concurrent conversations allowed by this repository. Ignored when conversation-manager-ref is set.
type	The type of flow execution repository to use. *Valid values:* continuation, simple, client, singlekey

C.8 DWR configuration elements

When it comes to doing Ajax with Spring, DWR is the way to go. DWR provides a very nice set of configuration elements that help you configure DWR-accessible beans directly in the Spring application context.

To use these elements, you'll need to have the very latest version of DWR in your application's classpath. At the time of this writing, the latest version of DWR is 2.0-M3 and can be downloaded from

https://dwr.dev.java.net/servlets/ProjectDocumentList.

Schema: http://www.directwebremoting.org/schema/spring

Usage: The following `<beans>` declaration declares the dwr schema in the `dwr` namespace (in addition to the default `beans` schema):

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:dwr="http://www.directwebremoting.org/schema/spring-dwr"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/
            ➥ spring-beans-2.0.xsd
        http://www.directwebremoting.org/schema/spring-dwr
        http://www.directwebremoting.org/schema/
            ➥ spring-dwr-2.0.xsd">
…
</beans>
```

<dwr:auth>

Applies security constraints to DWR-remoted objects.

May be included in: `<dwr:remote>`

Attribute	Description
`method`	The name of the method to secure.
`role`	The security role required to invoke the method.

<dwr:config-param>

Specifies a configuration parameter for the controller.

May be included in: `<dwr:controller>`

Attribute	Description
`name`	The name of the parameter
`value`	The value of the parameter

`<dwr:configuration>`

Parent element for basic DWR configuration within a Spring context. If you wish to declare a DWR converter or creator in Spring, they'll need to be declared within a `<dwr:configuration>`.

May be included in: `<beans>`

May contain: `<dwr:convert>`, `<dwr:create>`, `<dwr:signatures>`

`<dwr:controller>`

Declares a Spring MVC controller that handles DWR requests. This makes it possible to configure DWR completely in Spring, without a `DwrServlet` in web.xml.

May be included in: `<dwr:beans>`
May contain: `<dwr:config-param>`

Attribute	Description
`debug`	Specifies whether or not debug mode is on. *Valid values:* `true`, `false` *Default value:* `false`
`id`	The ID of the controller bean.
`name`	The name of the controller.

`<dwr:convert>`

Declares how a complex type should be converted from Java to JavaScript.

May be included in: `<dwr:configuration>`
May contain: `<dwr:exclude>`, `<dwr:include>`

Attribute	Description
`class`	The fully qualified class name of the Java object to be converted.
`javascript`	The name given to the converted type in JavaScript.
`type`	The converter type. *Valid values:* `array`, `bean`, `collection`, `enum`, `map`

`<dwr:create>`

Declares a DWR creator.

May be included in: `<dwr:configuration>`

Attribute	Description
`class`	The fully qualified class name of the Java object to expose in JavaScript.
`javascript`	The name given to the object in JavaScript.
`type`	The creator type. *Valid values:* `new`, `null`, `scripted`, `spring`, `jsf`, `struts`, `pageflow`

<dwr:data>

Specifies data for DWR signatures.

May be included in: <dwr:signatures>

<dwr:exclude>

Declares that a bean's method should not be included in the client-side interface of the remoted bean. By default, all public methods of the server-side object are available in JavaScript. <dwr:exclude> can be used to exclude methods that you do not want exposed to the client.

May be included in: <dwr:convert>, <dwr:remote>

Attribute	Description
method	The name of the method to exclude from the client-side interface.

<dwr:filter>

Declares a filter to be applied per request for methods exposed.

May be included in: <dwr:remote>

Attribute	Description
class	The class name of a filter to be applied.

<dwr:include>

Explicitly declares a method that will be included in the client-side interface of the remoted bean. By default, all public methods of the server-side object are available in JavaScript. If <dwr:include> is used, only those methods that are explicitly included will be exposed to the client-side interface.

May be included in: <dwr:convert>, <dwr:remote>

Attribute	Description
method	The name of the method to be included in the client-side interface.

<dwr:latencyfilter>

Configures an Ajax filter that simulates network latency by delaying invocation of the remote method. Half of the time specified in delay is spent before the invocation and half is spent after the invocation.

May be included in: <dwr:remote>

Attribute	Description
delay	The total amount of time, in milliseconds, to delay the invocation of the remote method. Half of this value is spent before the invocation; the other half is spent after.

`<dwr:remote>`

Exposes a bean as a DWR-remoted bean that can be accessed in client-side JavaScript.

May be included in: `<bean>`

May contain: `<dwr:auth>`, `<dwr:convert>`, `<dwr:exclude>`, `<dwr:filter>`, `<dwr:include>`, `<dwr:latencyfilter>`

`javascript`	The name that the bean will be known as in JavaScript.

`<dwr:signatures>`

Specifies Java signatures for exported methods. This can aid in the resolution of types stored in collections that are passed in as parameters. (See http://getahead.org/dwr/server/dwrxml/signatures for a discussion of how signatures work.)

May be included in: `<dwr:configuration>`

appendix D: Spring JSP tag library reference

When developing the view layer of a Spring MVC application, it's often necessary to bind form fields to a controller's command object. You may also want to resolve text from a properties file instead of hard-coding it in your JSP files.

This appendix catalogs the JSP tags that come with Spring 2, including the new form-binding tag library. Acegi's view-layer authorization tag library is also documented here.

D.1 Legacy Spring tag library

This is the tag library that is available with all versions of Spring. In this early tag library, form binding is all done through a single `<spring:bind>` tag. While this offered some rudimentary form-binding functionality, the `<spring:bind>` tag proved to be cumbersome in its use. Consequently, Spring 2 introduced a new set of form-binding tags.

Nevertheless, the legacy tag library is still useful for nonbinding activities such as resolving message properties and themes. And, of course, if your project hasn't made the move to Spring 2, the `<spring:bind>` tag is the only option that you have for form binding.

URI: http://www.springframework.org/tags

Usage: Add the following JSP tag library declaration to the JSP files that will be using this tag library:

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

<spring:bind>

Binds information about a command object (or a property of the command object) to the `status` variable.

Variable: `status`

Attribute	Description
`htmlEscape`	Specifies whether or not to perform HTML escaping on the values bound by this tag. Overrides any value set by `<spring:htmlEscape>`. *Valid values:* `true`, `false` *Default:* `false`
`ignoreNestedPath`	If `true`, specifies that nested paths should be ignored. *Valid values:* `true`, `false` *Default:* `false`
`path`	The path to the command object property.

<spring:escapeBody>

Applies HTML and/or JavaScript escaping to the enclosed content.

Attribute	Description
`htmlEscape`	Specifies whether or not HTML escaping should be applied. *Default:* determined by `<spring:htmlEscape>`
`javaScriptEscape`	Specifies whether or not JavaScript escaping should be applied. *Default:* `false`

<spring:hasBindErrors>

Binds the errors for an object to the `errors` variable. Like `<spring:bind>`, but only concerns itself with errors and not the name of a field or its value.

Attribute	Description
`htmlEscape`	Specifies whether or not HTML escaping is to be applied. *Default:* determined by `<spring:htmlEscape>`
`name`	The name of the object to be inspected for errors.

<spring:htmlEscape>

Sets the default HTML escape policy for the current page.

Atttribute	Description
`defaultHtmlEscape`	Whether or not to escape HTML by default. *Default:* `false` (no escaping)

<spring:message>

Supports externalization of messages using a Spring `MessageSource` configured in Spring.

Attribute	Description
`arguments`	Sets optional message arguments to be available when rendering the message.
`argumentSeparator`	The separator character to use when tokenizing `arguments`. *Default:* comma (`,`)
`code`	The message code to use when looking up a message.
`htmlEscape`	Whether or not to apply HTML escaping to the message. *Default:* determined by `<spring:htmlEscape>`
`javaScriptEscape`	Specifies whether or not to apply JavaScript escaping to the message. *Default:* `false`
`message`	Specifies a Spring `MessageSourceResolvable` object that will be used to resolve the message.

`<spring:message>` *(continued)*

Attribute	Description
`scope`	Used with `var` to determine the scope that the message variable will be created in.
`text`	The default text to render if the message cannot be found.
`var`	A variable to export the message. If not used, `message` is rendered directly to the output.

`<spring:nestedPath>`

Supports nested properties of the command object by exporting a `nestedPath` variable. You often don't need to use `nestedPath` directly, as it will be used by the other binding tags to construct the full path to the object property.

Attribute	Description
`path`	Sets the outer path that encloses the nested path.

`<spring:theme>`

Supports externalization of messages based on a theme. Resolves the message from a Spring theme.

Attribute	Description
`arguments`	Sets optional message arguments to be available when rendering the message.
`argumentSeparator`	Specifies the character used to separate values in `arguments`. Defaults to comma (`,`).
`code`	The message code to use when looking up a message.
`htmlEscape`	Specifies whether or not to apply HTML escaping to the message. *Default:* determined by `<spring:htmlEscape>`
`javaScriptEscape`	Specifies whether or not to apply JavaScript escaping to the message. *Default:* `false`
`message`	Specifies an argument to `MessageSourceResolvable`.
`scope`	Used with `var` to determine the scope that the message variable will be created in.
`text`	The default text to render if the message cannot be found.
`var`	A variable to export the message. If not used, `message` is rendered directly to the output.

<spring:transform>

Enables transformation of a value not contained with the command object using the `PropertyEditors` associated with the command object.

A common example is displaying a list of dates from a list to populate a `Date` property on the command object. Using `<spring:transform>` the list of dates can be formatted using the command object's `PropertyEditors`, even though the dates in the list aren't in the command object itself.

Attribute	Description
`htmlEscape`	Specifies whether or not to apply HTML escaping to the value. *Default:* determined by `<spring:htmlEscape>`
`scope`	Used with `var` to determine the scope that the formatted variable will be created in.
`value`	The value to be formatted.
`var`	A variable to export the formatted value. If not used, the formatted value is rendered directly to the output.

D.2 Form binding tags

Prior to Spring 2, the `<spring:bind>` tag and the `status` variable that it creates were the only way to bind command object properties to the fields of a form. But `<spring:bind>` is somewhat clumsy to use and was not as intuitive as the form tags offered by some other MVC frameworks.

Thankfully, Spring 2 includes a richer set of JSP tags for form binding. These tags are much clearer and simpler to use. This section serves as a reference for the new JSP form-binding tags.

Most of the new tags have a `path` attribute that binds them to a specific command object property. This attribute is the only one that is required. However, you may want to set additional attributes on the HTML element that is rendered. For that purpose, the form-binding tags include several HTML pass-through attributes that are simply used to set the same attribute on the rendered HTML.

Two pass-through attributes to take special note of are `cssClass` and `cssStyle`. These attributes pass through to the HTML `class` and `style` attributes, respectively.

Also, many of the form tags have a special `cssErrorClass` attribute. If the field has any errors associated with it then the HTML `class` attribute of the element will be set to the value specified by `cssErrorClass`.

URI: http://www.springframework.org/tags/form

Usage: Add the following JSP tag library declaration to the JSP files that will be using this tag library:

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

<form:checkbox>

Renders a check box (`<input type="checkbox">`) in HTML that is bound to a command object property.

HTML pass-though attributes: `accesskey`, `cssClass`, `cssErrorClass`, `cssStyle`, `dir`, `disabled`, `id`, `lang`, `onblur`, `onchange`, `onclick`, `ondblclick`, `onfocus`, `onkeydown`, `onkeypress`, `onkeyup` `onmousedown`, `onmousemove`, `onmouseout`, `onmouseover`, `onmouseup`, `tabindex`, `title`, `value`

Attribute	Description
`path`	The path to the command object property (relative to the command object set with `commandName` in `<form:form>`).

<form:errors>

Renders an HTML `<span>` tag containing field errors for the command field.

HTML pass-through attributes: `cssClass`, `cssStyle`, `delimiter`, `dir`, `id`, `lang`, `onclick`, `ondblclick`, `onkeydown`, `onkeypress`, `onkeyup`, `onmousedown`, `onmousemove`, `onmouseout`, `onmouseover`, `onmouseup`, `tabindex`, `title`

Attribute	Description
`element`	Specifies the HTML element that is used to render the enclosing errors.
`path`	The path to the command object property (relative to the command object set with `commandName` in `<form:form>`).

<form:form>

Renders an HTML `<form>` tag. Also creates a context within which the other Spring form tags are bound to the command object.

HTML pass-through attributes: `action`, `cssClass`, `cssStyle`, `dir`, `enctype`, `id`, `lang` `method`, `name`, `onclick`, `ondblclick`, `onkeydown`, `onkeypress`, `onkeyup`, `onmousedown`, `onmousemove`, `onmouseout`, `onmouseover`, `onmouseup`, `onreset`, `onsubmit`, `title`

Attribute	Description
`commandName`	The name of the command object that this form should be bound to.
`htmlEscape`	Enable/disable HTML escaping of rendered values.

<form:hidden>

Renders an HTML hidden field (<input type="hidden">) that is bound to a command object property.

HTML pass-through attributes: id

Attribute	Description
htmlEscape	Enable/disable HTML escaping of rendered values.
path	The path to the command object property (relative to the command object set with commandName in <form:form>).

<form:input>

Renders an HTML text field (<input type="text">) that is bound to a command object property.

HTML pass-through attributes: accesskey, alt, autocomplete, cssClass, cssErrorClass, cssStyle, dir, disabled, id, lang, maxlength, onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup, onselect, readonly, size, tabindex, title

Attribute	Description
htmlEscape	Enable/disable HTML escaping of rendered values.
path	The path to the command object property (relative to the command object set with commandName in <form:form>).

<form:label>

Renders a form field label (<label>).

HTML pass-through attributes: cssClass, cssErrorClass, cssStyle, dir, for, ID, lang, onclick, ondblclick, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup, tabindex, title

Attribute	Description
delimiter	The delimiter for rendering multiple error messages. *Default:*
path	The path to the command object property (relative to the command object set with commandName in <form:form>).

<form:option>

Renders an HTML <option> element. Sets the <option> element's selected attribute based on the value bound to <form:select>.

HTML pass-through attributes: disabled, label, value

Attribute	Description
htmlEscape	Enable/disable HTML escaping of rendered values.

`<form:options>`

Renders one or more HTML `<option>` elements from a collection.

Attribute	Description
`htmlEscape`	Enable/disable HTML escaping of rendered values.
`itemLabel`	The property name of the collection element that will contain the label of the option.
`items`	A collection containing objects that will be used as options.
`itemValue`	The property name of the collection element that will contain the value of the option.

`<form:password>`

Renders an HTML password field (`<input type="password">`) that is bound to a property of the command object.

HTML pass-through attributes: `accesskey`, `alt`, `autocomplete`, `cssClass`, `cssErrorClass`, `cssStyle`, `dir`, `disabled`, `id`, `lang`, `maxlength`, `onblur`, `onchange`, `onclick`, `ondblclick`, `onfocus`, `onkeydown`, `onkeypress`, `onkeyup`, `onmousedown`, `onmousemove`, `onmouseout`, `onmouseover`, `onmouseup`, `onselect`, `readonly`, `size`, `tabindex`, `title`

Attribute	Description
`htmlEscape`	Enable/disable HTML escaping of rendered values.
`path`	The path to the command object property (relative to the command object set with `commandName` in `<form:form>`).
`showPassword`	If set to `true`, the password will be displayed. Defaults to `false`.

`<form:radiobutton>`

Renders an HTML radio button (`<input type="radio">`) that is bound to a property of the command object.

HTML pass-through attributes: `accesskey`, `cssClass`, `cssErrorClass`, `cssStyle`, `dir`, `disabled`, `id`, `lang`, `onblur`, `onchange`, `onclick`, `ondblclick`, `onfocus`, `onkeydown`, `onkeypress`, `onkeyup`, `onmousedown`, `onmousemove`, `onmouseout`, `onmouseover`, `onmouseup`, `tabindex`, `title`, `value`

Attribute	Description
`htmlEscape`	Enable/disable HTML escaping of rendered values.
`path`	The path to the command object property (relative to the command object set with `commandName` in `<form:form>`).

<form:select>

Renders an HTML <select> element that is bound to a property of the command object.

HTML pass-through attributes: accesskey, cssClass, cssErrorClass, cssStyle, dir, disabled, id, lang, multiple, onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup, size, tabindex, title

Attribute	Description
htmlEscape	Enable/disable HTML escaping of rendered values.
itemLabel	Name of the property mapped to the inner text of the option tag
items	The collection, map, or array of objects used to generate the inner option tags
itemValue	Name of the property mapped to value attribute of the option tag
path	The path to the command object property (relative to the command object set with commandName in <form:form>).

<form:textarea>

Renders an HTML <textarea> element that is bound to a property of the command object.

HTML pass-through attributes: accesskey, cols, cssClass, cssErrorClass, cssStyle, dir, disabled, id, lang, onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup, onselect, readonly, rows, tabindex, title

Attribute	Description
htmlEscape	Enable/disable HTML escaping of rendered values.
path	The path to the command object property (relative to the command object set with commandName in <form:form>).

D.3 Acegi's authorization tag library

The Acegi Security Framework provides a handful of JSP tags that are used to conditionally display information in the rendered view, depending on the user's permissions.

URI: http://acegisecurity.org/authz

Usage: Add the following JSP tag library declaration to the JSP files that will be using this tag library:

```
<%@ taglib prefix="authz" uri="http://acegisecurity.org/authz" %>
```

<authz:acl>, <authz:accesscontrollist>

Conditionally renders a tag body if the user has one of the specified permissions to the domain object.

Attribute	Description
`domainObject`	The domain object for which permissions are being evaluated.
`hasPermission`	A comma-separated list of integers, each pertaining to a required bit mask permission from a subclass of `AbstractBasicAclEntry`.

<authz:authentication>

Renders information about the user. The user information is retrieved from the object returned from `Authorization.getPrincipal()`, which is often an instance of `UserInfo`.

Attribute	Description
`methodPrefix`	A prefix to apply to `operation` to determine the method that will be called. *Valid values:* `get`, `is` *Default:* `get`
`operation`	Combined with `methodPrefix` to determine a method to call on the user's `Authentication` object.

<authz:authorize>

Conditionally renders the body of the tag, depending on whether or not the user has been granted certain authorities.

Attribute	Description
`ifAllGranted`	A comma-separated list of authorities, all of which the user must have for the tag body to be rendered.
`ifAnyGranted`	A comma-separated list of authorities, of which the user must be granted at least one for the tag body to be rendered.
`ifNotGranted`	A comma-separated list of authorities, of which the user must not be granted any for the tag body to be rendered.

appendix E: Spring Web Flow definition reference

Spring Web Flow is an exciting new addition to the Spring family. It enables developers to build conversational web applications where the application's overall flow is defined separately from the application's logic and view code.

XML is the mechanism typically used to define a flow. This appendix serves as a reference to Spring Web Flow's XML schema. It's very important to understand that the elements in this appendix are *not* bean definition elements and should *not* appear within a Spring application context definition. A flow definition is defined in a completely separate XML file from those that contain bean definitions.

Schema: http://www.springframework.org/schema/webflow

Usage: The elements described in this appendix are specifically for defining a flow to be executed in Spring Web Flow. These are not bean definition elements and should not be used within a Spring application context definition. The root of a Spring Web Flow definition file is the `<flow>` element. It should appear as follows:

```
<flow xmlns="http://www.springframework.org/schema/webflow"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation=
        "http://www.springframework.org/schema/webflow
         http://www.springframework.org/schema/
              webflow/spring-webflow-1.0.xsd
…
</flow>
```

<action>

Defines a flow action to be executed.

May be included in: `<action-state>`, `<end-actions>`, `<entry-actions>`, `<exit-actions>`, `<render-actions>`, `<start-actions>`, `<transition>`

May contain: `<attribute>`

Attribute	Description
`bean`	Refers to the ID of a bean in the Spring application context.
`method`	The name of a method to invoke on the action if the action extends `MultiAction`. The method should have a signature in the following form: `public Event <methodName>( RequestContext context);`
`name`	An optional name for the action. If used the re-sent event will be qualified by this name. For example, if this action is named `doStuff` and signals a `success` result event, the fully qualified event will be `doStuff.success`.

<action-state>

Defines an action state. The action performed by this action state is specified by the nested <action> or <bean-action> elements.

May be included in: <flow>

May contain: <action>, <attribute>, <bean-action>, <entry-actions>, <evaluate-action>, <exception-handler>, <exit-action>, <set>, <transition>

Attribute	Description
id	The ID of the action state.

<argument>

Defines a specific argument to be passed to a <bean-action>.

May be included in: <method-arguments>

Attribute	Description
expression	An expression specifying a value to be passed to the method invoked in the <bean-action>.
parameter-type	The type of the method parameter. If specified and the argument value does not match the parameter type, a type conversion will be attempted.

<attribute>

Declares an attribute that describes a flow, state, or transition.

May be included in: <action>, <action-state>, <bean-action>, <decision-state>, <end-state>, <flow>, <subflow-state>, <transition>, <view-state>

May contain: <value>

Attribute	Description
name	The name of the attribute.
type	The type of the attribute value, used to facilitate type conversion from a String value.
value	The value of the attribute. Can be used instead of a <value> child element.

<attribute-mapper>

Declares an attribute mapping to and from a subflow.

May be included in: <subflow-state>

May contain: <input-mapper>, <output-mapper>

Attribute	Description
bean	Refers to a custom mapper as a bean in the Spring application context. May be used instead of child <input-mapper> and <output-mapper> elements.

`<bean-action>`

Specifies a flow action as a bean/method combination. This is a lightweight alternative to implementing Spring Web Flow's `Action` interface as it allows any bean to participate as an action in a flow.

May be included in elements: `<action-state>`, `<end-actions>`, `<entry-actions>`, `<exit-actions>`, `<render-actions>`, `<start-actions>`, `<transition>`

May contain: `<attribute>`, `<method-arguments>`, `<method-result>`

Attribute	Description
`bean`	Refers to a bean in the Spring application context to be invoked.
`method`	The bean method to be invoked.
`name`	An optional name for the action. If used, the re-sent event will be qualified by this name. For example, if this action is named `doStuff` and signals a `success` result event, the fully qualified event will be `doStuff.success`.

`<decision-state>`

Defines a decision state. Conditions and the resulting transitions are specified by one or more child `<if>` elements.

May be included in: `<flow>`

May contain: `<attribute>`, `<entry-actions>`, `<exception-handler>`, `<exit-actions>`, `<if>`

Attribute	Description
`id`	The ID of the state.

`<end-actions>`

Specifies a collection of one or more actions to be performed as a flow ends.

May be included in: `<flow>`

May contain: `<action>`, `<bean-action>`, `<evaluate-action>`, `<set>`

`<end-state>`

Defines the end state of the flow. Upon entering an end state, the conversation is over and the user is presented with a page specified by the `view` attribute.

May be included in: `<flow>`

May contain: `<attribute>`, `<entry-actions>`, `<exception-handler>`, `<output-mapper>`

Attribute	Description
`id`	The ID of the state.
`view`	The view to be presented to the user when the flow concludes. Refers to a logical view name that can be resolved by a Spring MVC view resolver.

<entry-actions>

Declares one or more actions to be performed upon entry to a flow state.

May be included in: <action-state>, <decision-state>, <end-state>, <subflow-state>, <view-state>

May contain: <action>, <bean-action>, <evaluate-action>, <set>

<evaluate-action>

Defines an action as an arbitrary expression against the flow request context. Can be used to invoke any method on a flow-managed bean.

May be included in: <action-state>, <end-actions>, <entry-actions>, <exit-actions>, <render-actions>, <start-actions>, <transition>

May contain: <evaluation-result>

Attribute	Description
expression	An expression that references a method of a flow-scoped bean.
name	An optional name qualifier for this evaluate action. When specified, this action will qualify execution result event identifiers by prefixing them with this name.

<evaluation-result>

Specifies how the result of <evaluate-action> will be exposed to the flow.

May be included in: <evaluate-action>

Attribute	Description
name	The name of the scoped variable to hold the result.
scope	The scope within which the result should reside. *Valid values:* request, flash, flow, conversation, default *Default value:* default

<exception-handler>

Specifies an exception handler for a flow or state.

May be included in: <action-state>, <decision-state>, <end-state>, <flow>, <subflow-state>, <view-state>

Attribute	Description
bean	Refers to a bean in the Spring application context that is a custom exception handler, implementing either StateExceptionHandler or FlowExecutionExceptionHandler.

<exit-actions>

Specifies one or more actions to be performed before transitioning away from a state.

May be included in: <action-state>, <decision-state>, <subflow-state>, <view-state>

May contain: <action>, <bean-action>, <evaluate-action>, <set>

<flow>

Defines a flow. This is the root element of a flow definition.

May be included in: <inline-flow>

May contain: <action-state>, <attribute>, <decision-state>, <end-actions>, <end-state>, <exception-handler>, <global-transitions>, <import>, <inline-flow>, <input-mapper>, <output-mapper>, <start-actions>, <start-state>, <subflow-state>, <var>, <view-state>

<global-transitions>

Defines one or more transitions that can be used throughout a flow (by all states).

May be included in: <flow>

May contain: <transition>

<if>

Defines a condition and resulting transition for a <decision-state>.

May be included in: <decision-state>

Attribute	Description
else	An optional state to transition to if the test expression evaluates to false.
test	A boolean expression defining criteria to be tested.
then	The state to transition to if the expression evaluates to true.

<import>

Imports a flow definition into the current flow. Encourages flow reuse.

May be included in: <flow>

Attribute	Description
resource	The resource containing the flow definition to be imported.

<inline-flow>

Defines an inline flow.

May be included in: <flow>

May contain: <flow>

Attribute	Description
id	The ID of the inline flow.

<input-attribute>

Defines an input attribute.

May be included in: <input-mapper>

Attribute	Description
name	The name of the input attribute.
required	Specifies whether or not this input attribute is required.
scope	The scope of the input attribute. If not specified, the default scope type is used.

<input-mapper>

Defines an input mapper for a flow or subflow.

May be included in: <attribute-mapper>, <flow>
May contain: <input-attribute>, <mapping>

<mapping>

Defines a mapping rule, mapping the value of a source expression to a property of a target data structure.

May be included in: <input-mapper>, <output-mapper>

Attribute	Description
from	The source value type. A type conversion will be performed if the source type differs from the target type.
required	If true, this is a required mapping. An error will occur if the source is null.
source	An expression that resolves to the value to be mapped.
target	An expression that defines the target property to be set.
target-collection	An expression that defines a collection that the mapped value should be added to. (Use this instead of target.)
to	The target value type. A type conversion will be performed if the target value differs from the source type.

<method-arguments>

Declares a collection of one or more arguments to be passed in an invocation of a <bean-action>.

May be included in: <bean-action>
May contain: <argument>

<method-result>

Specifies where the result of a <bean-action> invocation should be placed.

May be included in: <bean-action>

Attribute	Description
name	The name of an attribute that will contain the return value of the target <bean-action> method.
scope	The scope of the return value attribute. *Valid values:* request, flash, flow, conversation, default *Default value:* default

<output-attribute>

Defines an output attribute.

May be included in: <output-mapper>

Attribute	Description
name	The name of the output attribute.
required	Specifies whether or not this output attribute is required.
scope	The scope of the output attribute. If not specified, the default scope type is used.

<output-mapper>

Defines an output mapper for a flow or subflow.

May be included in: <attribute-mapper>, <end-state>, <flow>
May contain: <mapping>, <output-attribute>

<render-actions>

Specifies one or more actions to be performed prior to rendering the view of a view state.

May be included in: <view-state>
May contain: <action>, <bean-action>, <evaluate-action>, <set>

<set>

Sets a scoped attribute value.

May be included in: <action-state>, <end-actions>, <entry-actions>, <exit-actions>, <render-actions>, <start-actions>, <transition>
May contain: <attribute>

Attribute	Description
attribute	The name of the attribute to set. May be a nested path using JavaBeans notation.
name	An optional name qualifier for this set action. When specified, this action will qualify execution result event identifiers by prefixing them with this name.

`<set>` *(continued)*

Attribute	Description
`scope`	The scope of the attribute. Valid values: `default`, `request`, `flash`, `flow`, `conversation` *Default value:* `default`
`value`	The attribute value expression.

`<start-actions>`

Specifies one or more actions to be performed as a flow begins.

May be included in: `<flow>`

May contain: `<action>`, `<bean-action>`, `<evaluate-action>`, `<set>`

`<start-state>`

Declares the beginning state of a flow.

May be included in: `<flow>`

`idref`	References the ID of the state that starts the flow.

`<subflow-state>`

Declares a subflow state. The execution of the current flow is suspended and a new subflow begins.

May be included in: `<flow>`

May contain: `<attribute>`, `<attribute-mapper>`, `<entry-actions>`, `<exception-handler>`, `<exit-actions>`, `<transition>`

Attribute	Description
`flow`	The name of the subflow.
`id`	The ID of the subflow state.

`<transition>`

Defines a transition.

May be included in: `<action-state>`, `<global-transitions>`, `<subflow-state>`, `<view-state>`

May contain: `<action>`, `<attribute>`, `<bean-action>`, `<evaluate-action>`, `<set>`

Attribute	Description
`on`	The criteria that triggers this transition. Typically, a static value that indicates the last event that occurred in the flow. May also be a `boolean` expression.
`on-exception`	The fully qualified class name of an exception type that should trigger the transition. (Used instead of the `on` attribute.)
`to`	The ID of a state that the flow should transition to upon being triggered.

`<value>`

The value of an attribute (as text contained in this element). This is the longhand alternative to the `value` attribute of the `<attribute>` element.

May be included in: `<attribute>`

`<var>`

Defines a flow variable. Flow variables are automatically created when a flow starts.

May be included in: `<flow>`

Attribute	Description
`bean`	References a bean in the Spring application context that defines the initial flow variable value. The bean must not be scoped as a singleton. Not required if the `name` attribute matches the bean ID.
`class`	An alternative to the `bean` attribute that specifies the type of the variable to be instantiated directly.
`name`	The name of the variable. When used without the `bean` or `class` attribute, this name is also used as the ID of a bean in the Spring application context that will be used as the variable's initial value. (The bean must not be scoped as a singleton.)
`scope`	The scope of the variable. *Valid values:* `request`, `flash`, `flow`, `conversation`, `default` *Default value:* `default`

`<view-state>`

Declares a view state in the flow. Displays a view to the user.

May be included in: `<flow>`

May contain: `<attribute>`, `<entry-actions>`, `<exception-handler>`, `<exit-actions>`, `<render-actions>`, `<transition>`

Attribute	Description
`id`	The ID of the view state.
`view`	The logical name of the view (to be looked up using a Spring MVC view resolver).

appendix F: Customizing Spring configuration

One of the most significant new features in Spring 2 is the ability to create custom Spring configuration XML elements. No more will you be limited to only `<bean>` elements. With Spring 2 you can extend Spring's configuration with custom elements that are more terse and clearer to read.

Throughout the printed book, you found examples of how common Spring configuration tasks have been greatly simplified by the addition of new elements that come packaged as part of the Spring 2 distribution. For example, in chapter 4 you saw how aspects can be declared with elements like `<aop:aspect>` and `<aop:pointcut>`. And in chapter 5, we looked at how to declare transactions with `<tx:advice>`.

While the prepackaged configuration elements are very handy, you may find that you'd like to create your own. To illustrate how custom configuration elements are useful, let's first look at a `<bean>` that is declared without using a custom element:

```
<bean id="babelFish" class=
   ➥ "org.springframework.remoting.jaxrpc.
        ➥ JaxRpcPortProxyFactoryBean">
  <property name="wsdlDocumentUrl"
      value="http://www.xmethods.com/sd/2001/BabelFishService.wsdl" />
  <property name="portInterface"
      value="com.sia.tryit.BabelFishRemote" />
  <property name="serviceInterface"
      value="com.sia.tryit.BabelFishService" />
  <property name="namespaceUri"
      value="http://www.xmethods.net/sd/BabelFishService.wsdl" />
  <property name="serviceName" value="BabelFishService" />
  <property name="portName" value="BabelFishPort" />
</bean>
```

This `<bean>` declaration exhibits a few shortcomings of using plain-vanilla `<bean>` and `<property>` elements to declare a Spring bean:

- It's not immediately evident what this bean does. You learned in chapter 8 that a `JaxRpcPortProxyFactoryBean` is used to configure a web service in Spring. But if this is the first time you've seen `JaxRpcPortProxyFactoryBean`, you're probably scratching your head wondering what all of that XML means.
- The XML is verbose. `JaxRpcPortProxyFactoryBean` requires that several properties be set. While the `<property>` element gets the job done, it does consume a lot of space in the configuration file.

- Although you may not fully understand what a `JaxRpcPortProxyFactoryBean` does, you do know that is the class being used. Although this is not typically a problem when developing Spring applications, it may be too much information if you're developing a Spring-based component library. As it is, the client of your component library knows exactly which class implements the component. If you later decide to switch to a different implementation class (or to even rename/repackage the class), your users would have to change their code to reflect your API changes.

At the time of this writing, there's no other way to declare a `JaxRpcPortProxyFactoryBean` in Spring other than to use `<bean>` and `<property>` elements. Spring doesn't come with a configuration element specifically for configuring web services. This presents us with an opportunity to develop one as an example of how to develop custom Spring configuration elements.

In Spring, one or more configuration elements are grouped together under a namespace. There are five steps to creating a custom Spring XML namespace:

1. Define the namespace grammar using XML Schema.
2. Create a namespace handler class.
3. Create a bean definition parser for each of the elements in the namespace.
4. Declare the namespace and its handler for deployment.
5. Bundle the custom configuration in a JAR file.

The first thing we must do is to decide what our custom XML should look like. For that, we'll define an XML grammar using XML schema.

F.1 Defining a namespace

As you've seen, the XML required to declare a `JaxRpcPortProxyFactoryBean` is a bit too verbose and nonexpressive for our taste. Wouldn't it be better if we could declare a web service in Spring using something like this:

```
<ws:proxy id="foo"
    wsdlDocumentUrl=
        ➥ "http://www.xmethods.com/sd/2001/BabelFishService.wsdl"
    portInterface="com.sia.tryit.BabelFishRemote"
    serviceInterface="com.sia.tryit.BabelFishService"
    namespaceUri="http://www.xmethods.net/sd/BabelFishService.wsdl"
    serviceName="BabelFishService"
    portName="BabelFishPort"
  />
```

This <ws:proxy> element carries the same information as the more verbose <bean> declaration, but is much more terse. And the name of the element gives us some clue as to what it does.

XML Schema is an appropriate mechanism for defining the grammar of one or more XML elements. Listing F.1 shows an XML Schema that defines the <proxy> element.

Listing F.1 Defining the XML grammar for a custom configuration element

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema
    xmlns="http://www.springinaction.com/schema/spring/webservice"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace=
        "http://www.springinaction.com/schema/spring/webservice"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified">

  <xsd:element name="proxy">   <- Declares <proxy> element
    <xsd:complexType>
      <xsd:attribute name="id" type="xsd:string" />
      <xsd:attribute name="wsdlDocumentUrl" type="xsd:string"
          use="required" />
      <xsd:attribute name="portInterface" type="xsd:string"
          use="required" />
      <xsd:attribute name="serviceInterface" type="xsd:string"
          use="required" />                                      Declares
      <xsd:attribute name="namespaceUri" type="xsd:string"       attributes of
          use="required" />                                      <proxy>
      <xsd:attribute name="serviceName" type="xsd:string"
          use="required" />
      <xsd:attribute name="portName" type="xsd:string"
          use="required" />
      <xsd:attribute name="serviceFactoryClass"
          type="xsd:string" />
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

The attributes of the <proxy> element mirror the properties of the JaxRpcPortProxyFactoryBean class. All of them are described in XML as xsd:string and most of them are required.

This brings up another benefit of a custom configuration element. Although JaxRpcPortProxyFactoryBean will verify the fields that are required at runtime, this XML Schema will help Schema-aware XML editors catch missing attributes as they're being edited.

NOTE Although I wrote the XSD file in listing F.1 by hand, you may want to consider using an XSD inference tool. This is especially true if you're unfamiliar with XSD or if your custom XML will be quite complex. An XSD inference tool takes an example XML file and makes reasonable guesses to generate an XSD schema that will validate the example XML. Several XML editors come with an XSD inference feature. For a standalone XSD inference tool, I like Trang (http://www.thaiopensource.com/relaxng/trang.html).

XML Schema defines the grammar used to express the custom configuration element(s). But there needs to be some logic written to tell Spring what to do with those elements. Next, we'll create a namespace handler that will put some meaning behind the grammar.

F.2 *Creating namespace handlers*

The simplest and most common way to build a namespace handler is to subclass Spring's `NamespaceHandlerSupport` class. `WebServiceNamespaceHandler` (listing F.2) extends `NamespaceHandlerSupport` to tell Spring how to interpret our custom `<proxy>` element.

Listing F.2 A namespace handler for handling web service `<proxy>` elements

```
package com.springinaction.config;
import org.springframework.beans.factory.xml.
    ➥ NamespaceHandlerSupport;

public class WebServiceNamespaceHandler
    extends NamespaceHandlerSupport {
  public WebServiceNamespaceHandler() {}

  public void init() {
    registerBeanDefinitionParser("proxy",
        new WebServiceProxyBeanDefinitionParser());
  }
}
```

Maps WebServiceProxyBeanDefinitionParser to <proxy>

Namespace handler classes typically don't process the individual elements. Instead, a namespace handler dispatches element parsing to one or more bean definition parsers. In the case of `WebServiceNamespaceHandler`, we register

WebServiceProxyBeanDefinitionParser as the class that will do the actual handling of the <proxy> element. WebServiceProxyBeanDefinitionParser is shown in listing F.3.

Listing F.3 definition parser for the <proxy> element

```
package com.springinaction.config;
import org.springframework.beans.MutablePropertyValues;
import org.springframework.beans.factory.config.BeanDefinition;
import org.springframework.beans.factory.support.
    ➥ BeanDefinitionReaderUtils;
import org.springframework.beans.factory.support.
    ➥ BeanDefinitionRegistry;
import org.springframework.beans.factory.support.RootBeanDefinition;
import org.springframework.beans.factory.xml.
    ➥ AbstractBeanDefinitionParser;
import org.springframework.beans.factory.xml.ParserContext;
import org.springframework.remoting.jaxrpc.
    ➥ JaxRpcPortProxyFactoryBean;
import org.springframework.util.StringUtils;
import org.w3c.dom.Element;

public class WebServiceProxyBeanDefinitionParser
    extends AbstractBeanDefinitionParser {

  private static final String ID = "id";
  private static final String WSDL_URL = "wsdlDocumentUrl";
  private static final String PORT_INTERFACE = "portInterface";
  private static final String SERVICE_INTERFACE =
      "serviceInterface";
  private static final String NAMESPACE_URI = "namespaceUri";
  private static final String SERVICE_NAME = "serviceName";
  private static final String PORT_NAME = "portName";
  private static final String SERVICE_FACTORY_CLASS =
      "serviceFactoryClass";

  protected BeanDefinition parseInternal(
      Element element, ParserContext context) {
    BeanDefinitionRegistry registry = context.getRegistry();

    RootBeanDefinition beanDef = new RootBeanDefinition();      Creates bean
    beanDef.setBeanClass(JaxRpcPortProxyFactoryBean.class);     definition

    String id = element.getAttribute(ID);
    if(!StringUtils.hasText(id)) {
      id = BeanDefinitionReaderUtils.generateBeanName(         Sets bean ID
          beanDef, registry, false);
    }
    MutablePropertyValues mpv = new MutablePropertyValues();
  mpv.addPropertyValue(WSDL_URL,
    element.getAttribute(WSDL_URL));
  mpv.addPropertyValue(PORT_INTERFACE,
    element.getAttribute(PORT_INTERFACE));
  mpv.addPropertyValue(SERVICE_INTERFACE,
    element.getAttribute(SERVICE_INTERFACE));
  mpv.addPropertyValue(NAMESPACE_URI,
    element.getAttribute(NAMESPACE_URI));
  mpv.addPropertyValue(SERVICE_NAME,
    element.getAttribute(SERVICE_NAME));

  String portName = element.getAttribute(PORT_NAME);
  if(StringUtils.hasText(portName)) {
   mpv.addPropertyValue(PORT_NAME, portName);
  }

  String serviceFactory =
    element.getAttribute(SERVICE_FACTORY_CLASS);
  if(StringUtils.hasText(serviceFactory)) {
   mpv.addPropertyValue(
     SERVICE_FACTORY_CLASS, serviceFactory);
  }

  beanDef.setPropertyValues(mpv);

  registry.registerBeanDefinition(id, beanDef);

  return beanDef;
 }
}
```

Sets required attributes

Sets optional attributes

Registers bean definition

The primary job of a bean definition parser is to parse the XML of a configuration element and to register one or more beans with the Spring container. `WebServiceProxyBeanDefinitionParser`'s whole purpose in life is to simplify configuration of `JaxRpcPortProxyFactoryBean`, so that is the class of the bean that it will register.

`WebServiceProxyBeanDefinitionParser` starts by creating a bean definition object and setting that bean definition's class to `JaxRpcPortProxyFactoryBean`. It then looks at each of the attributes for the `<proxy>` element (as defined in the XML Schema) and maps their values to properties of the bean definition (by way of a `MutablePropertyValues` object). Once all of the properties have been set, it registers the bean definition with the bean definition registry. The Spring container takes over from there and creates the bean.

F.2.1 Writing a simple bean definition parser

One thing that may have caught your eye about the WebServiceProxyBeanDefinitionParser is that it directly maps the attributes of the <proxy> element to properties of the same name on JaxRpcPortProxyFactoryBean. For example, the wsdlDocumentUrl attribute of the <proxy> element gets mapped directly to the wsdlDocumentUrl property on JaxRpcPortProxyFactoryBean. When you are defining a custom element that is doing a simple one-to-one mapping of attributes to properties such as this, there is a simpler way to define the bean definition parser.

Spring's AbstractSimpleBeanDefinitionParser is a special implementation of a bean implementation parser that automatically does one-to-one mappings from configuration attributes to bean properties. Listing F.4 shows a new implementation of WebServiceProxyBeanDefinitionParser that is made much simpler by extending AbstractSimpleBeanDefinitionParser.

Listing F.4 A simpler form of a web service bean definition parser

```
package com.springinaction.config;
import org.springframework.beans.factory.xml.
         ➥ AbstractSimpleBeanDefinitionParser;
import org.springframework.remoting.jaxrpc.
    ➥ JaxRpcPortProxyFactoryBean;
import org.w3c.dom.Element;

public class WebServiceProxyBeanDefinitionParser
    extends AbstractSimpleBeanDefinitionParser {
  protected Class getBeanClass(Element element) {
    return JaxRpcPortProxyFactoryBean.class;
  }
}
```

The only method that you must implement when extending AbstractSimpleBeanDefinitionParser is getBeanClass(). This method tells AbstractSimpleBeanDefinitionParser what bean you want configured in the Spring context. In this case, we tell it that we're configuring a JaxRpcPortProxyFactoryBean.

F.3 Packaging custom configuration elements

Now that you've defined the grammar of the custom XML element and have written a namespace handler and a bean definition parser to process the element, we're almost ready to bundle up the configuration element for use in any project that needs to declare a web service bean. But first, we need a way to tell Spring about the namespace and how it should be handled.

The Spring container looks for two files under the META-INF/ directory in its classpath to give it clues about custom configuration elements. These two files are:

- *spring.schemas*—Maps a schema URI to an actual XML Schema in the classpath
- *spring.handlers*—Maps a namespace URI to the namespace handler class that will process configuration elements in that namespace

Custom namespaces are declared in a Spring configuration file using the standard approach for declaring a namespace in any XML file. For example, if you consider a typical Spring configuration file, you might find that the `<beans>` element is written as follows:

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:aop="http://www.springframework.org/schema/aop"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/
      ➥ spring-beans-2.0.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/
      ➥ spring-aop-2.0.xsd">
…
</beans>
```

In this example, there are two namespaces in play:

- The default namespace is the Spring `beans` namespace. This namespace's URI is `http://www.springframework.org/schema/beans` and its schema URI is `http://www.springframework.org/schema/beans/spring-beans-2.0.xsd`.
- The `aop` namespace references Spring's set of configuration elements for aspect-oriented programming (see chapter 4 of the printed book). Its URI is `http://www.springframework.org/schema/aop` and its schema URI is `http://www.springframework.org/schema/aop/spring-aop-2.0.xsd`.

When Spring encounters a custom namespace in its configuration file, it will use the mapping(s) defined in the spring.schemas file to map the namespace's logical schema URI to a physical XML Schema definition in the classpath. For our custom namespace, we want to map the logical schema URI `http://www.springinaction.com/schema/spring/webservice/spring-webservice.xsd` to the physical spring-webservice.xsd file defined in listing F.1. Consequently, here's what the spring.schemas file will look like to declare that mapping:

```
http\://www.springinaction.com/schema/spring/webservice/spring-
   ➥ webservice.xsd=com/springinaction/config/spring-webservice.xsd
```

Here we're saying that the XML Schema definition can be found in the classpath in com/springinaction/config.

But the schema definition isn't enough for Spring to be able to use your custom configuration element. You also must tell Spring which namespace handler to use when processing the XML. That's what the spring.handlers file is for. This file maps a namespace URI to the fully qualified class name of the namespace handler that processes the namespace's XML. Here's what the spring.handlers file looks like to map `http://www.springinaction.com/schema/spring/webservice` to be handled by `WebServiceNamespaceHandler`:

```
http\://www.springinaction.com/schema/spring/webservice=
    com.springinaction.config.WebServiceNamespaceHandler
```

With spring.schemas and spring.handlers defined, you're ready to package the custom configuration namespace.

F.3.1 Packaging the custom namespace

At this point, you could begin using the custom configuration namespace in your application as is. As long as all of the classes and files created in this section are in the proper location in the classpath, they'll be available to your application. But the real benefit of creating a custom namespace is in creating reusable configuration elements that can be used across many projects. Therefore, you'll want to create a JAR file containing the namespace code.

You can use any number of mechanisms for producing the JAR file, including Ant's `<jar>` task, running Maven 2's "package" goal, or simply using the `jar` utility at the command line. Regardless of which approach you take, you'll want to be sure that the structure of the JAR file matches what Spring will expect. To recap, here's the JAR file structure for the web service namespace we created in this section:

```
/META-INF/spring.handlers
/META-INF/spring.schemas
/com/springinaction/config/spring-webservice.xsd
/com/springinaction/config/WebServiceNamespaceHandler.class
/com/springinaction/config/WebServiceProxyBeanDefinitionParser.class
```

Now place the JAR file in your application's classpath and you're ready to use the new configuration element.

F.3.2 *Using the custom namespace*

As with Spring's built-in configuration elements, you'll need to declare your custom namespace in the <beans> element:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:ws="http://www.springinaction.com/schema/spring/webservice"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/
            ➥ spring-beans-2.0.xsd
        http://www.springinaction.com/schema/spring/webservice
        http://www.springinaction.com/schema/spring/webservice/
            ➥ spring-webservice.xsd">
…
</beans>
```

Here we've bound the namespace to the ws prefix (short for *web service*). Declared like this, the <proxy> element can be used as follows:

```
<ws:proxy id="foo"
    wsdlDocumentUrl=
        "http://www.xmethods.com/sd/2001/BabelFishService.wsdl"
    portInterface="com.sia.tryit.BabelFishRemote"
    serviceInterface="com.sia.tryit.BabelFishService"
    namespaceUri="http://www.xmethods.net/sd/BabelFishService.wsdl"
    serviceName="BabelFishService"
    portName="BabelFishPort"
  />
```

As you can see, the <ws:proxy> element is significantly simpler than the equivalent JaxRpcPortProxyFactoryBean declaration. It's also much more expressive in that <ws:proxy> is clear in its purpose.