20. Portlet MVC Framework

20.1 Introduction

JSR-286 The Java Portlet Specification

For more general information about portlet development, please review the JSR-286 Specification.

In addition to supporting conventional (servlet-based) Web development, Spring also supports JSR-286 Portlet development. As much as possible, the Portlet MVC framework is a mirror image of the Web MVC framework, and also uses the same underlying view abstractions and integration technology. So, be sure to review the chapters entitled Chapter 17, *Web MVC framework* and Chapter 18, *View technologies* before continuing with this chapter.

Bear in mind that while the concepts of Spring MVC are the same in Spring Portlet MVC, there are some notable differences created by the unique workflow of JSR-286 portlets.

The main way in which portlet workflow differs from servlet workflow is that the request to the portlet can have two distinct phases: the action phase and the render phase. The action phase is executed only once and is where any 'backend' changes or actions occur, such as making changes in a database. The render phase then produces what is displayed to the user each time the display is refreshed. The critical point here is that for a single overall request, the action phase is executed only once, but the render phase may be executed multiple times. This provides (and requires) a clean separation between the activities that modify the persistent state of your system and the activities that generate what is displayed to the user.

Spring Web Flow

Spring Web Flow (SWF) aims to be the best solution for the management of web application page flow.

SWF integrates with existing frameworks like Spring MVC, Struts, and JSF, in both servlet and portlet environments. If you have a business process (or processes) that would benefit from a conversational model as opposed to a purely request model, then SWF may be the solution.

SWF allows you to capture logical page flows as self-contained modules that are reusable in different situations, and as such is ideal for building web application modules that guide the user through controlled navigations that drive business processes.

For more information about SWF, consult the Spring Web Flow website.

The dual phases of portlet requests are one of the real strengths of the JSR-286 specification. For example, dynamic search results can be updated routinely on the display without the user explicitly rerunning the search. Most other portlet MVC frameworks attempt to completely hide the two phases from the developer and make it look as much like traditional servlet development as possible - we think this approach removes one of the main benefits of using portlets. So, the separation of the two phases is preserved throughout the Spring Portlet MVC framework. The primary manifestation of this approach is that where the servlet version of the MVC classes will have one method that deals with the request, the portlet version of the MVC classes will have two methods that deal with the request: one for the action phase and one for the render phase. For example, where the servlet version of `AbstractController` has the `handleRequestInternal(..)` method, the portlet version of `AbstractController` has `handleActionRequestInternal(..)` and `handleRenderRequestInternal(..)` methods.

The framework is designed around a `DispatcherPortlet` that dispatches requests to handlers, with configurable handler mappings and view resolution, just as the `DispatcherServlet` in the web framework does. File upload is also supported in the same way.

Locale resolution and theme resolution are not supported in Portlet MVC - these areas are in the purview of the portal/portlet container and are not appropriate at the Spring level. However, all mechanisms in Spring that depend on the locale (such as internationalization of messages) will still function properly because `DispatcherPortlet` exposes the current locale in the same way as `DispatcherServlet`.

20.1.1 Controllers - The C in MVC

The default handler is still a very simple `Controller` interface, offering just two methods:

- `void handleActionRequest(request,response)`
- `ModelAndView handleRenderRequest(request,response)`

The framework also includes most of the same controller implementation hierarchy, such as `AbstractController`, `SimpleFormController`, and so on. Data binding, command object usage, model handling, and view resolution are all the same as in the servlet framework.

20.1.2 Views - The V in MVC

All the view rendering capabilities of the servlet framework are used directly via a special bridge servlet named `ViewRendererServlet`. By using this servlet, the portlet request is converted into a servlet request and the view can be rendered using the entire normal servlet infrastructure. This means all the existing renderers, such as JSP, Velocity, etc., can still be used within the portlet.

20.1.3 Web-scoped beans

Spring Portlet MVC supports beans whose lifecycle is scoped to the current HTTP request or HTTP `Session` (both normal and global). This is not a specific feature of Spring Portlet MVC itself, but rather of the `WebApplicationContext` container(s) that Spring Portlet MVC uses. These bean scopes are described in detail in Section 5.5.4, “Request, session, and global session scopes”

20.2 The `DispatcherPortlet`

Portlet MVC is a request-driven web MVC framework, designed around a portlet that dispatches requests to controllers and offers other functionality facilitating the development of portlet applications. Spring's `DispatcherPortlet` however, does more than

just that. It is completely integrated with the Spring `ApplicationContext` and allows you to use every other feature Spring has.

Like ordinary portlets, the `DispatcherPortlet` is declared in the `portlet.xml` file of your web application:

```
<portlet>
    <portlet-name>sample</portlet-name>
    <portlet-class>org.springframework.web.portlet.DispatcherPortlet</portlet-class>
    <supports>
        <mime-type>text/html</mime-type>
        <portlet-mode>view</portlet-mode>
    </supports>
    <portlet-info>
        <title>Sample Portlet</title>
    </portlet-info>
</portlet>
```

The `DispatcherPortlet` now needs to be configured.

In the Portlet MVC framework, each `DispatcherPortlet` has its own `WebApplicationContext`, which inherits all the beans already defined in the Root `WebApplicationContext`. These inherited beans can be overridden in the portlet-specific scope, and new scope-specific beans can be defined local to a given portlet instance.

The framework will, on initialization of a `DispatcherPortlet`, look for a file named `[portlet-name]-portlet.xml` in the `WEB-INF` directory of your web application and create the beans defined there (overriding the definitions of any beans defined with the same name in the global scope).

The config location used by the `DispatcherPortlet` can be modified through a portlet initialization parameter (see below for details).

The Spring `DispatcherPortlet` has a few special beans it uses, in order to be able to process requests and render the appropriate views. These beans are included in the Spring framework and can be configured in the `WebApplicationContext`, just as any other bean would be configured. Each of those beans is described in more detail below. Right now, we'll just mention

them, just to let you know they exist and to enable us to go on talking about the `DispatcherPortlet`. For most of the beans, defaults are provided so you don't have to worry about configuring them.

Table 20.1. Special beans in the `WebApplicationContext`

Expression	Explanation
handler mapping(s)	(Section 20.5, “Handler mappings”) a list of pre- and post-processors and controllers that will be executed if they match certain criteria (for instance a matching portlet mode specified with the controller)
controller(s)	(Section 20.4, “Controllers”) the beans providing the actual functionality (or at least, access to the functionality) as part of the MVC triad
view resolver	(Section 20.6, “Views and resolving them”) capable of resolving view names to view definitions
multipart resolver	(Section 20.7, “Multipart (file upload) support”) offers functionality to process file uploads from HTML forms
handler exception resolver	(Section 20.8, “Handling exceptions”) offers functionality to map exceptions to views or implement other more complex exception handling code

When a `DispatcherPortlet` is setup for use and a request comes in for that specific `DispatcherPortlet`, it starts processing the request. The list below describes the complete process a request goes through if handled by a `DispatcherPortlet`:

1. The locale returned by `PortletRequest.getLocale()` is bound to the request to let elements in the process resolve the locale to use when processing the request (rendering the view, preparing data, etc.).
2. If a multipart resolver is specified and this is an `ActionRequest`, the request is inspected for multiparts and if they are found, it is wrapped in a `MultipartActionRequest` for further processing by other elements in the process. (See Section 20.7, “Multipart (file upload) support” for further information about multipart handling).

3. An appropriate handler is searched for. If a handler is found, the execution chain associated with the handler (pre-processors, post-processors, controllers) will be executed in order to prepare a model.
4. If a model is returned, the view is rendered, using the view resolver that has been configured with the `WebApplicationContext`. If no model is returned (which could be due to a pre- or post-processor intercepting the request, for example, for security reasons), no view is rendered, since the request could already have been fulfilled.

Exceptions that are thrown during processing of the request get picked up by any of the handler exception resolvers that are declared in the `WebApplicationContext`. Using these exception resolvers you can define custom behavior in case such exceptions get thrown.

You can customize Spring's `DispatcherPortlet` by adding context parameters in the `portlet.xml` file or portlet init-parameters. The possibilities are listed below.

Table 20.2. `DispatcherPortlet` initialization parameters

Parameter	Explanation
`contextClass`	Class that implements `WebApplicationContext`, which will be used to instantiate the context used by this portlet. If this parameter isn't specified, the `XmlPortletApplicationContext` will be used.
`contextConfigLocation`	String which is passed to the context instance (specified by `contextClass`) to indicate where context(s) can be found. The String is potentially split up into multiple Strings (using a comma as a delimiter) to support multiple contexts (in case of multiple context locations, for beans that are defined twice, the latest takes precedence).
`namespace`	The namespace of the `WebApplicationContext`. Defaults to `[portlet-name]-portlet`.
`viewRendererUrl`	The URL at which `DispatcherPortlet` can access an instance of `ViewRendererServlet` (see Section 20.3, “The `ViewRendererServlet`”).

20.3 The `ViewRendererServlet`

The rendering process in Portlet MVC is a bit more complex than in Web MVC. In order to reuse all the view technologies from Spring Web MVC, we must convert the `PortletRequest` / `PortletResponse` to `HttpServletRequest` / `HttpServletResponse` and then call the `render` method of the `View`. To do this, `DispatcherPortlet` uses a special servlet that exists for just this purpose: the `ViewRendererServlet`.

In order for `DispatcherPortlet` rendering to work, you must declare an instance of the `ViewRendererServlet` in the `web.xml` file for your web application as follows:

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

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

To perform the actual rendering, `DispatcherPortlet` does the following:

1. Binds the `WebApplicationContext` to the request as an attribute under the same `WEB_APPLICATION_CONTEXT_ATTRIBUTE` key that `DispatcherServlet` uses.
2. Binds the `Model` and `View` objects to the request to make them available to the `ViewRendererServlet`.
3. Constructs a `PortletRequestDispatcher` and performs an `include` using the `/WEB- INF/servlet/view` URL that is mapped to the `ViewRendererServlet`.

The `ViewRendererServlet` is then able to call the `render` method on the `View` with the appropriate arguments.

The actual URL for the `ViewRendererServlet` can be changed using `DispatcherPortlet`'s `viewRendererUrl` configuration parameter.

20.4 Controllers

The controllers in Portlet MVC are very similar to the Web MVC Controllers, and porting code from one to the other should be simple.

The basis for the Portlet MVC controller architecture is the `org.springframework.web.portlet.mvc.Controller` interface, which is listed below.

```
public interface Controller {

    /**
     * Process the render request and return a ModelAndView object which the
     * DispatcherPortlet will render.
     */
    ModelAndView handleRenderRequest(RenderRequest request, RenderResponse response)
        throws Exception;

    /**
     * Process the action request. There is nothing to return.
     */
    void handleActionRequest(ActionRequest request, ActionResponse response)

        throws Exception;
}
```

As you can see, the Portlet `Controller` interface requires two methods that handle the two phases of a portlet request: the action request and the render request. The action phase should be capable of handling an action request, and the render phase should be capable of handling a render request and returning an appropriate model and view. While the `Controller` interface is quite abstract, Spring Portlet MVC offers several controllers that already contain a lot of the functionality you might need; most of

these are very similar to controllers from Spring Web MVC. The `Controller` interface just defines the most common functionality required of every controller: handling an action request, handling a render request, and returning a model and a view.

20.4.1 `AbstractController` and `PortletContentGenerator`

Of course, just a `Controller` interface isn't enough. To provide a basic infrastructure, all of Spring Portlet MVC's `Controller`s inherit from `AbstractController`, a class offering access to Spring's `ApplicationContext` and control over caching.

Table 20.3. Features offered by the `AbstractController`

Parameter	Explanation
`requireSession`	Indicates whether or not this `Controller` requires a session to do its work. This feature is offered to all controllers. If a session is not present when such a controller receives a request, the user is informed using a `SessionRequiredException`.
`synchronizeSession`	Use this if you want handling by this controller to be synchronized on the user's session. To be more specific, the extending controller will override the `handleRenderRequestInternal(..)` and `handleActionRequestInternal(..)` methods, which will be synchronized on the user's session if you specify this variable.
`renderWhenMinimized`	If you want your controller to actually render the view when the portlet is in a minimized state, set this to true. By default, this is set to false so that portlets that are in a minimized state don't display any content.
`cacheSeconds`	When you want a controller to override the default cache expiration defined for the portlet, specify a positive integer here. By default it is set to `-1`, which does not change the default caching. Setting it to `0` will ensure the result is never cached.

The `requireSession` and `cacheSeconds` properties are declared on the `PortletContentGenerator` class, which is the superclass of `AbstractController`) but are included here for completeness.

When using the `AbstractController` as a base class for your controllers (which is not recommended since there are a lot of other controllers that might already do the job for you) you only have to override either the `handleActionRequestInternal(ActionRequest, ActionResponse)` method or the `handleRenderRequestInternal(RenderRequest, RenderResponse)` method (or both), implement your logic, and return a `ModelAndView` object (in the case of `handleRenderRequestInternal`).

The default implementations of both `handleActionRequestInternal(..)` and `handleRenderRequestInternal(..)` throw a `PortletException`. This is consistent with the behavior of `GenericPortlet` from the JSR- 168 Specification API. So you only need to override the method that your controller is intended to handle.

Here is short example consisting of a class and a declaration in the web application context.

```
package samples;

import javax.portlet.RenderRequest;
import javax.portlet.RenderResponse;

import org.springframework.web.portlet.mvc.AbstractController;
import org.springframework.web.portlet.ModelAndView;

public class SampleController extends AbstractController {

    public ModelAndView handleRenderRequestInternal(RenderRequest request, RenderResponse response) {
        ModelAndView mav = new ModelAndView("foo");
        mav.addObject("message", "Hello World!");
        return mav;
    }
}

<bean id="sampleController" class="samples.SampleController">
    <property name="cacheSeconds" value="120"/>
```

```
</bean>
```

The class above and the declaration in the web application context is all you need besides setting up a handler mapping (see Section 20.5, “Handler mappings”) to get this very simple controller working.

20.4.2 Other simple controllers

Although you can extend `AbstractController`, Spring Portlet MVC provides a number of concrete implementations which offer functionality that is commonly used in simple MVC applications.

The `ParameterizableViewController` is basically the same as the example above, except for the fact that you can specify the view name that it will return in the web application context (no need to hard-code the view name).

The `PortletModeNameViewController` uses the current mode of the portlet as the view name. So, if your portlet is in View mode (i.e. `PortletMode.VIEW`) then it uses "view" as the view name.

20.4.3 Command Controllers

Spring Portlet MVC has the exact same hierarchy of *command controllers* as Spring Web MVC. They provide a way to interact with data objects and dynamically bind parameters from the `PortletRequest` to the data object specified. Your data objects don't have to implement a framework-specific interface, so you can directly manipulate your persistent objects if you desire. Let's examine what command controllers are available, to get an overview of what you can do with them:

- `AbstractCommandController` - a command controller you can use to create your own command controller, capable of binding request parameters to a data object you specify. This class does not offer form functionality, it does however offer validation features and lets you specify in the controller itself what to do with the command object that has been filled with the parameters from the request.
- `AbstractFormController` - an abstract controller offering form submission support. Using this controller you can model forms and populate them using a command object you retrieve in the controller. After a user has filled the form, `AbstractFormController` binds the fields, validates, and hands the object back to the controller to take appropriate action.

Supported features are: invalid form submission (resubmission), validation, and normal form workflow. You implement methods to determine which views are used for form presentation and success. Use this controller if you need forms, but don't want to specify what views you're going to show the user in the application context.

- `SimpleFormController` - a concrete `AbstractFormController` that provides even more support when creating a form with a corresponding command object. The `SimpleFormController` lets you specify a command object, a viewname for the form, a viewname for the page you want to show the user when form submission has succeeded, and more.
- `AbstractWizardFormController` – a concrete `AbstractFormController` that provides a wizard-style interface for editing the contents of a command object across multiple display pages. Supports multiple user actions: finish, cancel, or page change, all of which are easily specified in request parameters from the view.

These command controllers are quite powerful, but they do require a detailed understanding of how they operate in order to use them efficiently. Carefully review the Javadocs for this entire hierarchy and then look at some sample implementations before you start using them.

20.4.4 `PortletWrappingController`

Instead of developing new controllers, it is possible to use existing portlets and map requests to them from a `DispatcherPortlet`. Using the `PortletWrappingController`, you can instantiate an existing `Portlet` as a `Controller` as follows:

```
<bean id="myPortlet" class="org.springframework.web.portlet.mvc.PortletWrappingController">
    <property name="portletClass" value="sample.MyPortlet"/>
    <property name="portletName" value="my-portlet"/>
    <property name="initParameters">
        <value>config=/WEB-INF/my-portlet-config.xml</value>
    </property>
</bean>
```

This can be very valuable since you can then use interceptors to pre-process and post-process requests going to these portlets. Since JSR-286 does not support any kind of filter mechanism, this is quite handy. For example, this can be used to wrap the

Hibernate `OpenSessionInViewInterceptor` around a MyFaces JSF Portlet.

20.5 Handler mappings

Using a handler mapping you can map incoming portlet requests to appropriate handlers. There are some handler mappings you can use out of the box, for example, the `PortletModeHandlerMapping`, but let's first examine the general concept of a `HandlerMapping`.

Note: We are intentionally using the term “Handler” here instead of “Controller”. `DispatcherPortlet` is designed to be used with other ways to process requests than just Spring Portlet MVC’s own Controllers. A Handler is any Object that can handle portlet requests. Controllers are an example of Handlers, and they are of course the default. To use some other framework with `DispatcherPortlet`, a corresponding implementation of `HandlerAdapter` is all that is needed.

The functionality a basic `HandlerMapping` provides is the delivering of a `HandlerExecutionChain`, which must contain the handler that matches the incoming request, and may also contain a list of handler interceptors that are applied to the request. When a request comes in, the `DispatcherPortlet` will hand it over to the handler mapping to let it inspect the request and come up with an appropriate `HandlerExecutionChain`. Then the `DispatcherPortlet` will execute the handler and interceptors in the chain (if any). These concepts are all exactly the same as in Spring Web MVC.

The concept of configurable handler mappings that can optionally contain interceptors (executed before or after the actual handler was executed, or both) is extremely powerful. A lot of supporting functionality can be built into a custom `HandlerMapping`. Think of a custom handler mapping that chooses a handler not only based on the portlet mode of the request coming in, but also on a specific state of the session associated with the request.

In Spring Web MVC, handler mappings are commonly based on URLs. Since there is really no such thing as a URL within a Portlet, we must use other mechanisms to control mappings. The two most common are the portlet mode and a request parameter, but anything available to the portlet request can be used in a custom handler mapping.

The rest of this section describes three of Spring Portlet MVC's most commonly used handler mappings. They all extend `AbstractHandlerMapping` and share the following properties:

- `interceptors`: The list of interceptors to use. `HandlerInterceptor`s are discussed in Section 20.5.4, "Adding `HandlerInterceptor`s".
- `defaultHandler`: The default handler to use, when this handler mapping does not result in a matching handler.
- `order`: Based on the value of the order property (see the `org.springframework.core.Ordered` interface), Spring will sort all handler mappings available in the context and apply the first matching handler.
- `lazyInitHandlers`: Allows for lazy initialization of singleton handlers (prototype handlers are always lazily initialized). Default value is false. This property is directly implemented in the three concrete Handlers.

20.5.1 `PortletModeHandlerMapping`

This is a simple handler mapping that maps incoming requests based on the current mode of the portlet (e.g. 'view', 'edit', 'help'). An example:

```
<bean class="org.springframework.web.portlet.handler.PortletModeHandlerMapping">
    <property name="portletModeMap">
        <map>
            <entry key="view" value-ref="viewHandler"/>
            <entry key="edit" value-ref="editHandler"/>
            <entry key="help" value-ref="helpHandler"/>
        </map>
    </property>
</bean>
```

20.5.2 `ParameterHandlerMapping`

If we need to navigate around to multiple controllers without changing portlet mode, the simplest way to do this is with a request parameter that is used as the key to control the mapping.

`ParameterHandlerMapping` uses the value of a specific request parameter to control the mapping. The default name of the parameter is `'action'`, but can be changed using the `'parameterName'` property.

The bean configuration for this mapping will look something like this:

```
<bean class="org.springframework.web.portlet.handler.ParameterHandlerMapping”>
    <property name="parameterMap">
        <map>
            <entry key="add" value-ref="addItemHandler"/>
            <entry key="edit" value-ref="editItemHandler"/>
            <entry key="delete" value-ref="deleteItemHandler"/>
        </map>
    </property>
</bean>
```

20.5.3 `PortletModeParameterHandlerMapping`

The most powerful built-in handler mapping, `PortletModeParameterHandlerMapping` combines the capabilities of the two previous ones to allow different navigation within each portlet mode.

Again the default name of the parameter is "action", but can be changed using the `parameterName` property.

By default, the same parameter value may not be used in two different portlet modes. This is so that if the portal itself changes the portlet mode, the request will no longer be valid in the mapping. This behavior can be changed by setting the `allowDupParameters` property to true. However, this is not recommended.

The bean configuration for this mapping will look something like this:

```
<bean class="org.springframework.web.portlet.handler.PortletModeParameterHandlerMapping">
    <property name="portletModeParameterMap">
        <map>
            <entry key="view"> <!-- 'view' portlet mode -->
                <map>
                    <entry key="add" value-ref="addItemHandler"/>
                    <entry key="edit" value-ref="editItemHandler"/>
                    <entry key="delete" value-ref="deleteItemHandler"/>
```

```
                </map>
            </entry>
            <entry key="edit"> <!-- 'edit' portlet mode -->
                <map>
                    <entry key="prefs" value-ref="prefsHandler"/>
                    <entry key="resetPrefs" value-ref="resetPrefsHandler"/>
                </map>
            </entry>
        </map>
    </property>
</bean>
```

This mapping can be chained ahead of a `PortletModeHandlerMapping`, which can then provide defaults for each mode and an overall default as well.

20.5.4 Adding `HandlerInterceptor`s

Spring's handler mapping mechanism has a notion of handler interceptors, which can be extremely useful when you want to apply specific functionality to certain requests, for example, checking for a principal. Again Spring Portlet MVC implements these concepts in the same way as Web MVC.

Interceptors located in the handler mapping must implement `HandlerInterceptor` from the `org.springframework.web.portlet` package. Just like the servlet version, this interface defines three methods: one that will be called before the actual handler will be executed (`preHandle`), one that will be called after the handler is executed (`postHandle`), and one that is called after the complete request has finished (`afterCompletion`). These three methods should provide enough flexibility to do all kinds of pre- and post- processing.

The `preHandle` method returns a boolean value. You can use this method to break or continue the processing of the execution chain. When this method returns `true`, the handler execution chain will continue. When it returns `false`, the `DispatcherPortlet` assumes the interceptor itself has taken care of requests (and, for example, rendered an appropriate view) and does not continue executing the other interceptors and the actual handler in the execution chain.

The `postHandle` method is only called on a `RenderRequest`. The `preHandle` and `afterCompletion` methods are called on both an `ActionRequest` and a `RenderRequest`. If you need to execute logic in these methods for just one type of request, be sure to check what kind of request it is before processing it.

20.5.5 `HandlerInterceptorAdapter`

As with the servlet package, the portlet package has a concrete implementation of `HandlerInterceptor` called `HandlerInterceptorAdapter`. This class has empty versions of all the methods so that you can inherit from this class and implement just one or two methods when that is all you need.

20.5.6 `ParameterMappingInterceptor`

The portlet package also has a concrete interceptor named `ParameterMappingInterceptor` that is meant to be used directly with `ParameterHandlerMapping` and `PortletModeParameterHandlerMapping`. This interceptor will cause the parameter that is being used to control the mapping to be forwarded from an `ActionRequest` to the subsequent `RenderRequest`. This will help ensure that the `RenderRequest` is mapped to the same Handler as the `ActionRequest`. This is done in the `preHandle` method of the interceptor, so you can still modify the parameter value in your handler to change where the `RenderRequest` will be mapped.

Be aware that this interceptor is calling `setRenderParameter` on the `ActionResponse`, which means that you cannot call `sendRedirect` in your handler when using this interceptor. If you need to do external redirects then you will either need to forward the mapping parameter manually or write a different interceptor to handle this for you.

20.6 Views and resolving them

As mentioned previously, Spring Portlet MVC directly reuses all the view technologies from Spring Web MVC. This includes not only the various `View` implementations themselves, but also the `ViewResolver` implementations. For more information, refer to Chapter 18, *View technologies* and Section 17.5, “Resolving views” respectively.

A few items on using the existing `View` and `ViewResolver` implementations are worth mentioning:

- Most portals expect the result of rendering a portlet to be an HTML fragment. So, things like JSP/JSTL, Velocity, FreeMarker, and XSLT all make sense. But it is unlikely that views that return other document types will make any sense in a portlet context.
- There is no such thing as an HTTP redirect from within a portlet (the `sendRedirect(..)` method of `ActionResponse` cannot be used to stay within the portal). So, `RedirectView` and use of the `'redirect:'` prefix will **not** work correctly from within Portlet MVC.
- It may be possible to use the `'forward:'` prefix from within Portlet MVC. However, remember that since you are in a portlet, you have no idea what the current URL looks like. This means you cannot use a relative URL to access other resources in your web application and that you will have to use an absolute URL.

Also, for JSP development, the new Spring Taglib and the new Spring Form Taglib both work in portlet views in exactly the same way that they work in servlet views.

20.7 Multipart (file upload) support

Spring Portlet MVC has built-in multipart support to handle file uploads in portlet applications, just like Web MVC does. The design for the multipart support is done with pluggable `PortletMultipartResolver` objects, defined in the `org.springframework.web.portlet.multipart` package. Spring provides a `PortletMultipartResolver` for use with Commons FileUpload. How uploading files is supported will be described in the rest of this section.

By default, no multipart handling will be done by Spring Portlet MVC, as some developers will want to handle multiparts themselves. You will have to enable it yourself by adding a multipart resolver to the web application's context. After you have done that, `DispatcherPortlet` will inspect each request to see if it contains a multipart. If no multipart is found, the request will continue as expected. However, if a multipart is found in the request, the `PortletMultipartResolver` that has been declared in your context will be used. After that, the multipart attribute in your request will be treated like any other attribute.

Any configured `PortletMultipartResolver` bean *must* have the following id (or name):

> "`portletMultipartResolver`". If you have defined your `PortletMultipartResolver` with any other name, then the `DispatcherPortlet` will *not* find your `PortletMultipartResolver`, and consequently no multipart support will be in effect.

20.7.1 Using the `PortletMultipartResolver`

The following example shows how to use the `CommonsPortletMultipartResolver`:

```
<bean id="portletMultipartResolver"
        class="org.springframework.web.portlet.multipart.CommonsPortletMultipartResolver">

    <!-- one of the properties available; the maximum file size in bytes -->
    <property name="maxUploadSize" value="100000"/>
</bean>
```

Of course you also need to put the appropriate jars in your classpath for the multipart resolver to work. In the case of the `CommonsMultipartResolver`, you need to use `commons-fileupload.jar`. Be sure to use at least version 1.1 of Commons FileUpload as previous versions do not support JSR-286 Portlet applications.

Now that you have seen how to set Portlet MVC up to handle multipart requests, let's talk about how to actually use it. When `DispatcherPortlet` detects a multipart request, it activates the resolver that has been declared in your context and hands over the request. What the resolver then does is wrap the current `ActionRequest` in a `MultipartActionRequest` that has support for multipart file uploads. Using the `MultipartActionRequest` you can get information about the multiparts contained by this request and actually get access to the multipart files themselves in your controllers.

Note that you can only receive multipart file uploads as part of an `ActionRequest`, not as part of a `RenderRequest`.

20.7.2 Handling a file upload in a form

After the `PortletMultipartResolver` has finished doing its job, the request will be processed like any other. To use the `PortletMultipartResolver`, create a form with an upload field (see example below), then let Spring bind the file onto your form (backing object). To actually let the user upload a file, we have to create a (JSP/HTML) form:

```
<h1>Please upload a file</h1>
<form method="post" action="<portlet:actionURL/>" enctype="multipart/form-data">
    <input type="file" name="file"/>
    <input type="submit"/>
</form>
```

As you can see, we've created a field named “file” that matches the property of the bean that holds the `byte[]` array. Furthermore we've added the encoding attribute (`enctype="multipart/form-data"`), which is necessary to let the browser know how to encode the multipart fields (do not forget this!).

Just as with any other property that's not automagically convertible to a string or primitive type, to be able to put binary data in your objects you have to register a custom editor with the `PortletRequestDataBinder`. There are a couple of editors available for handling files and setting the results on an object. There's a `StringMultipartFileEditor` capable of converting files to Strings (using a user-defined character set), and there is a `ByteArrayMultipartFileEditor` which converts files to byte arrays. They function analogous to the `CustomDateEditor`.

So, to be able to upload files using a form, declare the resolver, a mapping to a controller that will process the bean, and the controller itself.

```
<bean id="portletMultipartResolver"
        class="org.springframework.web.portlet.multipart.CommonsPortletMultipartResolver"/>

<bean class="org.springframework.web.portlet.handler.PortletModeHandlerMapping">
    <property name="portletModeMap">
        <map>
            <entry key="view" value-ref="fileUploadController"/>
        </map>
    </property>
</bean>
```

```
<bean id="fileUploadController" class="examples.FileUploadController">
    <property name="commandClass" value="examples.FileUploadBean"/>
    <property name="formView" value="fileuploadform"/>
    <property name="successView" value="confirmation"/>
</bean>
```

After that, create the controller and the actual class to hold the file property.

```
public class FileUploadController extends SimpleFormController {

    public void onSubmitAction(ActionRequest request, ActionResponse response,
            Object command, BindException errors) throws Exception {

        // cast the bean
        FileUploadBean bean = (FileUploadBean) command;

        // let's see if there's content there
        byte[] file = bean.getFile();
        if (file == null) {
            // hmm, that's strange, the user did not upload anything
        }


        // do something with the file here
    }

    protected void initBinder(
            PortletRequest request, PortletRequestDataBinder binder) throws Exception {
        // to actually be able to convert Multipart instance to byte[]
        // we have to register a custom editor
        binder.registerCustomEditor(byte[].class, new ByteArrayMultipartFileEditor());
        // now Spring knows how to handle multipart object and convert
    }
}
```

```
public class FileUploadBean {

    private byte[] file;

    public void setFile(byte[] file) {
        this.file = file;
    }

    public byte[] getFile() {
        return file;
    }
}
```

As you can see, the `FileUploadBean` has a property of type `byte[]` that holds the file. The controller registers a custom editor to let Spring know how to actually convert the multipart objects the resolver has found to properties specified by the bean. In this example, nothing is done with the `byte[]` property of the bean itself, but in practice you can do whatever you want (save it in a database, mail it to somebody, etc).

An equivalent example in which a file is bound straight to a String-typed property on a form backing object might look like this:

```
public class FileUploadController extends SimpleFormController {

    public void onSubmitAction(ActionRequest request, ActionResponse response,
            Object command, BindException errors) throws Exception {

        // cast the bean
        FileUploadBean bean = (FileUploadBean) command;

        // let's see if there's content there
        String file = bean.getFile();
        if (file == null) {
            // hmm, that's strange, the user did not upload anything
        }
```

```
        // do something with the file here
    }

    protected void initBinder(
        PortletRequest request, PortletRequestDataBinder binder) throws Exception {

        // to actually be able to convert Multipart instance to a String
        // we have to register a custom editor
        binder.registerCustomEditor(String.class,
            new StringMultipartFileEditor());

        // now Spring knows how to handle multipart objects and convert
    }
}

public class FileUploadBean {

    private String file;

    public void setFile(String file) {
        this.file = file;
    }

    public String getFile() {
        return file;
    }
}
```

Of course, this last example only makes (logical) sense in the context of uploading a plain text file (it wouldn't work so well in the case of uploading an image file).

The third (and final) option is where one binds directly to a `MultipartFile` property declared on the (form backing) object's class. In this case one does not need to register any custom property editor because there is no type conversion to be performed.

```
public class FileUploadController extends SimpleFormController {
```

```
    public void onSubmitAction(ActionRequest request, ActionResponse response,
            Object command, BindException errors) throws Exception {

        // cast the bean
        FileUploadBean bean = (FileUploadBean) command;

        // let's see if there's content there
        MultipartFile file = bean.getFile();
        if (file == null) {
            // hmm, that's strange, the user did not upload anything
        }

        // do something with the file here
    }
}

public class FileUploadBean {

    private MultipartFile file;

    public void setFile(MultipartFile file) {
        this.file = file;
    }

    public MultipartFile getFile() {
        return file;
    }
}
```

20.8 Handling exceptions

Just like Servlet MVC, Portlet MVC provides `HandlerExceptionResolver`s to ease the pain of unexpected exceptions that occur while your request is being processed by a handler that matched the request. Portlet MVC also provides a portlet-specific, concrete `SimpleMappingExceptionResolver` that enables you to take the class name of any exception that might be thrown and map it to a view name.

20.9 Annotation-based controller configuration

Spring 2.5 introduced an annotation-based programming model for MVC controllers, using annotations such as `@RequestMapping`, `@RequestParam`, `@ModelAttribute`, etc. This annotation support is available for both Servlet MVC and Portlet MVC. Controllers implemented in this style do not have to extend specific base classes or implement specific interfaces. Furthermore, they do not usually have direct dependencies on Servlet or Portlet API's, although they can easily get access to Servlet or Portlet facilities if desired.

The following sections document these annotations and how they are most commonly used in a Portlet environment.

20.9.1 Setting up the dispatcher for annotation support

`@RequestMapping` will only be processed if a corresponding `HandlerMapping` (for type level annotations) and/or `HandlerAdapter` (for method level annotations) is present in the dispatcher. This is the case by default in both `DispatcherServlet` and `DispatcherPortlet`.

However, if you are defining custom `HandlerMappings` or `HandlerAdapters`, then you need to make sure that a corresponding custom `DefaultAnnotationHandlerMapping` and/or `AnnotationMethodHandlerAdapter` is defined as well - provided that you intend to use `@RequestMapping`.

```
<?xml version="1.0" encoding="UTF-8"?>
<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.xsd">
```

```
    <bean class="org.springframework.web.portlet.mvc.annotation.DefaultAnnotationHandlerMapping"/>

    <bean class="org.springframework.web.portlet.mvc.annotation.AnnotationMethodHandlerAdapter"/>

    // ... (controller bean definitions) ...

</beans>
```

Defining a `DefaultAnnotationHandlerMapping` and/or `AnnotationMethodHandlerAdapter` explicitly also makes sense if you would like to customize the mapping strategy, e.g. specifying a custom `WebBindingInitializer` (see below).

20.9.2 Defining a controller with `@Controller`

The `@Controller` annotation indicates that a particular class serves the role of a *controller*. There is no need to extend any controller base class or reference the Portlet API. You are of course still able to reference Portlet-specific features if you need to.

The basic purpose of the `@Controller` annotation is to act as a stereotype for the annotated class, indicating its role. The dispatcher will scan such annotated classes for mapped methods, detecting `@RequestMapping` annotations (see the next section).

Annotated controller beans may be defined explicitly, using a standard Spring bean definition in the dispatcher's context. However, the `@Controller` stereotype also allows for autodetection, aligned with Spring 2.5's general support for detecting component classes in the classpath and auto-registering bean definitions for them.

To enable autodetection of such annotated controllers, you have to add component scanning to your configuration. This is easily achieved by using the *spring-context* schema as shown in the following XML snippet:

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

```
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd">

    <context:component-scan base-package="org.springframework.samples.petportal.portlet"/>

    // ...

</beans>
```

20.9.3 Mapping requests with `@RequestMapping`

The `@RequestMapping` annotation is used to map portlet modes like 'VIEW'/'EDIT' onto an entire class or a particular handler method. Typically the type-level annotation maps a specific mode (or mode plus parameter condition) onto a form controller, with additional method-level annotations 'narrowing' the primary mapping for specific portlet request parameters.

`@RequestMapping` at the type level may be used for plain implementations of the `Controller` interface as well. In this case, the request processing code would follow the traditional `handle(Action|Render)Request` signature, while the controller's mapping would be expressed through an `@RequestMapping` annotation. This works for pre-built `Controller` base classes, such as `SimpleFormController`, too.
In the following discussion, we'll focus on controllers that are based on annotated handler methods.

The following is an example of a form controller from the PetPortal sample application using this annotation:

```
@Controller
@RequestMapping("EDIT")
@SessionAttributes("site")
public class PetSitesEditController {
```

```
    private Properties petSites;

    public void setPetSites(Properties petSites) {
        this.petSites = petSites;
    }

    @ModelAttribute("petSites")
    public Properties getPetSites() {
        return this.petSites;
    }

    @RequestMapping  // default (action=list)
    public String showPetSites() {
        return "petSitesEdit";
    }

    @RequestMapping(params = "action=add")  // render phase
    public String showSiteForm(Model model) {
        // Used for the initial form as well as for redisplaying with errors.
        if (!model.containsAttribute("site")) {
            model.addAttribute("site", new PetSite());
        }
        return "petSitesAdd";
    }

    @RequestMapping(params = "action=add")  // action phase
    public void populateSite(
            @ModelAttribute("site") PetSite petSite, BindingResult result,
            SessionStatus status, ActionResponse response) {

        new PetSiteValidator().validate(petSite, result);
        if (!result.hasErrors()) {
            this.petSites.put(petSite.getName(), petSite.getUrl());
            status.setComplete();
            response.setRenderParameter("action", "list");
```

```
        }
    }

    @RequestMapping(params = "action=delete")
    public void removeSite(@RequestParam("site") String site, ActionResponse response) {
        this.petSites.remove(site);
        response.setRenderParameter("action", "list");
    }
}
```

20.9.4 Supported handler method arguments

Handler methods which are annotated with `@RequestMapping` are allowed to have very flexible signatures. They may have arguments of the following types, in arbitrary order (except for validation results, which need to follow right after the corresponding command object, if desired):

- Request and/or response objects (Portlet API). You may choose any specific request/response type, e.g. PortletRequest / ActionRequest / RenderRequest. An explicitly declared action/render argument is also used for mapping specific request types onto a handler method (in case of no other information given that differentiates between action and render requests).
- Session object (Portlet API): of type PortletSession. An argument of this type will enforce the presence of a corresponding session. As a consequence, such an argument will never be `null`.
- `org.springframework.web.context.request.WebRequest` or `org.springframework.web.context.request.NativeWebRequest`. Allows for generic request parameter access as well as request/session attribute access, without ties to the native Servlet/Portlet API.
- `java.util.Locale` for the current request locale (the portal locale in a Portlet environment).
- `java.io.InputStream` / `java.io.Reader` for access to the request's content. This will be the raw InputStream/Reader as exposed by the Portlet API.
- `java.io.OutputStream` / `java.io.Writer` for generating the response's content. This will be the raw OutputStream/Writer as exposed by the Portlet API.

- `@RequestParam` annotated parameters for access to specific Portlet request parameters. Parameter values will be converted to the declared method argument type.
- `java.util.Map` / `org.springframework.ui.Model` / `org.springframework.ui.ModelMap` for enriching the implicit model that will be exposed to the web view.
- Command/form objects to bind parameters to: as bean properties or fields, with customizable type conversion, depending on `@InitBinder` methods and/or the HandlerAdapter configuration - see the "`webBindingInitializer`" property on `AnnotationMethodHandlerAdapter`. Such command objects along with their validation results will be exposed as model attributes, by default using the non-qualified command class name in property notation (e.g. "orderAddress" for type "mypackage.OrderAddress"). Specify a parameter-level `ModelAttribute` annotation for declaring a specific model attribute name.
- `org.springframework.validation.Errors` / `org.springframework.validation.BindingResult` validation results for a preceding command/form object (the immediate preceding argument).
- `org.springframework.web.bind.support.SessionStatus` status handle for marking form processing as complete (triggering the cleanup of session attributes that have been indicated by the `@SessionAttributes` annotation at the handler type level).

The following return types are supported for handler methods:

- A `ModelAndView` object, with the model implicitly enriched with command objects and the results of `@ModelAttribute` annotated reference data accessor methods.
- A `Model` object, with the view name implicitly determined through a `RequestToViewNameTranslator` and the model implicitly enriched with command objects and the results of `@ModelAttribute` annotated reference data accessor methods.
- A `Map` object for exposing a model, with the view name implicitly determined through a `RequestToViewNameTranslator` and the model implicitly enriched with command objects and the results of `@ModelAttribute` annotated reference data accessor methods.
- A `View` object, with the model implicitly determined through command objects and `@ModelAttribute` annotated reference data accessor methods. The handler method may also programmatically enrich the model by declaring a `Model` argument (see above).
- A `String` value which is interpreted as view name, with the model implicitly determined through command objects and `@ModelAttribute` annotated reference data accessor methods. The handler method may also programmatically enrich the

model by declaring a `Model` argument (see above).

- `void` if the method handles the response itself (e.g. by writing the response content directly).
- Any other return type will be considered a single model attribute to be exposed to the view, using the attribute name specified through `@ModelAttribute` at the method level (or the default attribute name based on the return type's class name otherwise). The model will be implicitly enriched with command objects and the results of `@ModelAttribute` annotated reference data accessor methods.

20.9.5 Binding request parameters to method parameters with `@RequestParam`

The `@RequestParam` annotation is used to bind request parameters to a method parameter in your controller.

The following code snippet from the PetPortal sample application shows the usage:

```
@Controller
@RequestMapping("EDIT")
@SessionAttributes("site")
public class PetSitesEditController {

    // ...

    public void removeSite(@RequestParam("site") String site, ActionResponse response) {
        this.petSites.remove(site);
        response.setRenderParameter("action", "list");
    }

    // ...
}
```

Parameters using this annotation are required by default, but you can specify that a parameter is optional by setting `@RequestParam`'s `required` attribute to `false` (e.g., `@RequestParam(value="id", required=false)`).

20.9.6 Providing a link to data from the model with `@ModelAttribute`

`@ModelAttribute` has two usage scenarios in controllers. When placed on a method parameter, `@ModelAttribute` is used to map a model attribute to the specific, annotated method parameter (see the `populateSite()` method below). This is how the controller gets a reference to the object holding the data entered in the form. In addition, the parameter can be declared as the specific type of the form backing object rather than as a generic `java.lang.Object`, thus increasing type safety.

`@ModelAttribute` is also used at the method level to provide *reference data* for the model (see the `getPetSites()` method below). For this usage the method signature can contain the same types as documented above for the `@RequestMapping` annotation.

Note: `@ModelAttribute` annotated methods will be executed *before* the chosen `@RequestMapping` annotated handler method. They effectively pre-populate the implicit model with specific attributes, often loaded from a database. Such an attribute can then already be accessed through `@ModelAttribute` annotated handler method parameters in the chosen handler method, potentially with binding and validation applied to it.

The following code snippet shows these two usages of this annotation:

```
@Controller
@RequestMapping("EDIT")
@SessionAttributes("site")
public class PetSitesEditController {

    // ...

    @ModelAttribute("petSites")
    public Properties getPetSites() {
        return this.petSites;
    }

    @RequestMapping(params = "action=add")  // action phase
    public void populateSite(
            @ModelAttribute("site") PetSite petSite, BindingResult result,
```

```
            SessionStatus status, ActionResponse response) {

        new PetSiteValidator().validate(petSite, result);
        if (!result.hasErrors()) {
            this.petSites.put(petSite.getName(), petSite.getUrl());
            status.setComplete();
            response.setRenderParameter("action", "list");
        }
    }
}
```

20.9.7 Specifying attributes to store in a Session with `@SessionAttributes`

The type-level `@SessionAttributes` annotation declares session attributes used by a specific handler. This will typically list the names of model attributes or types of model attributes which should be transparently stored in the session or some conversational storage, serving as form-backing beans between subsequent requests.

The following code snippet shows the usage of this annotation:

```
@Controller
@RequestMapping("EDIT")
@SessionAttributes("site")
public class PetSitesEditController {
  // ...
}
```

20.9.8 Customizing `WebDataBinder` initialization

To customize request parameter binding with PropertyEditors, etc. via Spring's `WebDataBinder`, you can either use `@InitBinder`-annotated methods within your controller or externalize your configuration by providing a custom `WebBindingInitializer`.

Customizing data binding with `@InitBinder`

Annotating controller methods with `@InitBinder` allows you to configure web data binding directly within your controller class. `@InitBinder` identifies methods which initialize the `WebDataBinder` which will be used for populating command and form object arguments of annotated handler methods.

Such init-binder methods support all arguments that `@RequestMapping` supports, except for command/form objects and corresponding validation result objects. Init-binder methods must not have a return value. Thus, they are usually declared as `void`. Typical arguments include `WebDataBinder` in combination with `WebRequest` or `java.util.Locale`, allowing code to register context-specific editors.

The following example demonstrates the use of `@InitBinder` for configuring a `CustomDateEditor` for all `java.util.Date` form properties.

```
@Controller
public class MyFormController {

    @InitBinder
    public void initBinder(WebDataBinder binder) {

        SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
        dateFormat.setLenient(false);
        binder.registerCustomEditor(Date.class, new CustomDateEditor(dateFormat, false));
    }

    // ...
}
```

Configuring a custom `WebBindingInitializer`

To externalize data binding initialization, you can provide a custom implementation of the `WebBindingInitializer` interface, which you then enable by supplying a custom bean configuration for an `AnnotationMethodHandlerAdapter`, thus overriding the default configuration.

20.10 Portlet application deployment

The process of deploying a Spring Portlet MVC application is no different than deploying any JSR-286 Portlet application. However, this area is confusing enough in general that it is worth talking about here briefly.

Generally, the portal/portlet container runs in one webapp in your servlet container and your portlets run in another webapp in your servlet container. In order for the portlet container webapp to make calls into your portlet webapp it must make cross-context calls to a well-known servlet that provides access to the portlet services defined in your `portlet.xml` file.

The JSR-286 specification does not specify exactly how this should happen, so each portlet container has its own mechanism for this, which usually involves some kind of “deployment process” that makes changes to the portlet webapp itself and then registers the portlets within the portlet container.

At a minimum, the `web.xml` file in your portlet webapp is modified to inject the well-known servlet that the portlet container will call. In some cases a single servlet will service all portlets in the webapp, in other cases there will be an instance of the servlet for each portlet.

Some portlet containers will also inject libraries and/or configuration files into the webapp as well. The portlet container must also make its implementation of the Portlet JSP Tag Library available to your webapp.

The bottom line is that it is important to understand the deployment needs of your target portal and make sure they are met (usually by following the automated deployment process it provides). Be sure to carefully review the documentation from your portal for this process.

Once you have deployed your portlet, review the resulting `web.xml` file for sanity. Some older portals have been known to corrupt the definition of the `ViewRendererServlet`, thus breaking the rendering of your portlets.